PASION:

OBJECT-ORIENTED SIMULATION ON THE PC

Stanislaw Raczynski
Panamerican University
Augusto Rodin 498
03910 Mexico City, Mexico

ABSTRACT

PASION is a process- and event-oriented simulation language designed for those who already know and use PASCAL. The language has a two level (process/event) structure and permits the use of all the Pascal structures. It also offers the main features of object-oriented programming. PASION provides necessary facilities to handle sequences of random events, queues and quasi-parallel processes, both discrete and continuous. A PASION source program consists of a sequence of process declarations and a main segment which initializes the simulation. At run time the program generates objects which represent model processes due to the process declarations. PASION provides tools which facilitate the building of complex models by the mechanism of inheritance.

INTRODUCTION

Looking at the existing simulation software and at the recent tendencies in programming, it can be seen that the existing simulation software becomes somewhat obsolete. PASION and its environment were created for the following reasons.

First, the new simulation software should be object-oriented not only because almost all new software is object-oriented (see Schmucker, 1986, for a review on object-oriented programming). The simulation software must have this orientation simply because the real world we intent to simulate is object-oriented. Any simulation language or package must permit to simulate dynamic objects in the general sense, that is, without regard to whether the object is discrete, continuous or combined.

Second, it is not a proper way to develop good simulation tools modifying or extending languages which are 20 or even 30 years old. In my didactic work I have been looking for a well structured and easy-to-teach simulation language. It seems that

the most complete one is Simula. However, it is somewhat difficult to teach Simula quickly, for its relation to Algol. The course on Simula simply does not fit into the simulation courses for non-programmers (e.g. for engineers who only know Basic and a little of Pascal or Fortran). The main requirements for the new language were as follows.

1. The language must be related to Pascal for the excellent didactic properties, clear and good program structure and the popularity of that language.

2. It must be object-oriented. This means that it must permit the user to describe the properties of objects (their behavior, interactions with other objects etc.), and to create and handle objects. It also should be possible to define a structure within classes of objects through the inheritance mechanism. This will permit the user to extend the complexity of his model using classes of objects created earlier, as implemented in Simula.

3. It must be designed for combined continuous/discrete models. This means that it should be little difference between continuous and discrete objects, and that these objects must be able to run simultaneously in the same model (in the same simulation program).

4. The language must offer all necessary mechanisms to control the model time, i.e. it must be equipped with a transparent "clock mechanism" which controls the model time, integration routines for continuous objects and event queue for the discrete part of the model.

5. The language must be equipped with an appropriate environment. The environment should offer a library of predefined processes (object classes), such as a general dynamic object, objects which generate plots of the model trajectories, generate history files, store the simulation results to be analyzed and used by other models or programs etc. The recent implementation of Pasion also includes a separate auxiliary modules such as a generator of queuing models, generator of continuous models (as Pasion processes) and a post-simulation analyzer for stochastic models.

6. It must run on microcomputers.

The recent implementation of PASION seems to satisfy, to some extend, the above requirements.

THE LANGUAGE

To describe a sequence of events we must specify each event and describe both dependence of each event on the model time, and the interactions between the events. A process-oriented language offers something more. Namely, it defines a structure within the set of events by introducing different processes. By the **process** we mean a generic segment of the program which declares a specific object type. This declaration describes the properties of objects which can execute events in relation to the model time. According to this declaration the corresponding objects can be created at run-time.

Let us consider, for example, a population of bacteria. A bacterium can divide, move, eat or die. Thus, each bacterium can be treated as a process which includes the following events: division, movement, eating and dying. This model, of course, consists of many objects of type "bacterium" which run concurrently. The objects are closely related to each other, because any "division" event creates a new object, and the bacteria can eat other bacteria and interact with the common environment. At run time, objects are generated according to process declarations. The objects can be activated, suspended or deleted. Once activated, an object executes its events and may interact with other objects. Objects may represent, for example, bacteria, cars, clients, shops with queues, continuous dynamic systems etc.

The following sketch shows the PASION program structure.

```
PROGRAM prname;
- - - - - {declarations of global variables, queues}
REF X,Y,Z:pname;
- - - - - {procedures and functions}
- - - - -

PROCESS pname,n;
ATR- - - -{attribute declarations}
          {local procedures and functions}

EVENT ename,tvname;
{local declarations, if any}
BEGIN
- - - - {event operations}
- - - -
- - - -
ENDEV;

EVENT ename2,tvname2;
{local declarations,if any}
BEGIN
- - - - {event operations}
- - - -
- - - -
ENDEV;
```

```
START
- - - - {the main program}
NEWPR X;
- - - -
NEWPR Y; NEWPR Z;
- - - -
$
```

where "prname" is the program name, "pname" is the name of a process, n is an integer, "ename" is the name of the "time-variable" of the corresponding event, used to schedule it. A program can contain one or more PROCESS declarations. The REF declaration introduces "reference variables" X, Y and Z of type "pname". In the main program these variables are used in the NEWPR instructions to create objects. The number of objects of type "pname" which can run concurrently is limited by the integer "n". The objects, after being created, should be given necessary parameters for their attributes. At least one event of each new object should be scheduled to activate the object. The scheduling operation has the form

```
tvname:=timevalue;
```

where "tvname" is the corresponding identifier which appears in the event header. This is not a substitution, but the scheduling operation which defines the moment at which the event will occur ("timevalue"). Each event can be scheduled many times, and the scheduling operation can appear in the main program or within any event. A simple and clear visibility rules permit to refer to any event of an object of any type from within any other object.

Let us consider a simple example of object-oriented simulation in PASION. To simulate the growth of a plant it is sufficient to describe the behavior of one "cell" of the plant. It can be a "branch element" which can generate one or more other branch elements which grow upwards, with random inclination. The branch element can also increase its thickness to "support" more branches. The program describes one branch element with two events: "generating new branch" and "to get fatter". The main program generates one initial core element which generates the branches (other objects of the same type), which, in turn, generate other branches etc. It is easy to show this process on the screen, as indicated in Fig.1. Observe that this simulation is not only the generation of the image of the plant. Each "branch element" of the plant is "alive", as an active object of the model and its behavior can be modified in order to experiment with the model.

CORRESPONDENCE BETWEEN MODELS AND PASION PROGRAMS

According to the commonly used simulation terminology (see Zeigler, 1976) a simulation model is composed by its **components** (e.g. clients in a shop). The state of each component is described by the corresponding set of **descriptive variables** and their activities are gives by the rules of interaction between the components. **Experimental frames** define the actually used set of descriptive variables and permit to determine the complexity of the model. The PASION language has all these basic modeling elements, as shown blow.

MODEL		PASION program
Components	<===>	Objects
Component specification	<===>	PROCESS declaration (object type)
Descriptive variables (including the state of the component)	<===>	PROCESS attributes
Component activities and the rules of interaction	<===>	Events
Experimental frames	<===>	Process hierarchy and inheritance

Inheritance enables programmers to create classes and therefore objects that are specializations of other objects. This enables the programmer to create complex models by reusing code created and tested before. Inheritance in PASION can be applied using prefixed process declarations. For example, if PA is the name of an existing process and we wish to create a new one, say PB, having all the properties of the process PA (this means all its attributes and events), it can be done using the name PA/PB instead of PB in the heading of the process declaration. While processing such declaration, the translator looks for the process PA (the parent process) and inserts all the attribute declarations and event descriptions from PA into the new process PB (derived process). Parent processes can reside in separate files, or be placed in the same source file. Thus, the user can prepare and store some useful source "capsules" and use them while creating new processes.

PASION ENVIRONMENT

There are few programming languages which can be effectively used without an appropriate environment. PASION is equipped with the Minimal PASION Programming Environment (MPPE) which consists of a library of predefined processes and other modules. It supports interactive simulation, graphics, statistical analyses of the results, continuous dynamic models and queuing models.

PASION PREDEFINED PROCESSES

The core of MPPE consists of the library of PASION predefined processes. These are generic program segments which generate processes. Predefined processes are written in PASION extended by a simple "meta-language" which permits a process to have formal parameters. The user invokes a predefined process by its name and specifies the actual parameters, which are passed to the corresponding process declaration in the user program by name, before the program is translated to PASCAL. The user can prepare his own predefined application-oriented processes and add them to the library.

The library provided with the PASION-to-PASCAL translator PAT4 contains the following predefined processes.

INTERP for graphical output, interactive simulation,

INTERB for graphical output (animated bar-graphs etc.), interactive simulation,

INTERN numerical output, interactive simulation,

SHOWP graphical output at given time instant,

STOR (and program VARAN) graphical output after the simulation is terminated, average trajectories, variance analysis, confidence intervals,

MONIT displays the existing objects in graphical form,

DYNAM simulates a continuous dynamic system,

LSTAT for queue statistics.

QUEUING MODEL GENERATOR

The Queuing Model Generator (QMG) is a module of the PASION environment. It contains a block-diagram editor which permits the user to define the structure of the model, and a program generator which generates the corresponding PASION code. This code is automatically translated to PASCAL. QMG is transparent, i.e. the languages (PASION and PASCAL) are not visible for "non-programmer" users. However, if the user knows PASION or PASCAL, then he can work with the resulting code, creating much complex models.

CONTINUOUS MODEL GENERATOR

This program was designed as a module of the simulation language PASION in order to facilitate simulation of dynamic continuous systems. The Continuous Model Generator (CMG) is a program generator which generates automatically source PASION and/or PASCAL code, according to the model specifications given by the user, mainly in graphical form. Actually two versions of CMG are available: Pasion version (CMG/PN) and Pascal version (CMG/PS). The only difference between these versions is that CMG/PN can generate PASION source code and CMG/PS can not. Thus, the CMG/PS version can be used as an independent simulation tool, without using any elements of the PASION system. The only additional software needed is a Pascal compiler to run the resulting program.

CMG is a program generator with built in graph diagram editor. The input to this program is the graphical description of a model created by the user on the screen in the interactive mode. The model may be linear or non-linear and may contain time-delay links. The model also can include "sample-and-hold" links, which permits to simulate digital control ("sampled-data") systems.

After completing the corresponding graph and defining the necessary model parameters, CMG generates the corresponding source code, so that no programming is needed to create the simulation program. The PASION output is created in the form of a PASION process declaration which can be inserted into any PASION (continuous, discrete or combined) model. The PASCAL output is generated as a complete PASCAL program which can be run using a PASCAL compiler.

The input to CMG is formulated in terms of graph diagram which describes the dynamics of the modeled system. By the graph diagram we mean a network composed of nodes and directed links. Nodes represent signals and links represent transfer functions.

CMG permits the following types of the links:

1. Static linear
2. Static non-linear

3. Dynamic linear
4. Time delay
5. Sample-and-hold
6. Superlink (a complex dynamic system)

The last link type (Superlink) permits to include whole dynamic model (specified earlier and stored in a file) to the model actually being created. This feature is useful while developing complex models, composed of submodels created and tested separately.

There are two output modes for the CMG module. The PASCAL mode and the PASION mode. In PASCAL mode the program generates a complete PASCAL source simulation program which can be compiled and ran using a PASCAL compiler. In the PASION mode, CMG produces the PASION source code being a PASION process declaration. This declaration can be included into any PASION program, and can generate one or more dynamic continuous objects. These objects can run concurrently with any other PASION objects of any type.

COMBINED MODELS AND STATE EVENTS

There are three ways to define continuous processes: (1) To describe a continuous process as a source PASION process, (2) to use the predefined process DYNAM and (3) to use the embedded global continuous process. The case (1) needs more programming and gives the programmer full control over the process. The corresponding process can be coded by the user or generated by the Continuous Model Generator CMG. Using the predefined process DYNAM the user only invokes the process and specifies the right-hand sides of the corresponding differential equations as its actual parameters. Case (3) is the shortest way to introduce a continuous process and only needs the expressions for the right-hand sides of the system equations to be coded. The embedded continuous process is global and visible from inside all existing objects. Any number of continuous objects of kind (1) and (2) can run simultaneously with other (discrete) object in the same model.

PASION permits to define "state-events" which occur when the continuous part of the model reaches a particular state.

IMPLEMENTATION

PASION-to-PASCAL translator runs on the IBM PC and compatibles. The code produced by the translator can be compiled by any PASCAL compiler. The resulting program expands dynamically while new objects appear, so that the number of objects which can run simultaneously depends on the amount of the operational memory available at the run time and on the size of data (attributes) of the objects. A new version of the PASION translator is being

developed which swaps the objects between the memory and a hard disc. This version uses the operational memory only to maintain arrays of pointers to the generated objects, so that the total amount of objects can theoretically be as great as 350.000 on a 640 Kbytes machine equipped with appropriate hard disk. However, this mode of simulation will be considerably slower than the "operational memory" version.

PASION has been used in teaching simulation methods. It is important to have an easy to learn simulation tool which may be used to illustrate the concepts of process declarations, objects, events, inheritance, preprocessing and animation, when the students have some knowledge on structural programming in PASCAL and does not have any experience in simulation.

LITERATURE

Schmucker, K.J.: "Object-oriented Languages for the Macintosh", BYTE 11 no. 8 (Aug.), 1986.

Raczynski, S.: "PASION - Pascal-related simulation Language for small systems", SIMULATION 46(6), June 1986.

Raczynski, S.: "Process hierarchy and inheritance in PASION", SIMULATION 50(6), June 1988.

Raczynski, S.: "PASION: The language and its environment", Proceedings of the SCS Multiconference on Modeling and Simulation on Microcomputers, San Diego, February 1988.

Raczynski, S.: "On a simulation experiment with a parallel algorithm for optimal control", Transactions of the Society for Computer Simulation, 5(1),1988.

Raczynski, S.: "Simulating our immune system", Proceedings of the SCS Multiconference, Modeling and Simulation on Microcomputers, San Diego, January 1989.

Zeigler, B.P. "Theory of Modeling and Simulation", John Wiley & Sons, New York 1976.

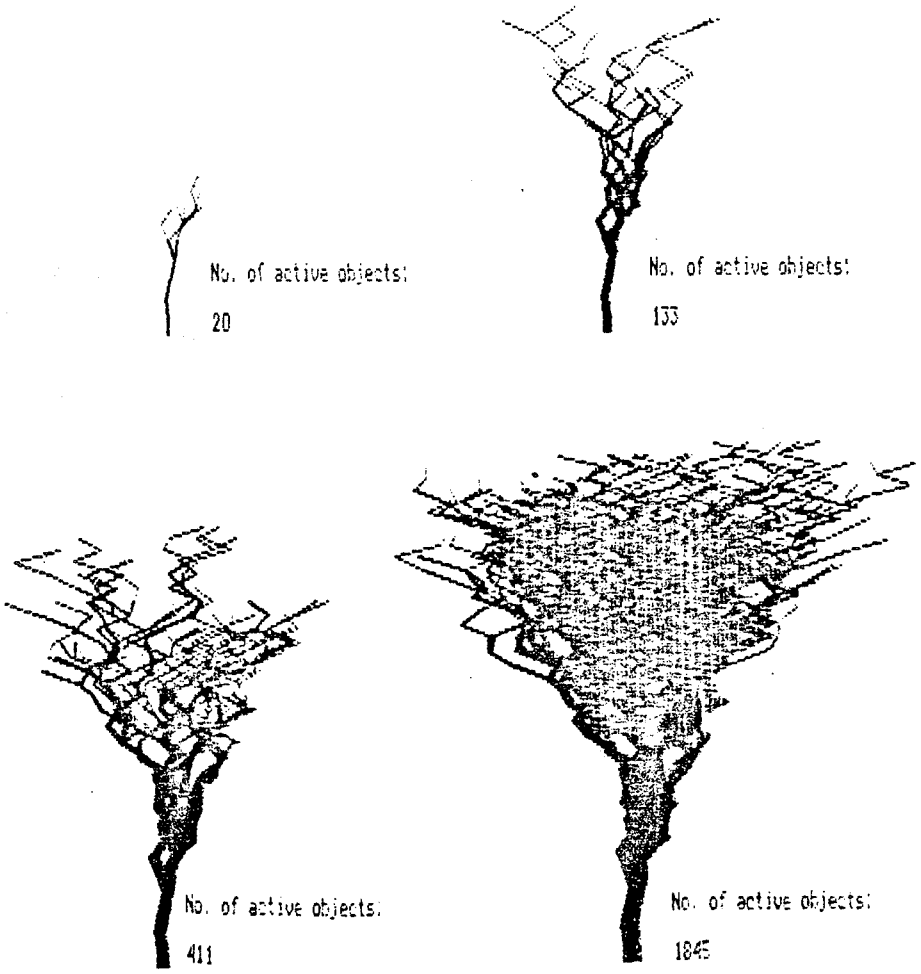
No. of active objects:
20
No. of active objects:
133
No. of active objects:
411
No. of active objects:
1845

Fig. 1