

EOSimulator: Event Oriented Simulator

Developer 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.

It is assumed that the developer has already acquired concepts in simulations and object oriented programming in C++. See references [1], [2] and [6].

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 Developer Manual

Index

Chapter 1 - Static Architecture	4
1.1 Macro Architecture	4
1.2 Core Module	5
1.2.1 Design Restrictions.....	6
1.2.2 Implementation Restrictions.....	6
1.3 Dist Module	7
1.3.1 Design Restrictions.....	7
1.3.2 Implementation Restrictions.....	7
1.4 Statics Module	8
1.5 Utils Module	9
1.5.1 Implementation Restrictions.....	9
Chapter 2 - Collaborations	10
2.1 Introduction	10
2.2 Model::connectToExp	10
2.3 Model::registerBEvent	11
2.4 Model::registerCEvent	12
2.5 Model::registerDist	13
2.6 Model::registerHistogram	14
2.7 Experiment::setSeed	15
2.8 Experiment::run	16
2.9 Model::schedule	18
2.10 Bins Collaborations	19
2.11 Distribution Constructor	20
Chapter 3 - References	21

Chapter 1 - Static Architecture

1.1 Macro Architecture

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.

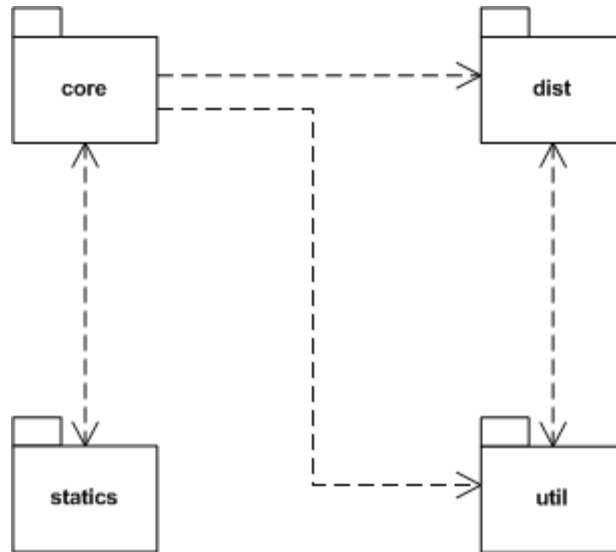


Figure 1-1: EOSimulator Macro Architecture

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. EOSimulator contains some of the most common distributions (normal, log normal, negative exponential) and one random number generator. It also designed to be extensible: distributions and generators can be easily added.

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.

EOSimulator Developer Manual

1.2.1 Design Restrictions

- Entities are either scheduled for a BEvent (contained in BCalendar) or waiting in a queue.
- The relation between Model and Bins, CEvents, Distributions and Histograms is through attributes: Model has one attribute for each of them. BEvents are attributes too, but they are stored in a collection in the Model.

1.2.2 Implementation Restrictions

- Entities have to be created dynamically. BEvents Bins, CEvents, Distributions and Histograms could be created dynamically or statically but they must be attributes of a subclass of Model. For more information see [5].

1.3 Dist Module

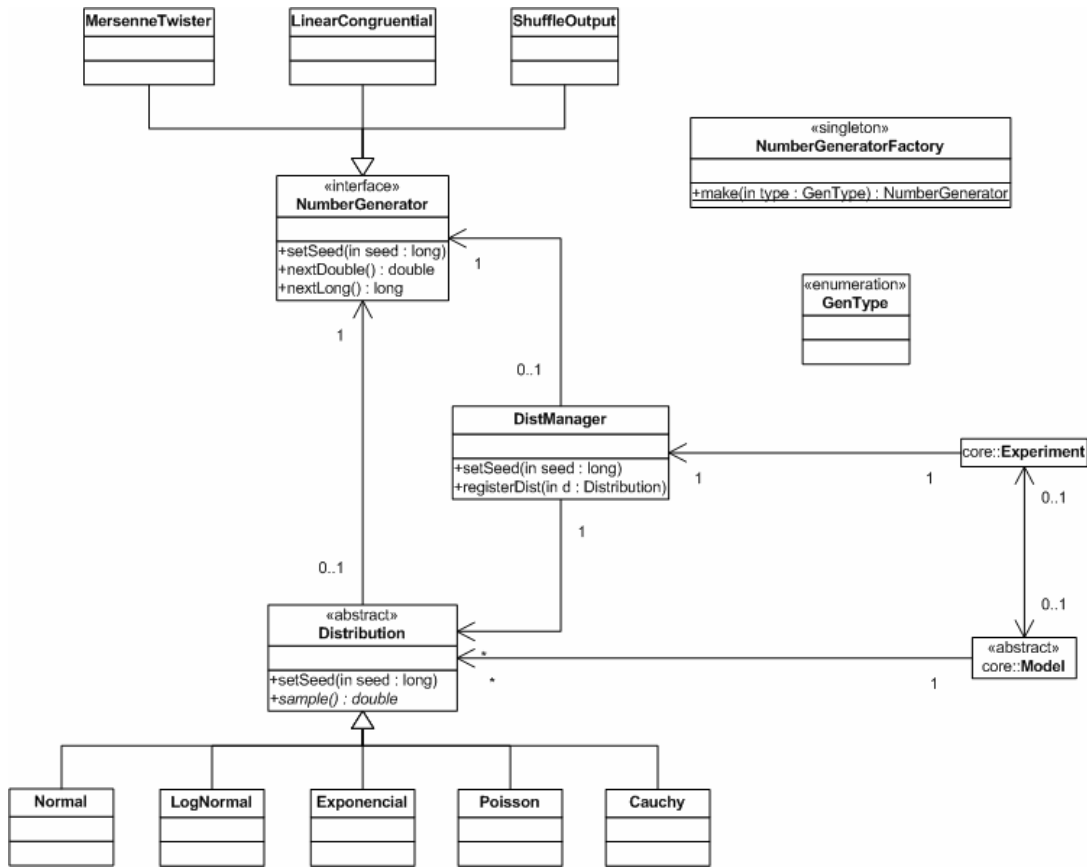


Figure 1-3: The Dist Module

In this module we have encapsulated pseudo-random generators and distribution sampling. All distributions are subclasses of *Distribution*, an abstract class which defines the behavior of every distribution in EOSimulator. Every distribution has a pseudo-random number generator, defined by the interface *NumberGenerator*.

Finally *NumberGeneratorFactory* creates *NumberGenerators* according to the corresponding *GenType* label. This design allows the user to define both new generators and distribution.

1.3.1 Design Restrictions

- To add new generators, *GenType* has to be modified giving a new label for the generator. Also *NumberGeneratorFactory::make* has to be modified to accept the new label.

1.3.2 Implementation Restrictions

- Distributions are responsible for deleting their generators when they are deleted.

1.4 Statics Module



Figure 1-4: The Statics Module

The data output is stored in the Statics's classes. The class *Histogram* defines the behavior of every histogram in EOSimulator. *TimeSeries* and *TimeWeighted* are both subclasses of *Histogram* whose operations differ very little from their parent class. These classes are particularly simple and collaborate only with *Model*, in order to be registered and don't present any particular restrictions.

1.5 Utils Module

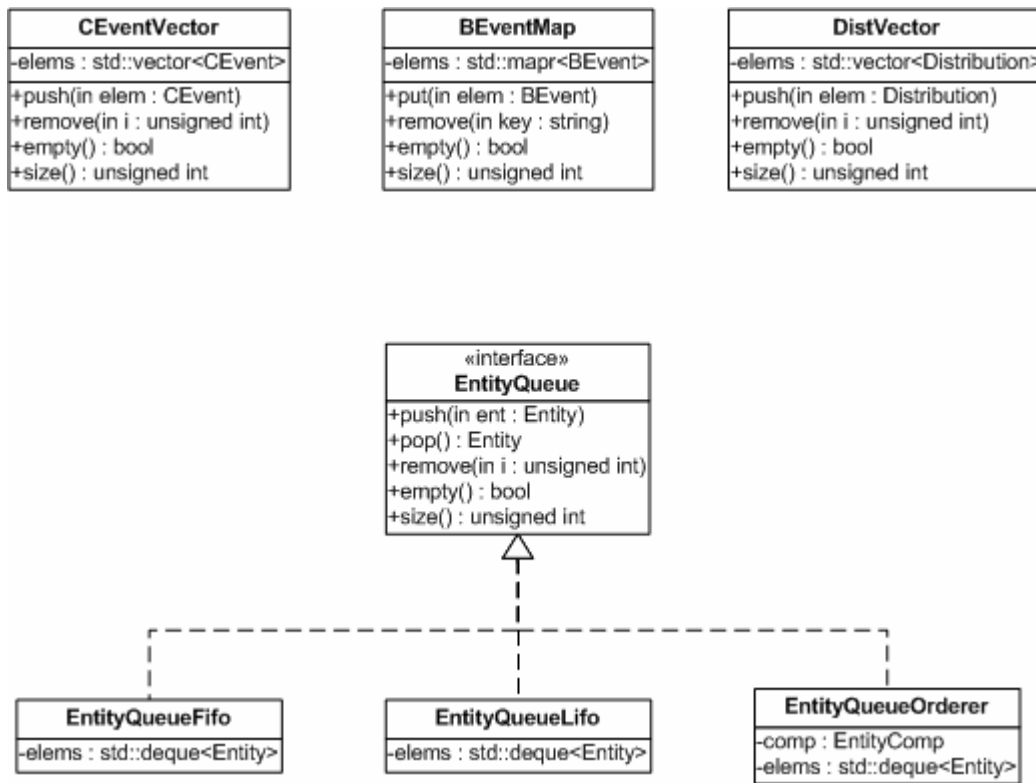


Figure 1-5: The Utils Module

The Utils Module contains the collections used in EOSimulator. They are based on C++ standard containers (*vector*, *deque* and *map*). The collections defined are basically wrappers; they hide the use of these containers from the user. Because these containers support random access, EOSimulator collections also support random access with *operator[]* efficiently.

1.5.1 Implementation Restrictions

- Although `EntityQueueOrdered::pop` is $O(1)$ due to `std::deque` implementation, the insertion algorithm is $O(n)$.
- The operation given by `EntityQueue` relies on the efficiency of the C++::std containers used. Changing the container may speed up some `EntityQueue` operations and delay others. The implementation given is very efficient for iterating in every queue and inserting forward and backward, but not in the middle.

Chapter 2 - Collaborations

2.1 Introduction

In this section we show the more important collaborations in EOSimulator. For each one we show a diagram, a brief description and the preconditions of the collaboration. The preconditions apply to the complete operation; this means the operation itself and the operations it unleashes, so they could be checked in more than one class. Preconditions are checked with the assert function which aborts the program if the condition fails.

2.2 Model::connectToExp

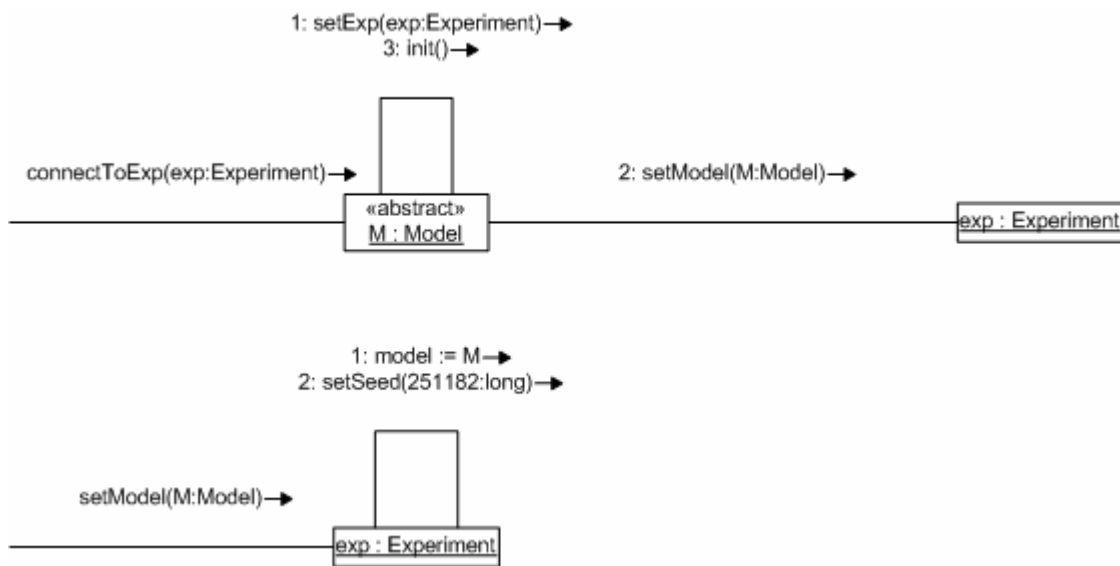


Figure 2-1: Model::connectToExp Collaboration Diagram

2.2.1 Description

This collaboration describes how a model and an experiment are connected. Notice that the abstract operation `Model::init()` is called in order to register the model's attributes. When `exp` sets `M` as his Model, it assigns `251182` as a default seed.

2.2.2 Precondition

- `exp` has to be a valid instance, this means not null.

2.3 Model::registerBEvent

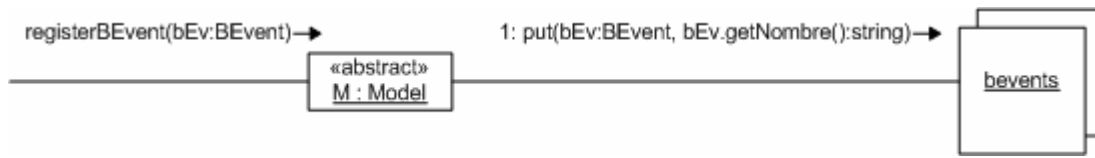


Figure 2-2: Model::registerBEvent Collaboration Diagram

2.3.1 Description

This operation registers a BEvent in its model. The container *bevents* is an instance of *BEventMap*. To insert *e* into *bevents*, *bevents* takes *e*'s name and use it as a key.

2.3.2 Precondition

- *M* has to be previously connected to an experiment in order to complete the operation.
- The name of *bEv* has to be different from any other BEvent registered in *bevents*.

2.4 Model::registerCEvent

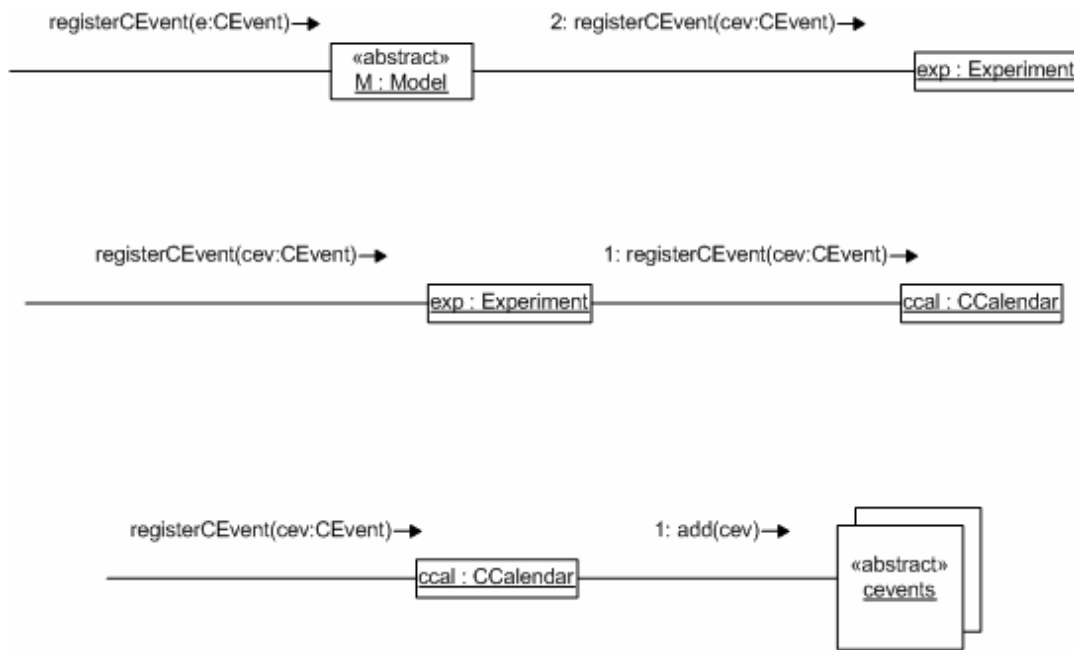


Figure 2-3: Model::registerCEvent Collaboration Diagram

2.4.1 Description

This collaboration shows how a *CEvent* is register in a model. Notice that *M* doesn't keep any reference to *e*, because it doesn't need it. CEvents are finally stored in *exp*'s *CCalendar*. *CCalendar* is the only class which contains all CEvents because its responsibility is to iterate over them after every B phase.

2.4.2 Precondition

- *M* has to be connected to an experiment.

2.5 Model::registerDist

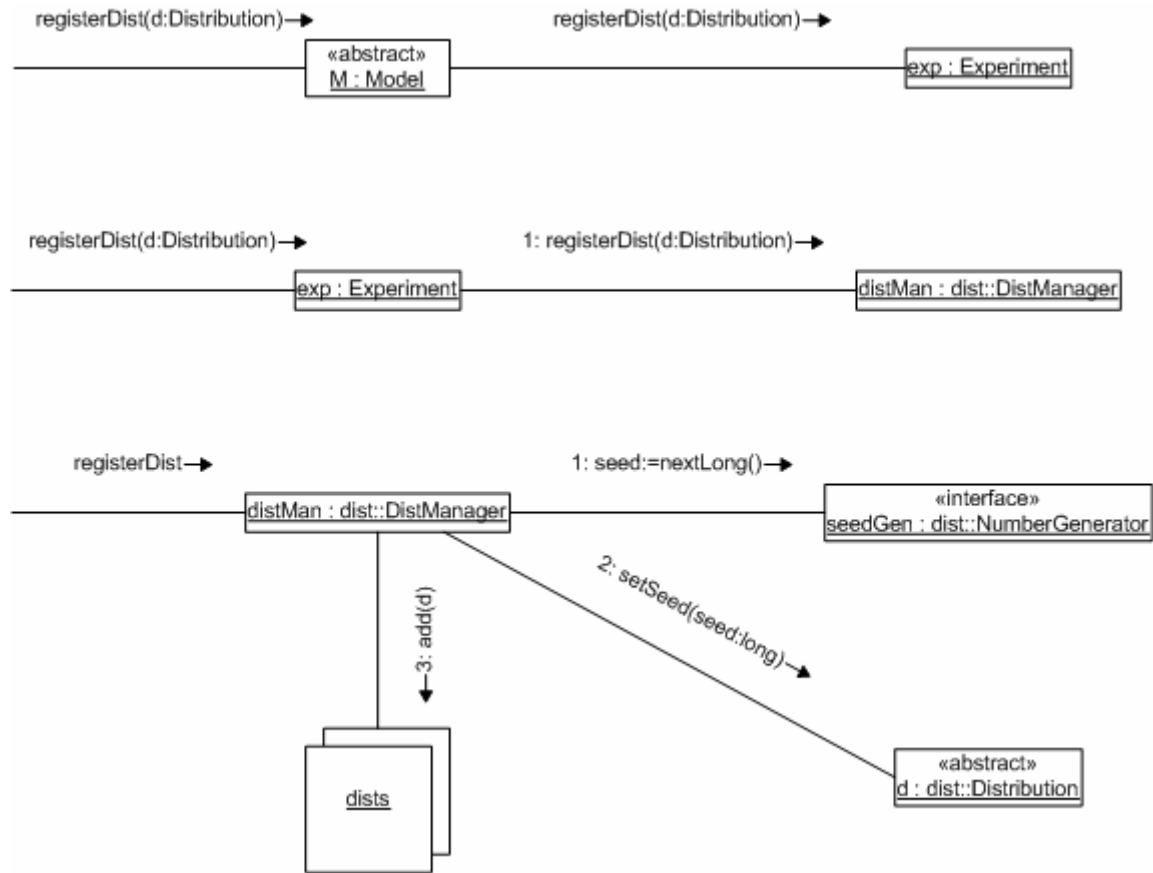


Figure 2-4: Model::registerDist Collaboration Diagram

2.5.1 Description

This collaboration shows how a *Distribution* is register in a model. Like in CEvent, *M* doesn't keep any reference to *d*, because it doesn't need it. All Distributions are finally stored in *exp*'s *DistManagerCCalendar* who initialized them with a new seed taken from *seedGen*. *DistManager* contains all *Distribution* to provide a more centralized stream management.

2.5.2 Precondition

- *M* has to be previously connected to an experiment.

2.6 Model::registerHistogram



Figure 2-5: Model::registerHistogram Collaboration Diagram

2.6.1 Description

This collaboration shows how a *Histogram* is register in a model. Like in CEvent, *M* doesn't keep any reference to *H*, only *H* has a reference to *M*.

2.6.2 Precondition

- *M* has to be previously connected to an experiment.

2.7 Experiment::setSeed

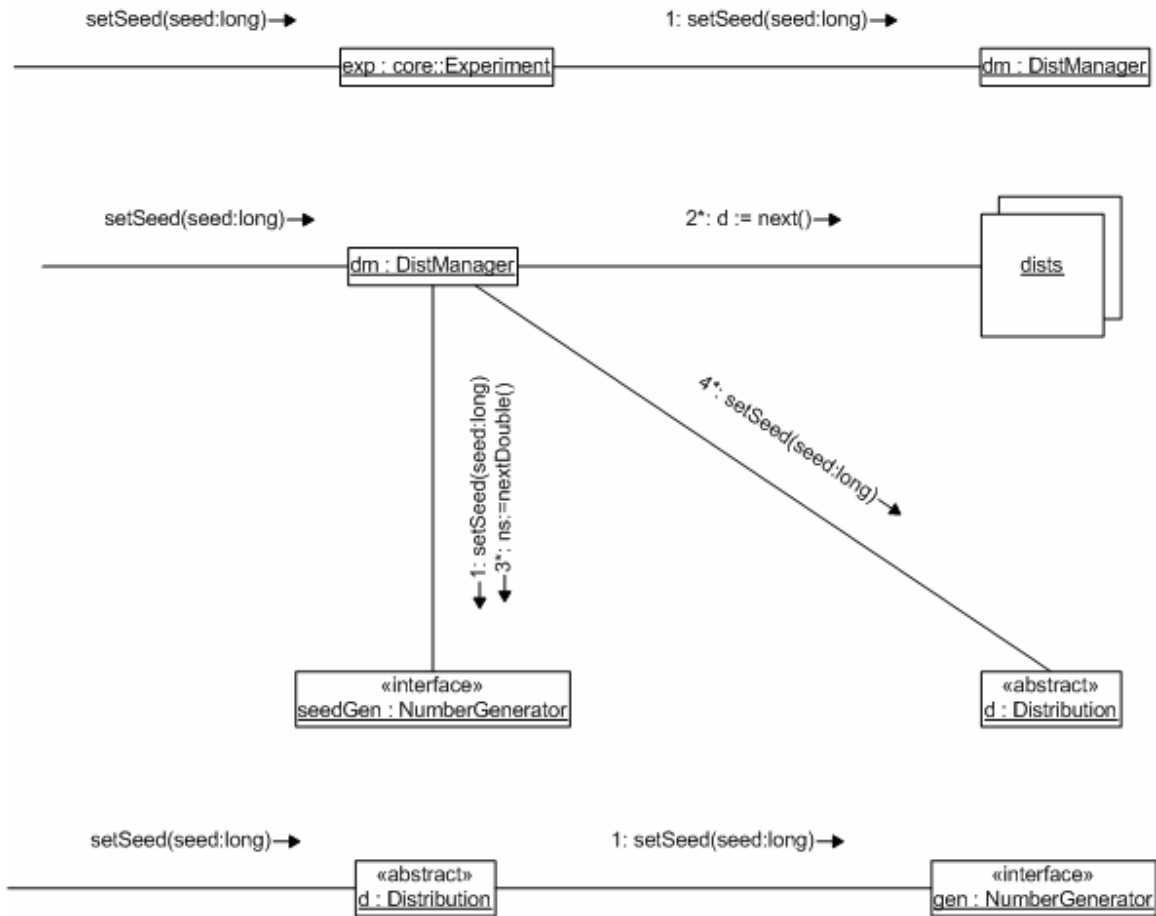


Figure 2-6: Experiment::setSeed Collaboration Diagram

2.7.1 Description

This diagram shows the process of changing a new seed in every Distribution and stream (which are the same). As we see, `exp` calls `dm::setSeed()` and it changes the seeds in every stream, using `seedGen` and `seed`.

2.7.2 Precondition

- `exp` has to be connected to a model.

2.8 Experiment::run

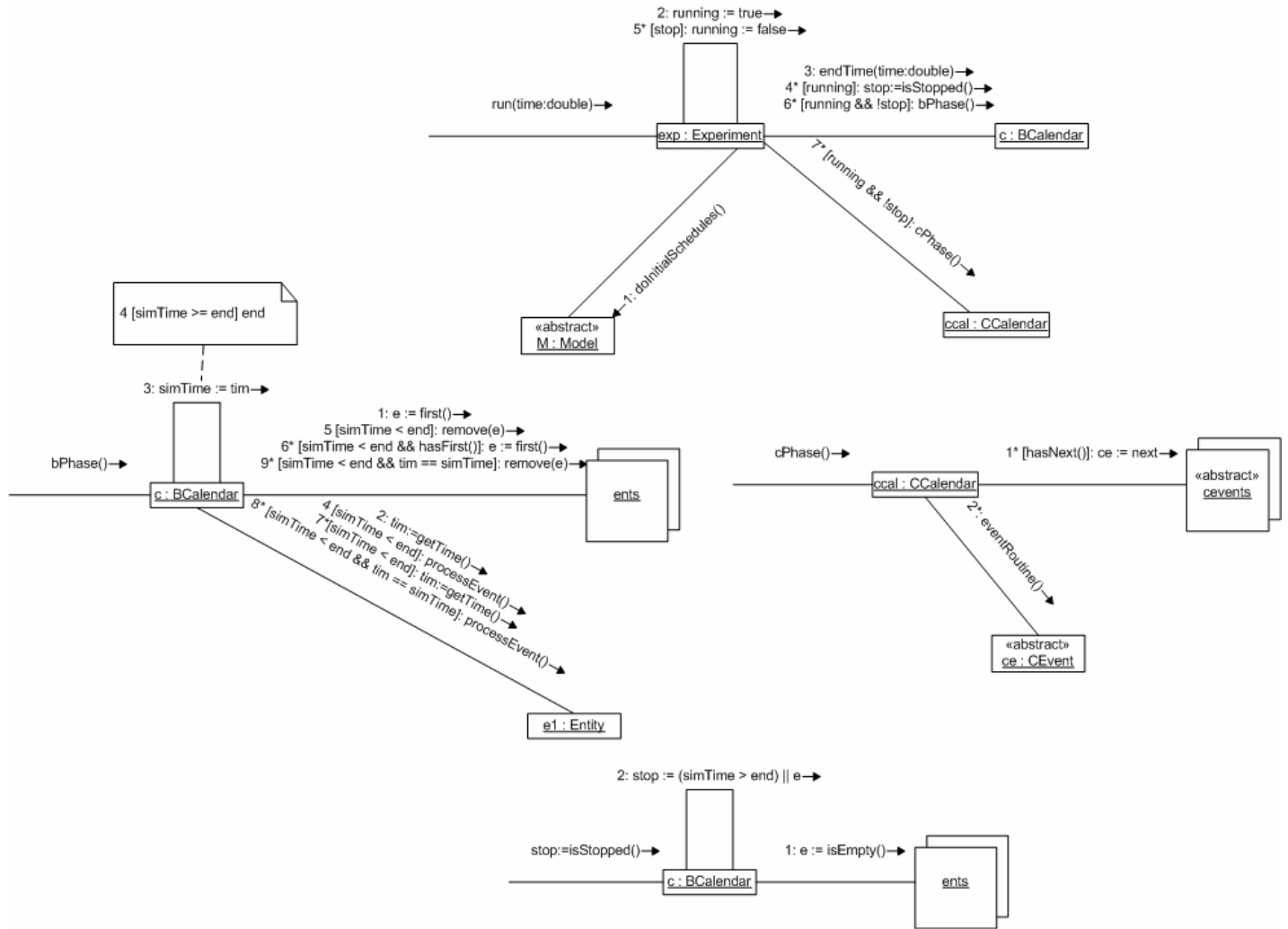


Figure 2-7: Experiment::run Collaboration Diagram

2.8.1 Description

This diagram shows how a simulation is run in EOSimulator. First *M::doInitialSchedules* is called in order to start the model. Then the end time is stored and the simulation starts. The main loop consist in checking if the simulation is finished (*BCalendar::isStopped*), then the A phase and B phase are executed (*BCalendar::bPhase*) and finally the C phase is executed (*CCalendar::cPhase*).

In *BCalendar::bPhase* the first entity in the queue is removed the simulation actual time is advanced up to the entity clock. Then every entity whose clock is equal to the actual simulation time is processed (*Entity::processEvent*).

EOSimulator Developer Manual

In *CCalendar::cPhase* every *CEvent* registered is executed (*CEvent::eventRoutine*).

Finally in *BCalendar::isStopped* the actual simulation time is compared to the end time and the queue of scheduled entity is checked to be empty.

2.8.2 Precondition

- *exp* has to be connected to *M*.
- *time* has to be greater or equal to the simulation actual time³ (*BCalendar::getSimTime*).

³ Remember that you could use an experiment which is connected to model and used before. See [4] and [5]

2.9 Model::schedule

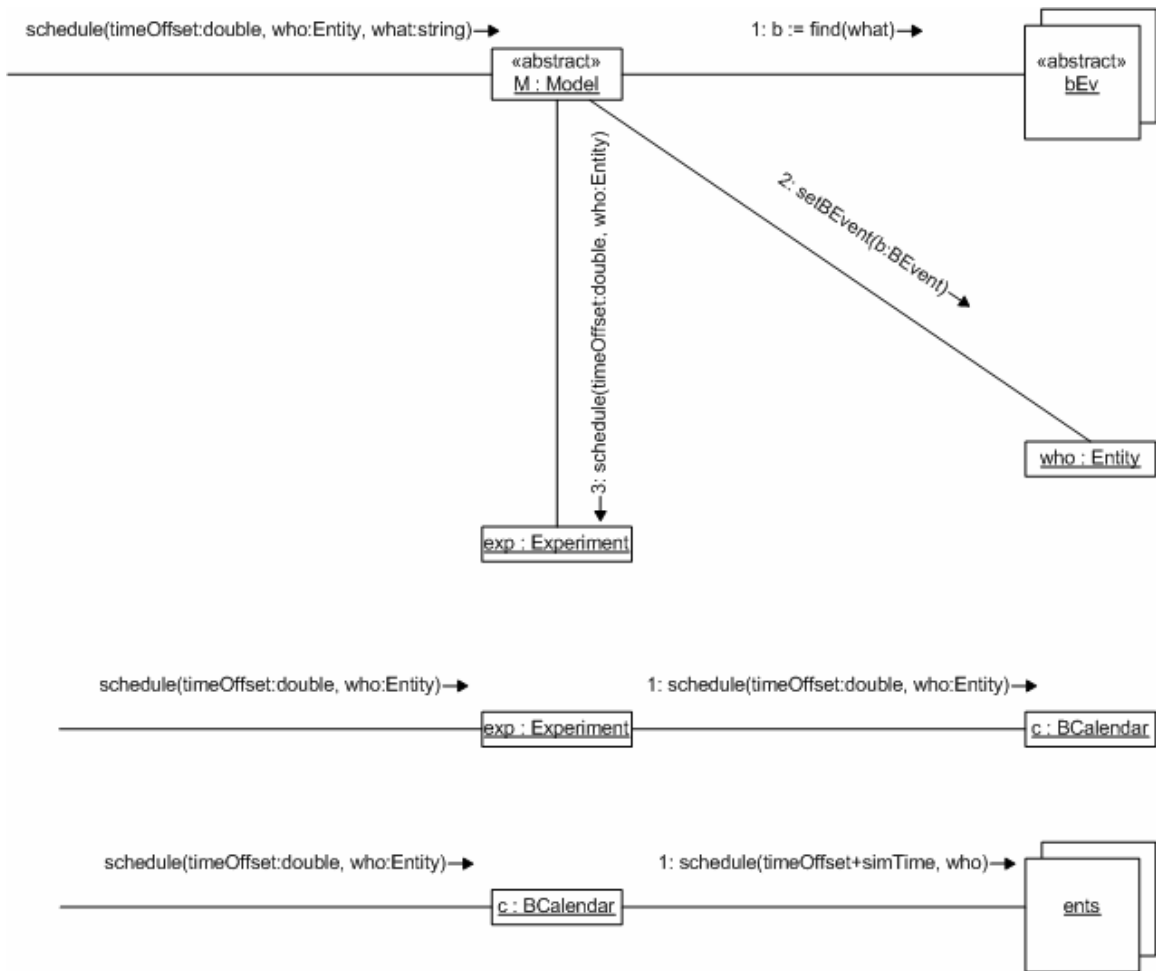


Figure 2-8: Model::schedule Collaboration Diagram

2.9.1 Description

To schedule the entity *who* given a time offset *offset* and a BEvent's name *what*, *M* fetches the BEvent whose name is *what*, assigns it to *who* and finally passes *who* to *exp*. Then *exp* passes *who* to *c* which schedules *who* correctly according to *offset*.

2.9.2 Precondition

- *M* has to be connected to an experiment.
- *M* has to have a BEvent whose name is *what* and *offset* has to be positive.
- *offset* has to be positive.

2.10 Bins Collaborations

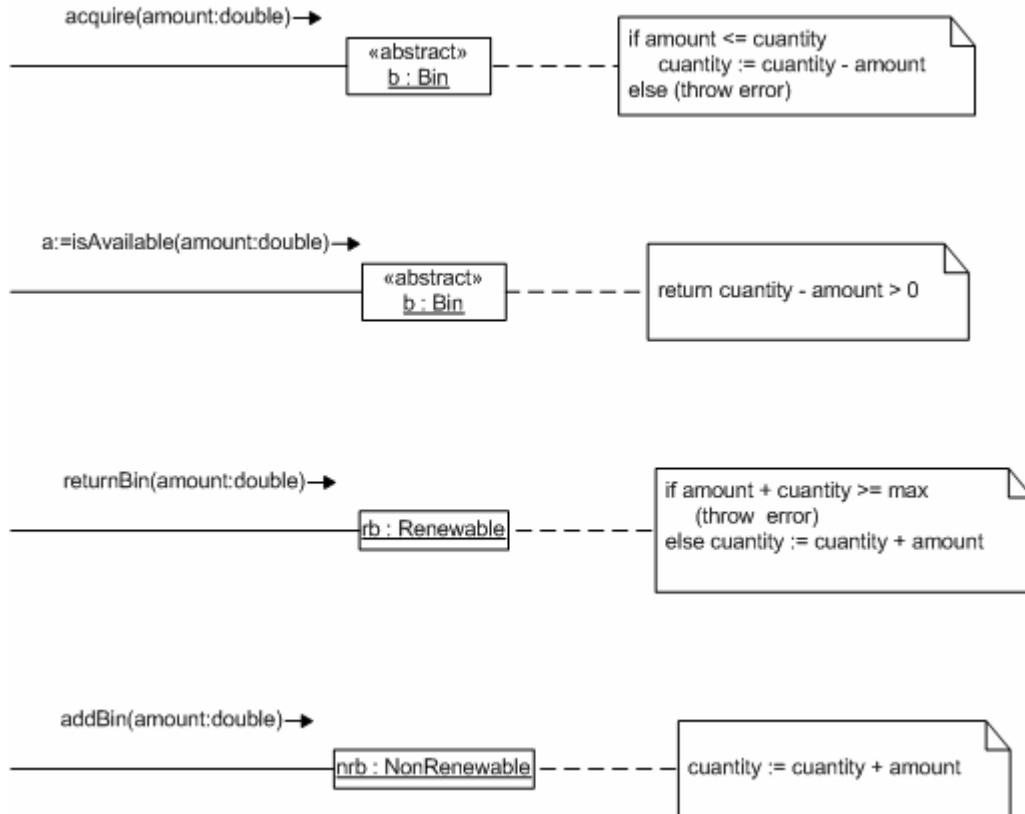


Figure 2-9: Bins Collaboration Diagram

2.10.1 Description

Here we describe all bins collaboration. As you can see these collaborations are very simple and they are complete described in the diagram.

2.10.2 Precondition

Preconditions are explained in the notes.

2.11 Distribution Constructor

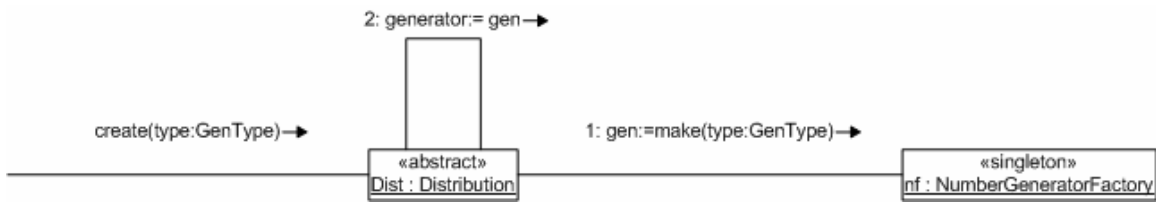


Figure 2-10: Distribution creation Collaboration Diagram

2.11.1 Description

The constructor of a *Distribution* calls the *NumberGeneratorFactory* in order to get a new generator whose type is specified by *type*. When *Dist* is deleted, it must delete the generator too.

2.11.2 Precondition

- *type* has to be a correct label.

Chapter 3 - 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
- [5] EOSimulator User Manual
- [6] Standard Template Library Programmer's Guide <http://www.sgi.com/tech/stl/>