

EOSimulator: Event Oriented Simulator

User Manual v1.01

**Facultad de Ingenieria
Universidad de la República
June 2005**

Preface

For many years the Departamento de Investigación Operativa of Facultad de Ingenieria has been using Pascal_Sim¹ in the courses of Discrete Event Simulation. This department was looking for new simulation software which had common features with Pascal_Sim and a more modern design. It has to be a library that support event scheduling world view with event method or three phase approach. For that purpose, EoSimulator was created.

EoSimulator is inspired in two different simulation libraries: Pascal_Sim and DesmoJ². Those libraries were selected because they support the event scheduling world view. EoSimulator looks like Pascal_Sim with a modular design that reminds DesmoJ architecture. It is a hybrid between both libraries. There are some features that are not implemented in EoSimulator like antithetic streams or entities engaged in many activities. This was done on purpose in order to ask students to implement some of them.

EoSimulator was design and implemented by Sebastián Alaggia under supervision of Ing. Maria Urquhart and Ing. Antonio Mauttone.

This is a user manual, not an introductory text in system simulations. It is assumed that those concepts have been already acquired by the student. See references [1] and [2]. The student also has to have some basic knowledge in object oriented systems and in C++ programming language.

Acknowledgments

Many people have provided help with the development of this library:

- Javier Cohenar for supplying code of random number generators.
- Bruno Martínez and Fernando Pardo for helping with the implementation of the library.
- Maria Urquhart and Antonio Mauttone for guiding the project.

Notation

Class's and operation's names are shown in this *typeface*.

¹ Pascal_Sim: Discrete event simulation package [1].

² DesmoJ: Simulation framework created by the University of Hamburg – Department of Computer Science. See [3].

EoSimulator User Manual

Index

Chapter 1 - Getting Started.....	4
1.1 General Description	4
1.2 Installing EoSimulator	4
1.2.1 Using EoSimulator Library	4
1.2.2 Compiling the source code	5
Chapter 2 - Modeling in EoSimulator	6
2.1 Introduction	6
2.2 Model Building	6
2.2.1 Model	6
2.2.2 BEvent.....	7
2.2.3 CEvent.....	7
2.2.4 Entity	8
2.2.5 EntityComp	8
2.3 Collections	8
2.4 Distributions	9
2.5 Data Collectors	9
Chapter 3 - Simulation in EoSimulator.....	10
3.1 Introduction	10
3.2 Running a simulation	10
3.2.1 Making one run	10
3.2.2 Making multiple runs	10
3.3 Errors	11
3.3.1 Bin Errors	11
3.3.2 Collection Errors	11
3.3.3 Entity Errors	12
3.3.4 Experiment Errors	12
3.3.5 Histogram Errors	12
3.3.6 Model Errors.....	12
Chapter 4 - Future extensions.....	14
Chapter 5 - References	15

Chapter 1 - Getting Started

1.1 General Description

EoSimulator is a discrete event simulation library created for educational purposes. It is based in event scheduling world view and supports event method (two phase executive) and three phase approach (three phase executive).

EoSimulator is implemented in C++. It has a modular architecture divided in four modules: *Core*, *Dist*, *Statics* and *Utils*. Each of these modules encapsulates a specific part of the library.

The *Core* module contains both the system modeling classes and the simulator's engine. This module separates system modeling from running simulation using classes such as *Model* and *Experiment*. It's intended that the programmer first build a model and then simulates it as he pleases.

The *Dist* module contains pseudo-random number generators and distributions. Like every simulation library, EoSimulator contains some of the most common distributions (normal, log normal, negative exponential) and one random number generator. It also provides the user with simple interfaces in order to add more distributions and generators.

The *Statics* module contains output data collectors. EoSimulator implements two kinds of collectors, time series histograms and time weighted histograms.

The *Utils* module has the collections defined by the library. The library provides the programmer with built-in containers. Its interface is quite standard and easy to understand.

In the rest of this manual, we will explain how to model a system, use of containers, use of distributions, use of data collector and how to run a simulation. Finally, we describe some of the improvements that could be done to EoSimulator and references.

1.2 Installing EoSimulator

There are two ways to install EoSimulator: using the compiled library or compile the source code. EoSimulator has been compiled successfully with g++ in Windows and Linux.

1.2.1 Using EoSimulator Library

Use the library to compile your programs. Add the following flags:

- `-I< EoSimulator_Library_path>/include`
- `-L< EoSimulator_Library_path>/lib`
- `-leosim`

EoSimulator User Manual

1.2.2 Compiling the source code

EoSimulator's code is open, so you can compile it yourself. There are two installers, one for Windows and other for Linux, but they are very much alike.

To install EoSimulator you have to modify the makefile. Fill in correctly the fields:

- CXXINCS: Default includes of C++ and *std* includes.
- BUILD: The path where you are going to install EoSimulator Library.

Then execute the install batch file (install.sh in Linux or install.bat in Windows) adding the path where EoSimulator will be installed:

- `sh install.sh <EoSimulator_Library_path>` (Linux),
- `install <EoSimulator_Library_path>` (Windows).

Note: To compile the source code successfully, make sure that the *make* and *g++* command is set in the environment variables.

Chapter 2 - Modeling in EoSimulator

2.1 Introduction

As we said above, EoSimulator is divided in four modules: *Core*, *Dist*, *Statics* and *Utils*. In order to build a simulation, you have to implement subclasses of Core's abstract classes. Those abstract classes are *Model*, *BEvent*, *CEvent*, *Entity* and *EntityComp* (if needed). You may use collections defined in *Utils*, pseudo-random number generators or distributions from *Dist* or output data collector from *Statics*.

2.2 Model Building

This is a main activity in the process of simulating a system. EoSimulator provides the user with five abstract classes that have to be derived properly to build a specific system. For most models you have to implement:

- one subclass of *Model*,
- one subclass of *BEvent* for each bounded event in your system,
- one subclass of *CEvent* for each conditional event in your system if you are using three phase approach,
- one subclass of *Entity* for each kind of entity in your system
- as many *EntityComp* subclasses as priority queues your system have.

Although, there are some implementation issues that you must pay attention while modeling your system with EoSimulator, otherwise it won't run properly or it won't run at all. If you don't follow these instructions and try to execute a simulation, it might abort displaying an error message.

2.2.1 Model

Model is an abstract class which represents a model of the system that we are going to simulate. It contains attributes such as bounded and conditional events, global entities, waiting queues, arrival distributions, service distributions and output data collectors. This class is responsible of:

- containing and registering every attribute,
- defining the initial state of the system,
- give access to those attributes which are required for both bounded and conditional events.

As we said above, *Model* is an abstract class. So, to create a specific model, we have to implement a subclass of *Model*. To do that we must declare every attribute and implement four methods: a constructor, a destructor (because EoSimulator is written in C++), *init* and *doInitialSchedules*.

Attributes such as events (bounded or conditional), distributions, containers and data collector have to be created, stored and deleted by the model. The library just keeps references to those attributes and doesn't delete any of them. Besides, these attributes could be created either statically or dynamically. For global entities see section 2.2.4.

EoSimulator User Manual

Then, we have to give a method to *init*. In this operation we must create (if we haven't done it yet) and register every attribute using the register operations defined in *Model*. If you don't register an attribute, it won't work properly. For more information, see [4].

Finally, *doInitialSchedules* defines the system initial state. Here you schedule the first entities to the corresponding bounded events. To schedule an entity use the operation *schedule* with the entity, the time when it is schedule (indicated by a time offset) and the name of the event. The last parameter is very important, it is the key used by the model to find the right event. On return, the entity is inside the calendar, pointing to the *BEvent* that it was scheduled and with the time when the event is going to happened.

2.2.2 *BEvent*

BEvent is an abstract class which represents a bounded event that is relevant to a system. *BEvents* are stateless objects; they must not have any attribute that change over a simulation. They are just treated as a sample of code which is executed by the entities. Remember that *BEvents* are *Model*'s attributes.

Like in *Model*, to create a subclass of *BEvent* you must implement three methods: a constructor, a destructor and *eventRoutine*.

Every *BEvent* has a *name* and belongs to a *Model*. *name* is very important because it is the key used by *Model* to schedule an entity. Besides, *BEvents* have the attribute *owner* (whose static type is *Model*) to have access to its model. These parameters appear in *BEvent* constructor and it must be called at the constructor of any subclass. It could be a good idea to keep all names in a separate header file in order to make fewer mistakes scheduling entities.

eventRoutine is the main operation of this class. It defines every action taken in that particular point in time. *BEvents* acts over one specific entity, and may change the state of the model. If one event creates an entity, follow the rules decrypted in section 2.2.4.

2.2.3 *CEvent*

CEvent is an abstract class which represents a conditional event that is relevant to a system. *CEvents* are stateless objects and *Model*'s attributes as well.

To create a subclass of *CEvent* you must implement three methods: a constructor, a destructor and *eventRoutine*. *CEvents* are very much alike *BEvents*, the only difference is that *BEvents* occur at a certain time to a certain group of entities. But, *CEvents* occur when its model meets certain conditions such as resources availability. That's why *BEvent::eventRoutine* is applied to an entity and *CEvent::eventRoutine* is applied to its model.

All *CEvents* belong to a *Model*. Like *BEvents*, they have the attribute *owner* (whose static type is *Model*) to have access to its model. This parameter appear in *CEvent* constructor and it must be called at the constructor of any subclass

EoSimulator User Manual

eventRoutine defines every action taken in this particular point in time. *CEvents* acts over many unspecific entities, and may change the state of the model. If one event creates an entity, follow the rules decryped in section 2.2.4.

2.2.4 Entity

Entity is a class that represents objects or components of the system whose activities are modeled. EoSimulator provides a primitive entity which has both a *BEvent* and a time stamp that indicates the entity's last bounded event, the *clock*. Those are private members and have restricted access (see [4]). If your model needs a more specific entity, just implement a new subclass.

There's a particular issue when it comes to entities: They must be created dynamically and the user is not supposed to delete them. Why? Because when a simulation ends, every entity is scheduled or waiting in a queue, so they are in an EoSimulator's container. Entity containers have pointers to its elements, in order to avoid copies. When their destructor is called, it destroys every entity stored. This policy is imposed by EoSimulator but it must be controlled by the user. For more information about containers, see section 2.3.

2.2.5 EntityComp

EntityComp is the only EoSimulator's abstract class that is used optionally. Its main operation is *compare*, which compare two entities and is abstract. These comparators are user defined classes. There are used to order queues. For more information, see [4] and section 2.3.

2.3 Collections

EoSimulator implements a number of collections that are used in the simulator and can be used by the programmer to model his system. These collections are based in C++ STL containers (C++ Standard Template Library) which is a standard supported by every C++ compiler. But this fact is transparent for the user. EoSimulator collections are located in the *Utils* module.

Collections in EoSimulator offer an easy interface and some error control. Although there are some differences between them, their design is very similar. These containers are typed; you can't put a *BEvent* in an *EntityQueue*. And they store pointers to the contained objects, to avoid copies.

Another common feature is the access methods. All containers provide random access to its elements. They are indexed by an *int* (*CEventVector*, *DistVector* and *EntityQueue*) or by a key (*BEventMap*). For example:

```
Entity* ent = queue[5];  
BEvent* bev = map["llegada"];
```

gets the 5° element in the EntityQueue queue and a BEvent whose name is "llegada" from a BEventMap. Also you can remove elements by giving the proper type of index. In

EoSimulator User Manual

case of indexing outside the container, EoSimulator immediately aborts displaying an error message.

All containers are created empty and ready to use, but when it comes to destructors things are a little bit complicated. As we said above, most model attributes could be created statically or dynamically but is responsibility of the user to delete them. On the other hand, entities must be created dynamically and EoSimulator deletes them. *EntityQueue* and its subclasses delete every pointer they store, because they assumed that all entities are created dynamically. *CEventVector*, *DistVector* and *BEventMap* don't make any assumptions so the programmer takes charge. For more information about container see [4].

2.4 Distributions

Pseudo-random number generators and distribution play a critical roll in computer simulations. Good generators ensure randomness between runs. So every simulation library must provide these facilities. EoSimulator implements some of the most common distributions and provides one generator, but its design allows more than one generator.

EoSimulator has no fixed number of streams. When a distribution is created, a new generator of user defined type is assigned to it. The generator's type is defined by a *GenType* label (*GenType* is declared together with the *NumberGenerator* interface). The generator is a Distribution's private attribute.

To get a sampled value from a distribution, invoke the operation *sample* on any Distribution's subclass (*LogNormalDist*, *NegexpDist*, *NormalDist*, *PoissonDist* or *UniformDist*). You can also change the generator's seed with the operation *setSeed*. EoSimulator doesn't support antithetic streams, but it can be easily implemented. For more information see [4].

2.5 Data Collectors

Another important part of simulation software is output data collectors. They are used to collect data from simulation's output parameters. Later, these data are analyzed statistically. The data collectors provided by EoSimulator are histograms (*TimeSeries* and *TimeWeighted*).

Histograms register the evolution of an output parameter in a run: for example queue lengths over time. A histogram only records *x*'s values in an interval *I*, which is divided in subintervals whose length is less or equal than *l*. So they must be created with a four parameters: the interval *I* defined by a lower bound and an upper bound, the length *l*, and a name. This data collector also calculates the mean value and the standard deviation of the collected data.

Histograms must be registered before used, like any other model attribute. Data are collected using the operation *log*. Histograms can be printed also. There are many ways of printing them: the operation *print()* prints the histogram in the standard output, *print(path)* prints it in the file whose path is *path*, *table()* prints a table with the values stored in the histogram in the standard output, and *table(path)* prints it in a file.

Chapter 3 - Simulation in EoSimulator

3.1 Introduction

At the moment we have only studied how to make model a system with EoSimulator. In this section we will learn how to simulate a model using the facilities provided by the library. Besides, we will describe the errors detected by EoSimulator.

3.2 Running a simulation

The process of simulate models is not as trivial as it appears. It is very important to design set of runs varying seeds in those streams used by the model. Otherwise all the experimentation done with the model will not be statistically valid. Remember, one simulation run is not enough to draw any conclusions about a system.

Luckily, EoSimulator makes this running process a little bit easy. The *Experiment* class handles the execution of a simulation. It contains a calendar, an executive and the streams.

3.2.1 Making one run

To run a simulation you have to write a program. In this program, create an instance of your system model and an experiment. Then connect them with the operation *Model::connectToExp*. In that operation, the model is initialized (with the operation *Model::init*) and connected to the experiment. Then use *Experiment::run* to run the simulation. This operation makes the initial schedules in the model (*Model::doInitialSchedules*) and simulates the model.

After the simulation is finished, print the histograms (the output data) and the program ends.

3.2.2 Making multiple runs

As we said above, one run is not enough to draw conclusions. So in order to make more runs, after the simulation is finished, create a new model and experiment. Then connect them and set a new seed in the experiment using *Experiment::setSeed*. This operation change the seed of every registered distribution in the model, and it let you have a different run (the default seed is 251182). Finally, run the simulation (*Experiment::run*).

But why we didn't keep the old experiment and model? Another possibility may be keep them, set a new seed and simulate again. This is possible because *Experiment* doesn't reset the simulation time. Check if the system you are modeling let you do that.

We could create a new model and connect it the old experiment. But, in the old experiment there may be some entities which are scheduled to arrive or to leave the system. If an entity is schedule to leave, it may return some units of bins causing errors in the output or even aborts the simulation. If we make a new experiment and connect it with an old model, the queues of the model will full of entities. In this case the model has already started (and if it can reach steady state) it may be stable. But, the clocks of the

EoSimulator User Manual

entities waiting in a queue have not been updated³, so they still contain older values and cause errors collecting waiting times.

3.3 Errors

EoSimulator control some common errors during execution, using assertions. When an assertion occurs, EoSimulator aborts the simulation displaying an error message. This message shows where assertion occurs, in which file and operation. In the next sections we are going to give advice about what to do when some errors occur.

3.3.1 Bin Errors

- 1 *Bin::Bin*: An error in the constructor of a bin only occurs when the initial quantity of the bin is negative.
- 2 *Renewable::Renewable*: This error occurs when the initial quantity of the bin is negative or greater than max. If quantity is negative, the error message shown is the error of Bin's constructor.
- 3 *Renewable::returnBin*: This error occurs when the units of bin that are returned overcome the max capacity of the bin. Check events that manipulate renewable bins.
- 4 *Bin::acquire*: This error occurs when you try to acquire more units than those which are available. Before invoking this operation, check the availability with *Bin::isAvailable*.
- 5 *NonRenewable::addBin*: This error occurs when you try to add negative units the bin, which means taking units. Use *Bin::acquire* instead.

3.3.2 Collection Errors

- 1 *operator[]*: These errors occur while iterating over a collection. It could happen because you have indexed outside the bounces of the collection (*EntityQueues*, *CEventVector* or *DistVector*) or with an unregistered key (*BEventMap*). Check those methods which iterate over the collection's type that matches the error.
- 2 *EntityQueue::pop*: These errors occur while getting the first element of a queue and removing it. The error occurs when the queue is empty. Another error which is common (and doesn't abort the simulation) is when you pop elements from ordered queues. The order is only preserved by the comparator passed in the constructor of the queue. If the elements don't follow the order you need, check the comparator. See [4].
- 3 *BEventMap::put*: These errors occur when you register two *BEvents* with the same name. Check that all *BEvents* have different names. See the suggestions given in section 2.2.2. Also check if you are registering *BEvents* only once in a *Model*.

³ The new experiment will start with time zero.

EoSimulator User Manual

- 4 *remove*: These errors occur while removing elements from a collection. The causes of that are exactly the same described in the point 1 of this section (bounces errors).

3.3.3 Entity Errors

- 1 *setBEvent*: This error occurs when a null *BEvent** is assigned to an Entity. This operation is used by EoSimulator and is involved with *Model::registerBEvent*. So it is an uncommon error, unless you invoke it, which is not recommendable.

3.3.4 Experiment Errors

- 1 *run*: This error occurs when the experiment is not connected to a model.
- 2 *setModel*: This error occurs when a null *Model** is assigned to the experiment. This operation is used by EoSimulator and is involved with *Model::connectToExp*. So it is an uncommon error, unless you invoke it, which is not recommendable.
- 3 *setSeed*: This error occurs when the experiment is not connected to a model.
- 4 *registerDist*, *registerCEvent*: This error occurs when the experiment is not connected to a model. This operation is used by EoSimulator and is involved with registration of attributes. So it is an uncommon error, unless you invoke it, which is not recommendable.

3.3.5 Histogram Errors

- 1 *Histogram::Histogram*: This error occurs if the main interval is badly defined (lower is greater then upper) or if the length of the subintervals is negative.
- 2 *log*: This error occurs when you try to log data to a non-registered histogram. *Histograms* have to be registered before used.
- 3 *setModel*: This error occurs when a null *Model** is assigned to the histogram or it is already registered to another model. This operation is used by EoSimulator and is involved with *Model::registerHistogram*
- 4 *printH*: This error occurs if you try to print an empty histogram. Although this operation is private, it is invoked by both *print()* and *print(const char*)*.
- 5 *printT*: This error occurs if you try to print an the table of an empty histogram. Although this operation is private, it is invoked by both *table()* and *table(const char*)*.
- 6 *getMaxX*, *getMinX*, *getMaxY*, *getMinY*, *getVariance*, *getMean*, *getName*: These errors occur if you try to get one of these values from an empty histogram.

3.3.6 Model Errors

- 1 *connectToExp*: This error occurs when a null *Experiment** is assigned to the experiment. If you are going to run multiple times a model and an experiment (you

EoSimulator User Manual

don't create new ones) connect them only once. Remember that this operation invokes *Model::init* so if you connect a model with more than one experiment some error may occur while registering attributes because some of them are register in the model an others in the experiment.

- 2 *registerBEvent*, *registerCEvent*, *registerDist*, *registerHistogram*: These errors occur mainly if the model is not connected to an experiment. But if the model is connected, some other errors may abort the simulation, see *BEventMap::put* and *Histogram::setModel*. They also may occur if you register a model to more than one experiment.
- 3 *schedule*: This error occurs when either a null *Entity** is given, the time offset is negative or the model is not connected to an experiment. Although all of this paramenters may be right, if there aren't any BEvent whose name is what_, a *BEventMap::operator[]* error will occur.
- 4 *getSimTime*: This errors occur when the model is not connected to an experiment.

Chapter 4 - Future extensions

Although it has some interesting features, EoSimulator is a very basic simulation library. Some features like antithetic streams or schedule entities to multiple events are not supported by EoSimulator. Those extensions are left to future designers or students. Here we mention some of them:

- Support the antithetic variables method for variance reduction;
- Incorporate more pseudo-random number generators;
- Incorporate more distributions;
- Support schedule entities to multiple events;
- Give some visual output interface;
- Give a more optimized calendar;
- And any other extension needed.

Chapter 5 - References

- [1] O’Keefe, R [1989] “Simulation modeling with Pascal” ISBN 0-13-811571-0.
- [2] Law, A [1991] “Simulation Modeling & Analysis” ISBN 0-07-100803-9
- [3] DesmoJ: A Framework for Discrete-Event Modeling and Simulation
<http://www.desmoj.de/>
- [4] EoSimulator Programmer’s Manual