DEOS – A Discrete Event Object-oriented Simulation Environment

A. Anglani, P. Caricato, A. Grieco, F. Nucci, M. Pacella
Dipartimento di Ingegneria dell'Innovazione Università degli Studi di Lecce
Via per Arnesano, Lecce, 73100 Italy, e-mail: antonio.grieco@unile.it

Abstract

After many years of development and after being used in many different fields, the traditional process-oriented simulation paradigm has shown all of its strengths and weaknesses. Although a wide success has been reached, which has led benefits to simulation theory and application in general, process-oriented simulation has shown several limits as well.

Many efforts have been made in order to apply the successful concepts of object-oriented analysis and design [Boo94] to simulation theory, especially in commercial products such as ModSim® [Cac92] and Simple++® [Aes]. Free *Open Source* [Pav99][Cub97] tools are also available, but only in very specific fields of application such as network design [Omn]. None of these products, anyway, can be regarded as a complete and general purpose framework that allows analysts to model real-world cases applying all of OOA concepts. Furthermore, none of them supports designers with ease-of-use and user-friendliness.

DEOS (Discrete Event Object-oriented Simulator) is a first step in the direction of realizing such a framework. It is based on a specifically designed class library that provides the underlying architecture needed to develop the entire simulation environment. A design tool is also provided, exploiting this architecture in order to provide the designer with a powerful and user-friendly modeling framework.

1. Introduction

Process-oriented modeling can be very effective and useful in many cases, but it is not flexible at all. Once a business or manufacturing activity has been modeled [Hlu99][Mac95][Peg95], it is often very expensive to modify this model in order to simulate a slightly different model (e.g. either different layouts or work policies). In some cases it can be as expensive as re-modeling the entire activity.

DEOS (Discrete Event Object-oriented Simulation) is a complete discrete event object-oriented simulation environment designed considering two main aspects:

- ease-of-use, in order to provide designers with a visual, user-friendly simulation tool
- extensibility, to enable experienced developers to take advantage of DEOS class library when building specific objects needed in certain disciplines

Hence, DEOS is aimed at two different kinds of user, providing each of them with different tools and facilities. DEOS is aimed at designers, providing them with a user-friendly object-oriented modeling and simulation tool. DEOS is also aimed at developers, providing them with a complete and powerful class library needed to enable them to build new classes to be painlessly used within the simulation tool in order to model real-world objects involved in the activity under analysis.

DEOS provides a complete framework to support designers to easily build their own simulation models, visually combining and connecting different pre-built customizable blocks.

Furthermore, DEOS is an *Open Source* object-oriented software; hence it supports developers in building their own "new" blocks that make use of the class library provided in order to co-operate with other objects during the simulation without having to manage simulation issues such as synchronization, events, message passing. This is accomplished by the DEOS class library and can be easily exploited using a standard and well-known language such as C++.

2. The Design Tool

Unlike traditional process-oriented simulation tools, the object-oriented paradigm allows designers to model a system through the objects that it contains rather than the processes taking place within the system. With such an object-oriented approach, every possible process is automatically modeled when each object has been modeled.

The DEOS simulation software takes advantage of the object oriented approach and of the newest advances in operating systems and user-interface technologies.

Each object involved in the model of a system is represented by a "box" and each box is associated with an "Object Inspector" (see Fig. 1), i.e. a window that lists all the properties (referred as "attributes" using the traditional OOP terminology) related to the object in use.

In such a context, objects deal with objects: e.g. a queue is an object that contains elements that are modeled as objects as well.

Furthermore, objects can be easily and visually linked each other using "arrows" (see Fig. 2). Each

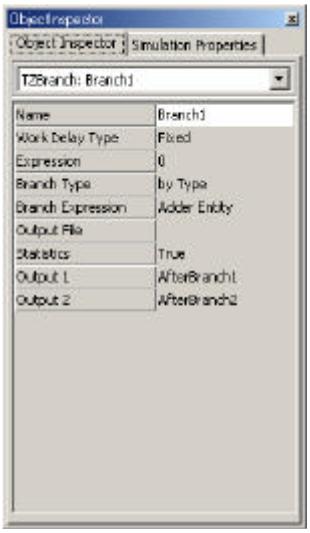

Fig. 1 – Object inspector

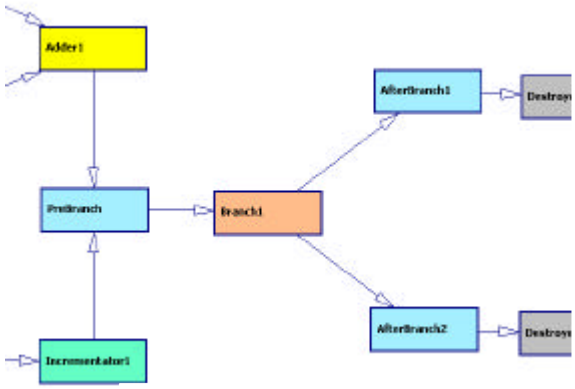

Fig. 2 – Connecting objects

arrow represents the object which is passed between two other objects that work on it.

The power of this approach is considerably evident with the growth of complexity of the objects involved. Indeed, using simple objects such as queues, simple machines or branches, DEOS resembles tools such as Arena®, that do provide visual simulation but not object-oriented features. Nevertheless, when dealing with real world models involving very complex objects - both in terms of objects that model machines and objects under processing - the superiority of the object-oriented approach is evident. Each box, indeed, represents an object (even very complex ones) and different objects of the same type (i.e. different instances of a common class) can be modeled and specialized, if necessary, simply changing their attributes.

Besides, objects are connected each other so that objects can flow through them. A change in the layout of a manufacturing system modeled with DEOS, for instance, can be easily modeled varying properties, editing connections and inserting other objects. Such an issue could require a relevant coding effort using traditional process-oriented simulation tools.

Finally, information flow through objects, objects generate events and other objects respond to these events. Using these features, a very easy implementation of message passing issues is achieved.

A basic library of simulation objects has been implemented in order to show some of the potentialities of the DEOS simulation environment. Here follows a brief description of these classes:

- *TCreator* – simulates the arrival of entities within the analyzed system according to a probability distribution
- *TDestroyer* – simulates the exit of entities from the modeled system
- *TQueue* – models a FIFO queue containing *TEntity* objects; the capacity of the queue can be dynamically specified by the designer
- *TIncrementer* – models a single-input single-output machine that receives a *TNumber* object, increases its value of a user-defined delta and makes it available after a certain user-defined delay; this class may be used to implement generic single-input single-output machines that process entities for a given time
- *TBranch* – models a single-input double-output machine encapsulating some user-defined logic to redirect the objects in input towards only one of the two outputs
- *TNumber* – unlike previous classes (that are inherited from *TResource*), this class descends from *TEntity* and is characterized by a *Value* numeric attribute. This class shows how any DEOS class can be specialized to manage entities of any kind and to have a consequent behavior

The design tool also provides the possibility to gather specialized parameters for each object in order to allow a statistical analysis of the simulations performed. Each simulation object, indeed, can be configured so that it keeps memory, while the simulations are executed, of certain parameters that are relevant for the kind of object considered. The information logged may be displayed and exported as an ASCII text file for further processing with third-party spreadsheet applications such as MS Excel® or Star Office Calc®. Nevertheless, a visualization software has also been implemented that parses the text files created and automatically builds charts in order to show the statistics related to each simulation object both grouping them by parameter or considering individually each parameter in any replication executed (see Fig. 3).

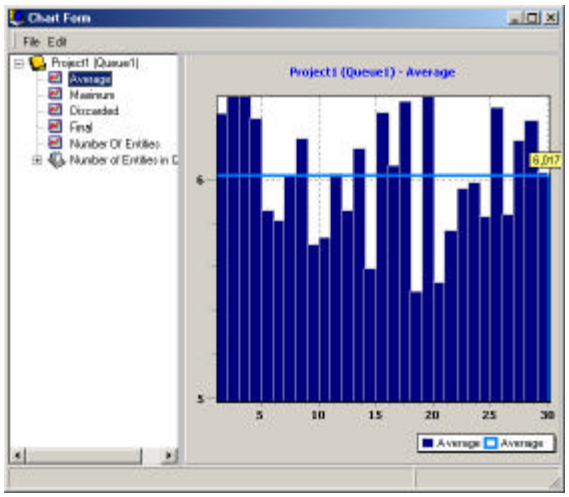

Fig. 3 – Statistic charts

3. Extending DEOS

DEOS provides a class library that allows software developers to extend the simulation environment defining new classes to model new objects that operate within the pre-existent environment and co-operate with other existing objects. Developers do not need to take explicitly care of synchronization issues related to the scheduling of events in the simulation timeline.

The classes provided to achieve such an abstraction can be grouped into three main areas:

- classes used to manage events
- classes that implement the timeline structure
- classes aimed at the description of simulation entities

Events are a main feature in any simulation tool and especially when dealing with discrete events models. Actually, in discrete events simulations, it is necessary to know the very instant when each event will occur within the timeline. The base class for events management is called *TSimulationEvent* and contains the information about the time when the event is to occur. In fact, a certain event may be generated before another one which has to occur first. It is up to the timeline structure, and therefore to the class that implements it, to process the events respecting the correct sequence.

The timeline structure allows the management and the execution of events according to a well-defined chronological sequence. It can be seen and modeled as a priority queue, where items, though inserted in any order, are always extracted following a certain parameter – in this case the time parameter. The timeline structure is provided by the DEOS class library through the *TTimeLine* class.

The classes needed to model simulation entities are all derived from a common ancestor - the *TZSimulationObject* base class. Each object involved in the simulation is linked to the timeline that manages the simulation through the *Timeline* attribute.

Several classes have been derived from the above mentioned basic classes in order to provide a functional simulation environment for the design tool.

Two important classes derived from *TSimulationEvent* using inheritance are *TStartWorkEvent* and *TEndWorkEvent*. These classes (see Fig. 4) extend the basic *TSimulationEvent* class in order to provide a model of events that occur when some resource starts working and when it completes its own job, keeping track of the resources involved in the accomplishment of the job.

The *TEntity* and *TResource* classes are inherited from *TSimulationObject* (see Fig. 5) and provide the abstraction to model the entities worked or processed by the resources involved in the simulation.

The *TResource* class has been designed in order to allow a black-box modeling of any machine with one or more inputs and outputs. When inheriting a specialized type of resource it is necessary to specialize all of its methods so that the new class behaves coherently with the simulation environment and implements the characteristics of the modeled resource. In particular, the behavior of a generic object derived from *TResource* should keep to the following scheme:

- a resource cannot work until its *WorkRequest* method is called by one of the input resources
- if the resource can execute the work-request, two events have to be generated and inserted into the timeline: a start-work event immediately after the work-request has been accepted and an end-work event some time after according to the particular behavior of the resource (that is the task of the *GetJobTime* method and *Expression* attributes)
- at the proper instant, the scheduled start-work and end-work event will invoke the *Work* and *EndWork* methods; the latter will then call the *WorkRequest* method of the output resources and then the *NextFree* method of every input resource connected in order to communicate its availability for further jobs
- the entity object worked by the resource is made available to other resources through the *GetOutput* method

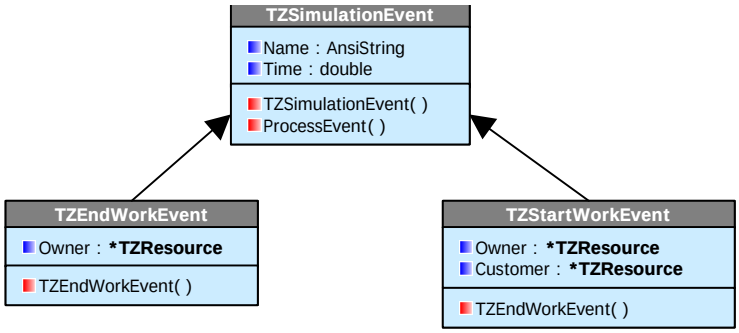

Fig. 4 – TSimulationEvent hierarchy

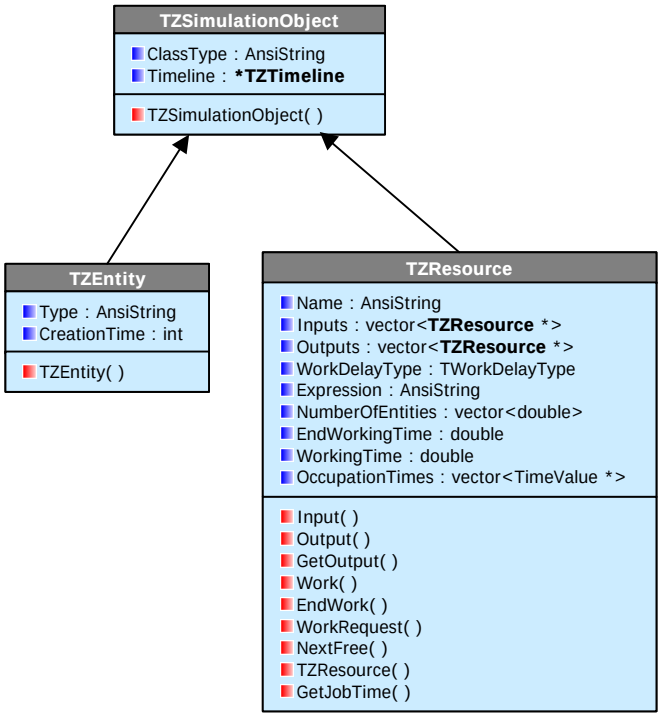

Fig. 5 – TSimulationObject hierarchy

4. The underlying architecture

The class library presented in the previous section can be used to build simulation models and to allow the execution of simulations. In order to make new classes available through the visual interface of the design tool, it is necessary to specialize a few classes that have been developed for this purpose.
These classes are designed to perform the following tasks:

- allowing to visually set up resources, entities and simulation parameters using the facilities of the MS Windows® OS
- management of statistics through both on-screen presentation and text file exporting
- input/output of simulation models layouts from/to file

For each new resource class created, an heir of *TClassManager* has to be implemented to manage the particular data structures needed by the resource, such as those involved in its behavior and those used to process the statistical information generated during the simulations.
The following classes (see Fig. 6) are also used by the design tool application:

- *TSimGraphicControl* – is the base class for resources and connectors painted on-screen
- *TsimulationForm* – inherits from *TForm* and manages objects of *TSimGraphicControl* type and allows their selection and displacement both individually and grouped, their deletion and the interconnection between a class manager and the object inspector
- *TClassBox* – inherits from *TSimGraphicControl* and manages the drawing of rectangles on the *TSimulationForm*; these are the graphical representation of resources. The class box may be moved or deleted and it

is possible to display and manage the properties of the resource through the object inspector

- *TClassConnector* – descends from *TSimGraphicControl* and its task is to connect two resources each other with a certain direction; a connector may be deleted but its movement is determined by the positions of the two resources that it connects
- *TobjectInspector* – is linked with *TSimulationForm* objects and with class managers in order to display the attributes of a certain resource. It contains two grids, one needed to display attributes and the other one used to show the simulation parameters (see Fig. 1). Besides it also contains two *combo-boxes* that contain respectively the list of objects present on the active simulation form and the list of values available for certain resource parameters (e.g. when it is possible to chose among several probability distributions)

Inprise C++ Builder® [Hol00][Rei99] development tool. A major effort will be made in future versions in order to grant a *plug-in structure* to DEOS, so that new classes (i.e. kinds of objects) might be defined and implemented using other languages and/or development tools and dynamically included within the design tool at run-time.

Another major issue that might be usefully developed in future versions of DEOS is its use as a real-time monitoring and control system. The complexity due to interfacing PCs with sensors and machines can be easily hidden to final users implementing suitable classes. These classes should manage interfacing issues and provide designers with a homogeneous environment. In such a context, representations of real objects connected to the PC would coexist with virtual objects that enable the user, or the system itself, to react to events generated by the monitored system.

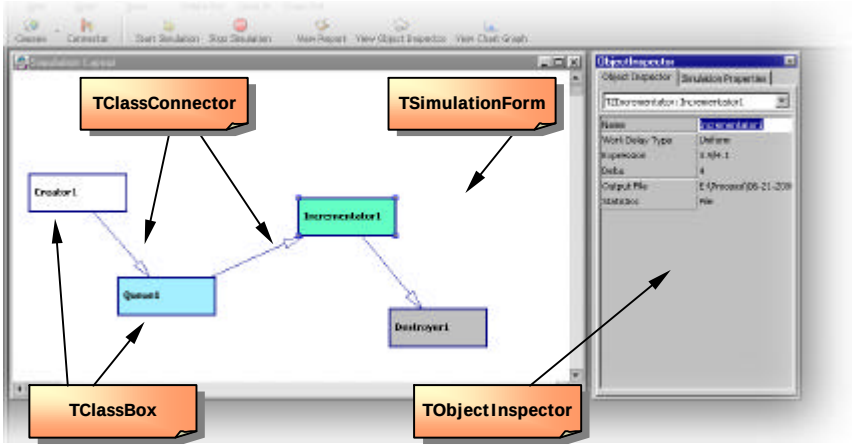

Fig. 6 – Support classes

5. Conclusions and future issues

The provided implementation of DEOS, though still in its early stage of development, shows many of DEOS potentialities.

The design tool, in spite of the few types of objects provided, points out all the versatility and usability of a visual tool, as well as the effectiveness and benefits of the graphical statistical analysis tools developed. This tool may be seen as somehow "complete", though some work is still to be done, for instance with regard to import/export and printing issues.

The class library has been designed and developed in order to make DEOS as extensible as possible. Nevertheless, the implementation of new kinds of objects requires a thorough knowledge of both standard ANSI C++ [Kal99] language and Borland/

Acknowledgements

This work has been partially funded by Ministero dell'Università e della Ricerca Scientifica e Tecnologica MURST, the National Research Council of Italy CNR.

References

Aes **SIMPLE++ reference manual** (Aesop GmbH Stuttgart)

Boo94 Booch **Object-Oriented Analysis and Design with Application** (Benjamin/ Cummings 1994)

Cac92 **MODSIM, The Language for Object-Oriented Programming Reference Manual**

(CACI Products 1992)

Cub97 Cubert, R. M., P. A. Fishwick
MOOSE: An Object-Oriented Multi-modeling and Simulation Application Framework
Submitted to *Simulation*, June 1997

Hlu99 V. Hlupic, R.J. Paul
Guidelines for Selection of Manufacturing Simulation Software
IIE Transaction, vol. 31 (1999), 21-29.

Hol00 Jarrod Hollingworth, Dan Butterfield, Bob Swart, Jamie Allsop
C++ Builder 5 Developer's Guide
(Trade Paper, 2000)

Kal99 Danny Kalev
Ansi/Iso C++ Professional Programmer's Handbook (Que Professional Series)
(Que, 1999)

Mac95 R. D. Macredie and R. J. Paul
Simulation modeling in manufacturing system design: an overview
International Journal of Manufacturing System Design, vol. 2, n. 3, (1995), 233-247

Omn OMNeT++ Discrete Event Simulation System
http://www.hit.bme.hu/phd/vargaa/omnetpp.htm

Pav99 Russell Pavlicek, Robin Miller
Embracing Insanity: Open Source Software Development
(Sams Pub.,1999)

Peg95 C.D. Pegden, R.E. Shannon, R.P. Sadowski
Introduction to Simulation Using SIMAN
(McGraw-Hill, 1995).

Rei99 Kent Reisdorph, Charlie Calvert
C++ Builder 4 Unleashed
(Sams Pub.,1999)