AN INTELLIGENT DYNAMIC SIMULATION ENVIRONMENT: AN OBJECT-ORIENTED APPROACH *

J. T. Robinson and R. A. Kisner**

CONF-880899--3

DE88 016362

Engineering Physics and Mathematics Division
Oak Ridge National Laboratory
P.O. Box 2008
Oak Ridge, TN 37831-6364

**Instrumentation and Controls Divsion
Oak Ridge National Laboratory
P.O. Box 2008
Oak Ridge, TN 37831-6008

Paper submitted to: Third International Symposium on Intelligent Control, Arlington, VA, August 24-26, 1988

* Research sponsored by the Advanced Controls Program of the Office of Reactor Technologies Development, of the U.S. Department of Energy, under contract No. DE-AC05-84OR21400 with Martin Marietta Energy Systems, Inc.

AN INTELLIGENT DYNAMIC SIMULATION ENVIRONMENT: AN OBJECT-ORIENTED APPROACH *

J. T. Robinson and R. A. Kisner**

Engineering Physics and Mathematics Division
Oak Ridge National Laboratory
P.O. Box 2008
Oak Ridge, TN 37831-6364

**Instrumentation and Controls Divsion
Oak Ridge National Laboratory
P.O. Box 2008
Oak Ridge, TN 37831-6008

ABSTRACT

This paper presents a prototype simulation environment for nuclear power plants which illustrates the application of object-oriented programming to process simulation. Systems are modeled using this technique as a collection of objects which communicate via message passing. The environment allows users to build simulation models by selecting iconic representations of plant components from a menu and connecting them with the aid of a mouse. Models can be modified graphically at any time, even as the simulation is running, and the results observed immediately via real-time graphics. This prototype illustrates the use of object-oriented programming to create a highly interactive and automated simulation environment.

INTRODUCTION

The advent of very powerful workstation class computers is making possible the application of new tools borrowed from the field of artificial intelligence to more traditional engineering tasks such as simulation. One such tool that has received a lot of attention lately is object-oriented programming [1,2]. Object-oriented programming is based on a hierarchical classification of procedures and data that allows the modeling of systems in terms of collections of objects. Objects may be grouped into classes which contain generic descriptions of state and behavior, which in turn can be created from existing classes by inheritance. Overruling mechanisms permit control of the mixing of parameters and procedures for the new composite class. The parameters describing the state of an object are referred to as its "instance variables." Objects communicate with each other by passing messages. The response of a class of objects to a message is governed by a procedure referred to as a "method."

Previous work has demonstrated the applicability of object-oriented programming to discrete-event simulations and rule-based simulations [3,4,5,6]. In this paper, we focus on its applicability to continuous process simulation by describing a simulation environment developed for power plants. A description of the environment will be followed by a discussion of some advantages of an object-oriented approach to process simulation.

DESCRIPTION OF THE SIMULATION ENVIRONMENT

The simulation environment has been developed on a LMI Lisp machine using the FLAVORS object-oriented language [7]. It consists of a library of class definitions and a user interface. The class library contains the generic descriptions of objects which define the domain of the simulation environment. The user interface provides the means to build, run, and interact with simulation models using interactive graphics.

Class Library

At the core of the simulation package is a library of class definitions and associated methods which define the response of members of the class to particular messages. The classes can be grouped into four categories:

(1) component-level objects,

(2) system-level objects,

(3) hybrid objects, and

(4) user interface and simulation control objects.

Component-level objects. Component-level objects are the basic building blocks from which simulation models are constructed. They may

represent either actual components in the plant, such as pumps, pipes, and valves; or be abstractions such as heat sources and transport delays. Objects from this category have methods for computing their gross averaged physical properties, hydraulic, and/or heat transfer characteristics. The methods are localized, that is, they do not describe how the objects interact with neighbors. This information is contained in the methods of system level objects.

Component-level objects are connected to form systems via their ports. Ports are classified by type and dimension, (e.g. fluid flow in m3/s) and may have a nominal direction (e.g. input or output). Ports are further divided into those accepting single or multiple connections.

As an example component-level class consider FLOW-CONTROL-VALVE. This class has the following instance variables:

inlet-connection
outlet-connection
hydraulic-system
cvmax
valve-position
flow-rate
pressure-drop

The variables *inlet-connection* and *outlet-connection* specify the classes' ports and are used to record the names of connected objects. A non-nil value for *hydraulic-system* indicates that a flow-control-valve is a component of a system level object. *Cvmax* is a parameter related to the size of a valve. The last three instance variables are dynamic in nature and specify the instantaneous state of the valve. The methods for FLOW-CONTROL-VALVE include accessor functions for all instance variables as well as *:compute-flow-rate* and *:compute-pressure-drop.*

System-level objects. For simulating a collection of tightly coupled objects it is not enough to describe their isolated behavior or even their behavior in relation to their immediate neighbors. To achieve simulations which are faithful and numerically stable the notion of a system is required. We have therefore introduced classes of system-level objects which are used for defining operations on groups of related objects. We have been careful in our treatment of systems not to destroy the modularity of the object-oriented approach. In particular, system parameters are not frozen at some initial state but are defined by reference to component-level objects. Thus future changes in a component-level object are immediately reflected in its associated system level object(s).

For illustration we describe the class THERMAL-SYSTEM which solves the energy conservation equations (heat conduction and convection) for groups of thermally connected objects. These equations are represented in the form

$$\frac{d\vec{T}}{dt} = \bar{A}(t)\,\vec{T} + \vec{f}(t)$$

where $\vec{T}$ represents the vector of temperatures, $\bar{A}(t)$ is a time-varying coefficient matrix, and $\vec{f}(t)$ is a time varying forcing function. The instance variables of class THERMAL-SYSTEM include:

components
A-matrix
f-vector, and
T-vector.

At instantiation an object of class THERMAL-SYSTEM automatically creates and initializes the matrix $\bar{A}$ and vectors $\vec{f}$ and $\vec{T}$ of the appropriate dimensions and assigns a row index number to each component. At each time step the THERMAL-SYSTEM object sends messages to all components instructing them to update their slot(s) in the coefficient matrix $\bar{A}$ or forcing vector $\vec{f}$, and then advances the resulting equation system one time step.

It should be noted that the user does not have to deal directly with system-level objects. They are automatically constructed prior to a simulation run based on the connections between component-level objects. This category of objects exists solely as a means of implementing efficient and numerically stable algorithms.

Hybrid objects. The descriptions above implies a two-level hierarchy of objects used to model systems, component-level and system-level objects. This is a bit over-simplified as many classes actually function as both component-level and system-level objects. An example is the HYDRAULIC-NETWORK class which represents parallel flow networks such as that illustrated in figure 1. The function of this class is to return a pressure drop or flow rate for the network given the complementary parameter. This object functions as both a system level object, coordinating the solution of the hydraulic equations among its components, and as a component-level object, functioning as a flow resistance in a larger hydraulic loop.

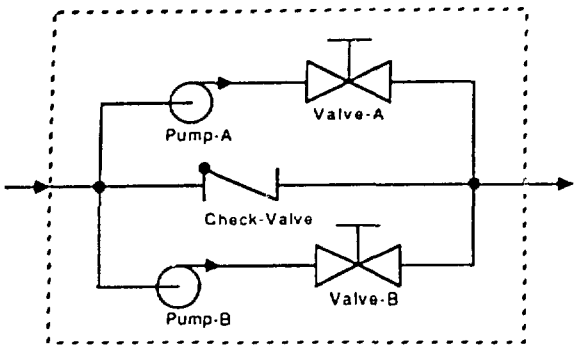

Figure 1. Example hydraulic network object.

Interface and Simulation Control. The final category of class definitions consists of user interface and simulation control objects. Examples include the MODEL-MANAGER class, object editors for input, strip-charts for output, and a system clock. There is always exactly one MODEL-MANAGER object per model which keeps a table of all objects and their connections, manages the windows, and provides a file system interface. Object editors are interactive forms which provide a convenient means for accessing and changing the instance variables of individual objects. An example object editor for a pipe is illustrated in figure 2. Strip charts provide continuously updated plots of chosen parameters during a simulation run. The system clock contains the current simulation time and the time step of integration. This object provides a means for adjusting the time step size during a simulation run.

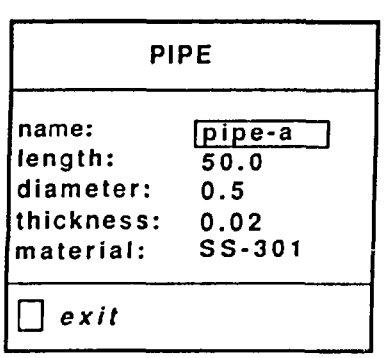

Figure 2. Pipe object editor.

User Interface

A requirement of the present simulation environment is that it have a highly interactive easy to use interface which will allow non-simulation experts to successfully construct a model. To achieve this the environment provides a full library of objects allowing the user to construct a model by simply drawing a process schematic and supplying plant-specific parameters. Interactive graphics are used for both model construction and conducting simulation runs. A model is constructed by placing and connecting icons on the screen with the aid of a mouse. A set of rules embedded in the model-manager class protects against illegal connections, such as two inputs or a fluid-flow input to a voltage output, by refusing to accept the second connection from the mouse. Parameters for individual objects are entered and/or changed through forms accessed by pointing at the objects icon with the mouse.

The use of a mouse to make connections between components eliminates or reduces a number of common sources of error related to syntax, spelling, and inconsistent or incomplete input-output specifications. These types of errors are frequent with textual based input and are often not caught until the compile stage, resulting in a time-consuming edit-compile cycle.

The use of graphical input and icons to construct simulation models is not unique. However, most graphical input packages developed until now produce code which must be pre-processed before it can be run. By contrast, the graphic interface described here is a means of directly manipulating objects which collectively make up the simulation "code." Hence the interface is not restricted to building models alone, but is the primary means of interacting with the simulation as it is running. Figure 3 is a representation of a typical screen as it appears during a simulation run. The screen is divided into three windows, the top containing a menu of options, the bottom left the plant schematic, and the bottom right strip charts for plotting output variables. An object's state or behavior can be changed in real-time during a simulation run by either pointing at its icon in the schematic window and changing a parameter (for example the set-point parameter of the controller object) or by sending it a message from the keyboard. The effects of the change or message are reflected immediately in the simulation output being plotted on the output window. This immediate feedback and elimination of the compile-link-run cycle is a substantial time-saver, particularly when engaged in exploratory modeling.

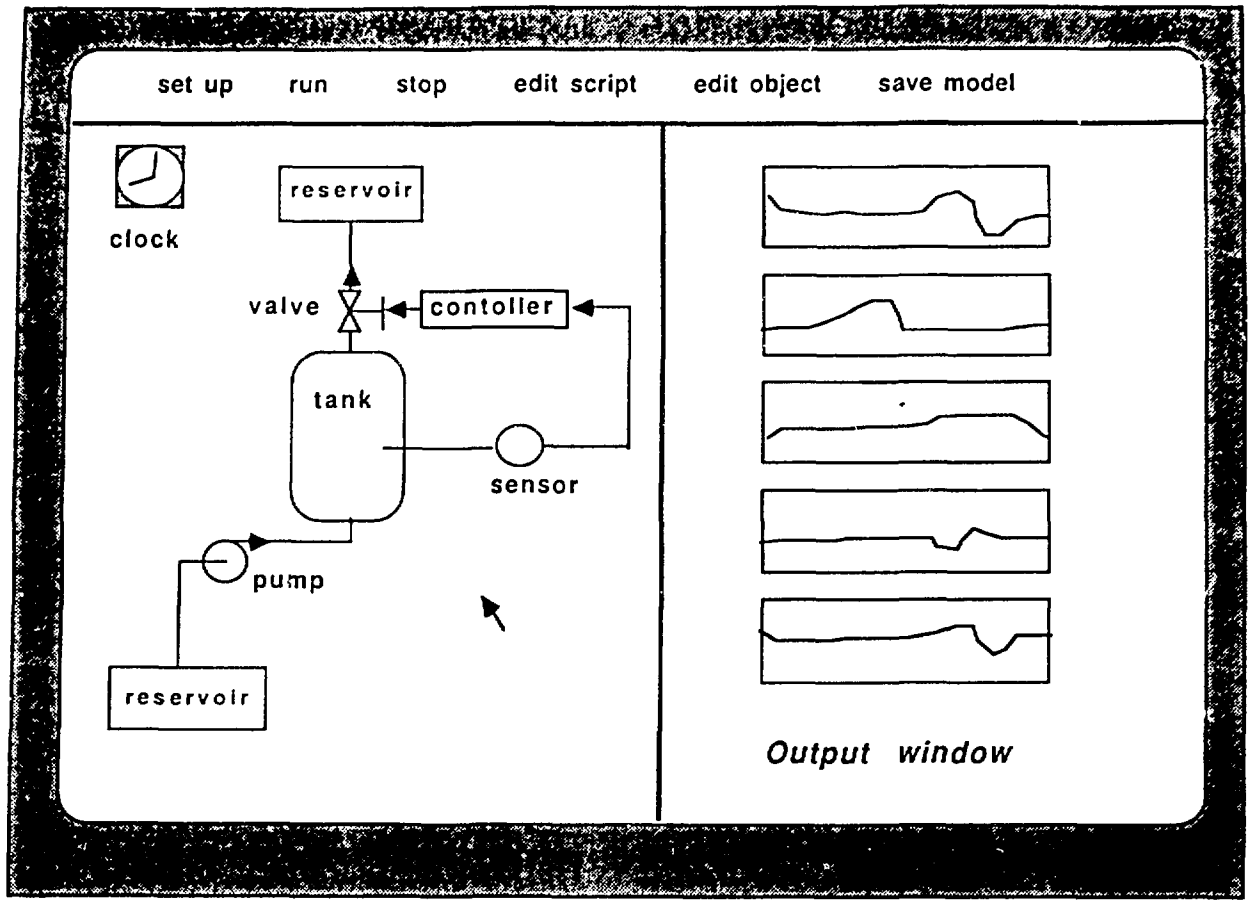

Figure 3. Simulation "run-mode" interface.

ADVANTAGES OF AN OBJECT-ORIENTED APPROACH

We believe that the object-oriented approach to programming offers a number of advantages for process simulation, the most significant being associated with the concepts of class, inheritance, and polymorphism as discussed below.

Classes are useful for building reusable generic descriptions of similar types of objects. This concept is particularly powerful when applied to modeling processes such as power and chemical plants since these systems are typically composed of a large collection of basic components which fall into a relatively small number of categories (tanks, pipes, pumps, valves, etc.). These categories can be conveniently described by class definitions with particular components being created by instantiation, overriding the default values of the class as necessary.

Inheritance can be exploited in two ways. The first is by creating a class hierarchy in which more generic attributes of similar objects are defined in top level classes which are then inherited by lower levels. For example, in our system there are a number of types of valves which all share the attribute "valve-position" and a restriction that it remain between 0 and 100% open. This attribute and the associated limit check are encoded in the class GENERIC-VALVE which has been used as the parent class for all valves (see figure 4). The use of a class hierarchy not only allows code to be reused, but results in improved maintainability since a single block of code shared by a large number of classes can replace individual routines for each class. Another view of inheritance useful if the language supports multiple parents is the "mixin" model. In this view, several classes can be mixed together to create a new composite class. We have used the multiple inheritance model to create a number of classes, including for example HYDRAULIC-SYSTEM. This class was created by mixing two previously defined classes, TANK and HYDRAULIC-LOOP. The new class computes average pressure via the methods incorporated into TANK and loop flow rate using the momentum conservation model of class HYDRAULIC-LOOP.

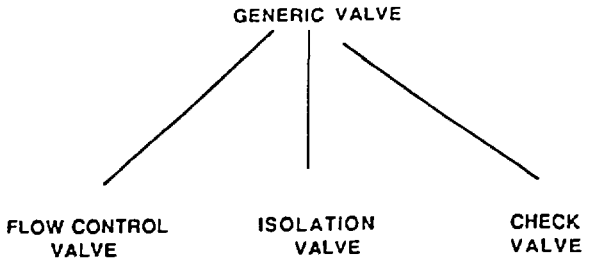

Figure 4. Valve class hierarchy.

Another feature of object-oriented programming that has proved useful for building a simulation package is polymorphism, the property that allows each class to define a unique response to the same message. The usefulness of this property can be illustrated with an example from the HYDRAULIC-LOOP class. An equation for flow rate around a closed loop can be derived by integrating the fluid momentum equation yielding:

$$\frac{dQ}{dt} = a \sum_i \sqrt{\Delta P_i} Q$$

where

Q is the loop averaged volumetric flow rate,
a is a parameter containing geometric coefficients, and
ΔP_i is the pressure drop across component i.

When this equation is implemented as a method, the derivative is obtained by sending a message to all components requesting their pressure drops ΔPi for the given flow rate Q. These components might include for example pipes, pumps, and valves, each with a different algorithm for computing pressure drop. In a traditional program we would need to determine *apriori* the type of each component in order to determine the correct subroutine for computing pressure drop. This would necessitate a test clause somewhere in the code which would need to be modified each time a new type of component is added to the simulation package, and might furthermore result in a proliferation of subroutine names such as compute-pressure-drop-valve, compute-pressure-drop-pipe, etc. With an object-oriented approach we send the same message to all objects in the loop "compute-pressure-drop", leaving it to the object itself to use the correct procedure.

FUTURE WORK

At the present time we are studying means of integrating the simulation environment with tools for linear analysis, frequency domain analysis, and symbolic equation manipulation to provide a capability for designing linear (state-space) and nonlinear control systems for nuclear reactors. As a first step towards integration with control design tools, we intend to improve our treatment of systems by introducing a consistent and formal representation scheme along the lines suggested by Åström and Kreutzer [8] to system-level classes. This should allow us to develop general algorithms to generate state-space representations, necessary for most modern control algorithms, from the object-oriented models. We also plan improvements to the user interface to allow zooming in and out of systems similar to the Hibliz system [9].

CONCLUSIONS

A simulation environment for power plants has been developed using object-oriented programming within a LISP environment. Advantages of an object-oriented approach to simulation include a high degree of modularity, the data abstraction afforded by classes, code reusability due to inheritance, and the ability to exploit the polymorphic nature of message passing to build very generic procedures. Finally, the interpretive nature of LISP eliminates the tedious compile-link-run cycle and allows dynamic, reconfigurable models to be built in a highly interactive manner.

REFERENCES

1. Cox, B.J. 1986. Object-oriented programming: an evolutionary approach. Addison-Wesley, Reading, MA.

2. B. Thompson. 1987. "Programming With Objects." AI Expert. September. 15-20.

3. Zeigler, B.P. 1987. "Hierarchical, modular discrete-event modelling in an object-oriented environment." Simulation (49)5. 219-230.

4. Stairmong, M.C. and Kreutzer, W.. 1988. "POSE: a Process-Oriented Simulation Environment embedded in SCHEME." Simulation (50)4. 143-153.

5. Ghaznavi-Collins, I. and Thelen, D. 1988. "An object oriented approach toward system architecture simulation." AI PAPERS.1988 (R.J. Uttamsingh ed.) *Simulation Series* (20) 4. SCS, San Diego, CA. 103-107.

6. Khlar, P. 1986. "Expressibility in ROSS, an Object-Oriented Simulation System." Artificial Intelligence in Simulation , (G.C. Vansteenkiste; E.J.H. Kerchoffs; B.P. Zeigler eds.). SCS, San Diego, CA. 147-156.

7. Stallman, R., Weinreb, D. and Moon, D. 1984. Lisp Machine Manual. Lisp Machine Incorporated, Los Angeles, CA.

8. Åström, K.J. and Kreutzer, W. 1986. "System Representations." Proceedings of the Third Symposium on Computer-Aided Control System Design, Arlington, VA, September 24-26, 13-18.

9. Elmqvist, H. and Mattsson, S.E. 1986. "A Simulator for Dynamical Systems Using Graphics and Equations for Modeling." Proceedings of the Third Symposium on Computer-Aided Control System Design, Arlington, VA, September 24-26, 134-139.