Written in 1996.
This document was converted from an old WordPerfect format. The resulting format may be distorted and the quality of the figures is rather poor. As it is a draft, it may contain some missprints snd other small errors.

A SMALL TOOL FOR COMPLEX SYSTEM SIMULATION [1]

Stanislaw Raczynski
Escuela de Ingeniería, Universidad Panamericana
Augusto Rodin 498, 03910 México D.F., Mexico
fax (525) 563 3747
E-mail: stanracz@prodigy.net.mx

ABSTRACT

A new tool for complex dynamic system simulation is presented. The system complexity is related to the diversity in the submodel types rather than to the number of components and the model size. The presented software supports model coupling as defined in the DEVS (Discrete Event Specification) formalism. Though the DEVS formalism is used, the model components can be of any type, supporting discrete and continuous models running in the same simulation program.

Keywords: Simulation Methodology, Modelling, DEVS Formalism,
Object-oriented Simulation

[1] This paper is a part of the research carried out in the Center of the McLeod Institute for Simulation Sciences at the Universidad Panamericana in Mexico City.

INTRODUCTION

Object-oriented simulation offers excellent tools for treating models of complex dynamic systems. First of all, we must decide when the system is really **complex.** Obviously, a system described by a huge set of equations is not necessarily complex. We can solve on a PC sets of thousands of simultaneous differential equations, but this does not mean that the model we solve is complex. On the other hand, a system may result to be complex even if the equations are apparently simple, but the system components have very distinct dynamic behaviour or if they are of different kind. Those are cases treated as complex in this paper. We say that a dynamic system is complex, if it has **multiple components which reveal different dynamic properties**. This may occur, for example, when all system components are continuous with concentrated parameters, but the model includes very fast and very slow parts. Other example is a system where discrete parts interact with continuous submodels of different speed and different kind, for example an electronic circuit that contains integrated circuits as well as electro-mechanical parts such as relays and motors. In other words, the model comlexity has little to do with the model size. The tool described here is partly the result of the author conviction that **small is beautiful** and that to simulate complexity we do not need huge software packages.

Using object-oriented approach we can simulate complex systems creating objects that simulate system submodels and run concurrently. The idea is to launch a set of objects and to coordinate them by other object that only connects the submodels and controls a general (global) interaction rules. Thus, a submodels of very different kind can run and interact in the same simulation program, some of them with integration step thousands of times smaller than others and some of them being discrete or combined.

THE DEVS FORMALISM AND MODEL COUPLING

The DEVS (Discrete Event Specification) formalism defines a model **M** as follows.

$$M = < X , S , Y , \delta_{int} , \delta_{ext} , \lambda , \tau >$$

where **X** is the input space, **S** is the system state space, **Y** is the output space, $\delta_{int} : S \rightarrow S$ 2 is the internal state transition function, $\delta_{ext} : Q \; x \; S \rightarrow S$ 3 is the external transition function, **Q** is the "total state" (the set of all previous and actual model states), $\lambda : Q \rightarrow Y$ 4 is the output function and τ 5 is the time-advance function. The DEVS formalism in widely used in many publications on modeling. In fact, it is not clear why it refers to "Discrete Event" models, while the formalism itself does not imply any discrete model character.

One of the most important applications of the DEVS formalism is a clear and accurate specification on **coupled models**. The **coupling specifications** help us to build and document complex models, created using the concept of **modular composition** (see Zeigler 1987, Srivastava and Ragade 1994, Chow and Zeigler 1994). The task of coupling includes the **external input coupling** that tells how the inputs of the composite model are identified with the inputs of the components. **External output coupling** defines the same for the coupled model and component outputs. **Internal coupling** specifies how the components inside the model are connected with each other. A coupled model **CM** is defined as follows.

$$CM = < D, \{ M_i \}, \{ I_i \}, \{ Z_{ij} \}, SELECT >$$

where **D** are the component names, $\mathbf{M_i}$ are the component DEVS specifications, $\mathbf{I_i}$ is the set of influencers of the component **i**, $\mathbf{Z_{ij}}$ is the i-to-j output transition function. The **SELECT** function handles the simultaneously scheduled internal events. In some versions of the DEVS specification (extended DEVS), the SELECT function is replaced by the ORDER function that serializes the execution of simultaneously received external events. The extended DEVS specification is an important tool while dealing with big coupled models that run on multiprocessor parallel machines. We will not discuss these issues here, because what we are talking about is a small PC implementation.

THE COMPLEX DYNAMIC SYSTEM SIMULATOR

An example of implementation is described, using the mechanisms and the env ironment of the PASION simulation system. A new module of PASION named Complex Dynamic System Simulator (CDSS) has been developed. It permits to construct models of complex systems practically without programming. The only arithmetic expressions that must be given by the user are the right hand sides of the differential equations for the submodels that require such equations, and the output operations. Other parts of the model may be constructed using processes generated by other PASION modules, such as the Continuous Model Generator that provides source code for models described by Signal Flow Diagrams, or the Queuing Model Generator. Other option of the submodel menu is just “Any process” that can be any PASION process, coded directly in the PASION language. The interactions between submodels are defined connecting the corresponding submodel outputs and inputs. The submodels appear on the screen as blocks with their inputs and outputs and the user defines the connections simply using mouse to put the corresponding arrows. Then, the PASION code is generated, translated to Pascal, compiled and run. The program provides a great variety of plots and numerical and graphical representations of the results.

Recall that PASION is an object-oriented, Pascal-related simulation language that handles models specified in terms of processes and events (Raczynski 1986, 1988). A Process is an object type declaration that defines the object properties, such as attributes and events. At run time, objects are created according to the declared processes and activated. Then, the objects run executing their events and interacting with each other. The event queue and the clock mechanism are hidden from the user and quite fast and effective. The PASION models can be expressed in terms of the DEVS formalism. Though PASION process declaration does not define formally model inputs and outputs, the relation between PASION processes and DEVS formalism is very clear. Both inputs and outputs and some model parameters are PASION Attributes declared using the ATR word instead of Pascal VAR. The attributes can be Pascal simple or structured variables of any type. This gives the model quite great versatility. The modeller should know which of the model attributes are inputs, which are outputs and which are model parameters. All other model variables are hidden form the outside world. The internal and external transition functions are coded as the process events. Using the object-oriented terminology, process attributes are model data and the events are

model methods. The difference between an event and a procedure is that the events can be scheduled, and that their execution occurs only through messages issued by the scheduling mechanism, hidden from the user. Special scheduling instructions are used to put the events on the event queue. As in any object-oriented application, the model objects are generated according to the process declarations at run time. The resulting objects can be activated, interact with each other, can generate other objects or wipe out other objects or themselves. There is no formal difference between PASION discrete-event and continuous processes. Simply a continuous object, described, for example, by a set of differential equations, has at least one event that integrates the system equations over a given time-interval and schedules itself to be executed repeatedly. This means that a continuous object occupies a space in the event queue. The loss of space and speed is not significant, because it is only one event message per "continuous" object at a time in the event queue. The cost (computer time) of simulating a continuous dynamic system depends on the time needed by the calculations of the right-hand sides of the system equations per model time unit. This time is always supposed to be greater than the time consumed by any event handling operations (otherwise the problem is trivial). Consequently, the simulation speed depends mainly on a particular numerical method that is applied to calculate the next system state and not on the mechanism of the event queue.

The PASION scheduling mechanism and its environment is a good tool to simulate complex systems. The idea is to connect different models through their input and output attributes and let the whole model run under the control of the global scheduling mechanism (called Clock Mechanism in PASION terminology). This results in a new model that has two levels of interactions: the internal interactions between objects inside each particular submodel (a model component) and the interactions between submodels. It is supposed that while the interactions of the first kind are strong and frequent, the interactions between submodels are not so frequent and somewhat "loose". The assessment of the "strength" of interactions between model components depends on the user. Interactions considered as "loose" at a certain coupling level, may be qualified as strong or frequent on another level. This article does not give any rules for such interaction qualification. The internal submodel interactions are executed due to the normal PASION mechanisms, according to the coded scheduling messages and to other instructions that can modify or reference to the attributes of an object from inside other objects. The interactions between submodels are controlled by a special connector object that is always active and sends the output submodel signals to corresponding inputs. The general model scheme is shown on figure 1. The predefined process type Con_Obs specifies the connector activities. It also "observes" the model trajectory and stores it for further processing. This obviously needs some hard disk space. The default time resolution for the observer process is 1/1000 of the model final time. This means that to store the results wee need the space (number of real variables) being approximately equal to the number of observed (not all) variables multiplied by 1000. This resolution for the files generated for further use by the VARAN5 utility (see Example section) is reduced to 100. Only one object of type Con_obs is generated for each model generated by CDSS. However, if the model is coupled into a submodel, as described in the next section, then the model just created is coupled together with its connector object.

The submodel types supported by CDSS are:
Differential Equations (ODE Model) - model described by a set of ordinary differential equations,
Signal Flow Diagram - model described by a Signal Flow Diagram, created by the CMG module of the PASION system,
Transfer Function - a linear dynamical system of any order, described by the transfer function,
Integrator - an integrator,
Sample-and-hold - the typical sampler used in models of digital control systems,
Algebraic Model/Function Generator - a static model, including only the input-output algebraic function. If number of inputs is zero, then the model becomes a signal generator.
PID Controller - used in control systems simulation,
Any Process/QMG Model - it may be any process coded by the user, a set of processes generated by the Queuing Model Generator QMG or a coupled CDSS model.
Bond Graphs - models of physical systems given in the form of bond graphs.

The time delays can be modeled as output -input links with delay or as separate CDSS blocks.

The above options offer a quite complete set of submodels. Taking into account the "Any process" option it can be seen that one can simulate practically everything. Note that the options include also very simple models like integrator or the PID controller. Simulating, for example, a control system, the user can compose it using the ready-to-use CDSS submodels, connect them and run. However, this is not a proper way to create complex models. In the case of a control system, more effective way is to first generate the code using the Signal Flow Diagram (CMG module) and then include it as a submodel to the new CDSS block. Such blocks as the PID or the algebraic model should be used only as auxiliary blocks, if necessary.

HIERARCHICAL MODEL COUPLING

The model structure composed of objects clustered into submodels and the connector object permits hierarchical model coupling. To create a coupled model, the user must choose a model components to be coupled (a set of submodels) and define the correspondence between the inputs and outputs of the coupled model and those of the submodels. Naturally, this process should result in a new model having reduced number of input and output signals, hiding the other signals form the outside world. Also all the submodel parameters and initial conditions are fixed and hidden. This makes the new coupled model ready to run without asking the user to define anything inside it. The coupled model can be stored in a file and used as a submodel in any other model created using the CDSS program. So, it is not important at which level of this process of coupling we are while running a particular CDSS session. If the newly created model is composed by simple submodels (integrator, ODE model, PID etc.), then we are on the lowest (zero) level of coupling. However, the user can add to the model a coupled model that is a result of multiple previous CDSS sessions, making the new model highly coupled and complex. The CDSS program is a code generator that produces PASION source code ready to be processed by the PAT4 PASION translator. Each submodel has its own connector object. In this way, the connector objects

are also subject to the hierarchical coupling, and will run concurrently, possible with different activation steps.

The whole model is processed by the PAT4 Pasion to Pascal translator and complied by a Pascal compiler. This means that all process declarations and their events are translated regardless to their place in the hierarchical coupling structure. In other words, the PASION translator does not recognize any model structure other than process/event and the resulting event queue is unique. It is not necessary for PAT4 to recognize other model structures, because all interactions between processes are well defined by the CDSS code generator. Recall that PASION language does not permit nested processes, so that coupling can not be realized by nesting processes. In fact, this is not necessary. The only difficulty that may appear are possible name conflicts. This is avoided by renaming processes. A submodel may be described by a single process (like, for example, a process generated by the PASION Continuous Model Generator CMG) or it may be a set of processes. For example, a typical queuing model generated by the QMG module, contains a generator of entities, the entity process (that describes the "life" of the entity passing through the model blocks) and the "top process" that initializes the model. The top process and "subordinated" processes scheme may also occur while including a model explicitly coded by the us er.

While coupling submodels, the user is only asked to give the name of the top process of the new coupled model. All top processes of the submodels are renamed as xxxN, where xxx is the name of the top process and N=1,2,3... . The names of the submodel processes, that are not top processes are renamed randomly using a 5-character code with random characters or digits. This does not eliminate the possibility of name conflict, but makes its probability equal to M/50,602,347 where M is the total number of process declarations.

It should be mentioned what CDSS and the whole PASION system does not support. The main feature not supported and widely treated in the DEVS literature is the collision handling. This problem appears when simultaneous events occur that may change the object state, and when the way how the state changes depends on the previous state. This makes the next state ambiguous. Many simulationists attempt to solve this problem introducing the SELECT or ORDER functions to the DEVS formalism, giving the user control over the collisions. My point is, however, that in the real word the simultaneous events does not exist. **The appearance of simultaneous events in simulation models is the result of naive attempts to change the real world to fit our simulation tools.** If a model of a real system leads to such colliding events we should revise the original problem. Or the model we created is wrong, or the original problem has no unique solution. If the behavior of the real system is ambiguous, the simulation model should also be ambiguous and giving the simulationist control over the ambiguity is an error. Instead, the problem should be randomized and, as the result we should obtain a set of trajectories instead of one deterministic solution. See Traub and Wozniakowski (1994) for interesting discussion on the role of randomization in computability problems. It should be emphasized that the above considerations only reflect the author's personal view on the problem of collision handling. In fact, the simultaneity of events, both in real world and in modeling is rather a philosophical problem. This article is merely a description of a small software toll and do not pretend to solve it. In many models the time discretization is introduced to increase the

efficiency (see *cellular models*, Chow, 1966). However, in my opinion such simplification makes the model validity doubtful. The authors of models of such kind never prove the model validity. Recall that to prove that a model is valid one must prove that the corresponding *validity graph* (see Zeigler, 1976) commutes. For more discussion on collision handling see Praehofer and Zeigler 1996, An interesting approach to DEVS.based models verification (not validation) is presented by Hong and Kim, 1996.

EXAMPLE

This is an example of a "small complex model". The model is small because it has few process declarations. Its complexity consists in putting submodels of very different kinds in the same program. Consider a real system consisting of a small shop where the clients enter, are attended by a vendor and then by a cashier. The input flow is of Poisson type with variable intensity. Inside the shop two queues may appear (vendor and cashier). It is winter and the shop has a simple temperature control system that includes a heater, a thermometer and a two-point (on-off) controller. Each client entering of leaving the shop opens the door that produces a strong negative disturbance affecting the temperature. On the other hand, a client who remains inside the shop is a source of heat that adds to the energy produced by the heater. The problem is to simulate the queues and the changes of temperature. Figure 2 shows the block diagram of the model, as it appears on the CDSS edit screen. Setp is a signal generator (temperature set point), Cont is the controller eps being the temperature error, Heater is an electric heater, Shop is the inside of the shop to be heated, Medi is a thermometer and Shop1 is a queuing model that simulates the client movements and queues. Auxi is an auxiliary algebraic model that simply calculates a weighted sum of the outputs from the Shop1 model and applies it to the actual shop temperature. X and Y are the lengths of the two queues, Z and U represent the state of the two servers (Busy or Free) and DOOROP tells if the door is closed or open. The numbers in brackets indicate the object instance numbers. Note that here the control system was created using a simple CDSS blocks. The other way to generate it may be to use a process generated by the CMG PASION module or to include a corresponding submodel of "Differential Equations" type.

CDSS supports three types of simulation experiments: a Single Simulation Run, Changing Parameter and Preparing files for the VARAN5 Program. The first one is a simple simulation run, where the results are given in the form of various time- or XY-plots of the observed variables. The Changing Parameter option is an experiment where a selected system parameter changes within given interval. The corresponding model trajectories are stored and then shown graphically. The third simulation mode may be used when the model includes at least one stochastic signal. A given number of (different) system trajectories are stored to be analyzed by the VARAN5 program of the PASION Simulation System. That program shows the reachable sets (max and min values reached), average trajectories and confidence intervals for each variable as functions of time, with a given confidence level.

Figure 3 shows the plot of the average length of the queue X (vendor) with the corresponding confidence intervals. These results were obtained by running the executable code of the simulation program on a PC. The whole process is as follows: first the user defines his CDSS model using submodels prepared earlier, or in the same CDSS session, as described before. Then, CDSS generates the complete PASION program to carry out a

selected simulation experiment. In this case, the experiment option was "Preparing files for the VARAN program". Then, the Pasion-to-Pascal translator PAT4 generates the Pascal code, that is compiled by a Pascal compiler. Total time of translating and compiling Pascal is not grater than about 15 seconds on a 486 machine for medium model, in the presented case shorter. The resulting program simulates the model and produces files for the VARAN5 utility and a number of graphics, not shown here. The plots of fig.3 and 4 were produced by the PASION VARAN5 utility. More graphics (zoom, XY plots) can be obtained using the CDSH program (auxiliary CDSS utility).There is no room here to show the PASION and PASCAL codes generated by CDSS. These code files are available from the author on request. For more information about PASION consult PASION manuals or visit the page http://www.mixcoac.upmx.mx/pasion/sumpas.htm

On figure 4 we can see the similar plot for the temperature inside the shop. It can be seen how the temperature drops when more clients enter the shop. In this particular case the discrete (queuing) submodel can be run separately and then its trajectory used as an input for the continuous part. Observe, however, that with a slight modification, where the bahavior of a client depends on the temperature, such decomposition would not be possible.

The model can be simplified by coupling the control part. Figure 5 shows the coupled model. Coup is the coupled control system. I311 is the input that corresponds to the input z in the original model and O211 is the output temp (temperature). The other variables and parameters of the coupled model are hidden. The simulation results obtained from this model are exactly the same as from the original one.

CONCLUSIONS

Object-oriented simulation tools offer an excellent environment for implementations of the concepts developed using the DEVS formalism, not only for discrete models. Model coupling is the main feature in creating models of complex systems, even on microcomputers. Further research should be concentrated on this issue. More powerful versions of the software described above are being developed for UNIX machines.

REFERENCES

Hong Gyung P. and Kim Tag g., A framework for Verifying Discrete Event Models Wirhin a DEVS-Based System Development Methodology, Transaction of the Society for Computer Simulation, Volume 13 no. 1, March 1996.

Chow A.C., Zeigler, B, 1994, parallel DEVS: A Parallel, Hierarchical, Modular Modeling Formalism, Proceedings of the 1994 Winter Simulation Conference.

Raczynski S., 1986, PASION - Pascal-related Simulation Language for Small Systems, SIMULATION 46(6).

Raczynski S., 1988, Process Hierarchy and Inheritance in PASION, SIMULATION 50(6).

Srivastava A., Rammohan K.R., Object-oriented Simulation of a SIMD Computer Using

OMT and DEVS Methodology, Proceedings of the Object Oriented Simulation Conference (OOS'94).

Traub J.F., Wozniakowski H., 1994, Breaking Intractability, Scientific American, 270(1), pp. 90B-93.

Zeigler B.P., Theory of Modeling and Simulation, Wiley-Interscience, New York, 1976.

Zeigler B.P., Hierarchical, modular discrete-event modeling in an object-oriented environment, SIMULATION 49(5), pp.219-230.

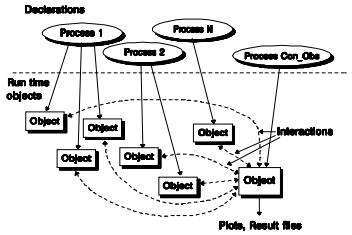

Figure 1. General structure of a CDSS model.

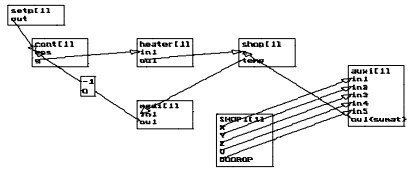

Figure 2. CDSS screen with the EXAMPLE model.

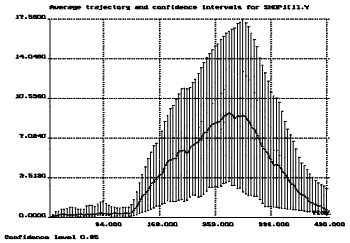

Figure 3. Average length and confidence intrvals fo the Cashier queue - EXAMPL

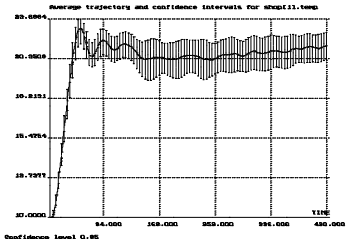

Figure 4. Average temperature with the corresponding confidence intervals - EXAMPLE.

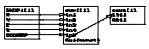

Figure 5. EXAMPLE - coupled version.