Web-Based Simulation with Sim4edu Omega-Epsilon

How to create and run a web-based simulation with the Sim4edu Omega-Epsilon Simulator [http://sim4edu.com/]

Gerd Wagner <G.Wagner@b-tu.de>

Thanks to Luis Gustavo Nardin for his valuable feedback that helped to improve this tutorial.

Published 2017-06-24

Abstract

This article shows how to create, and run, a discrete event simulation model with the JavaScript-based Sim4edu Omega-Epsilon Simulator [http://sim4edu.com/], which implements the *Object-Event* Simulation Language OESL [http://sim4edu.com/OES], representing a general *Discrete Event* simulation approach based on the object-event worldview. In OESL, a model normally defines various types of objects and events, but OESL also supports models without events, using pure *fixed-increment time progression* corresponding to implicit time events ("ticks"), which is a popular approach in social science simulation.

This tutorial is available in the following formats: HTML [Tutorial.html] PDF [IntroTutorial.pdf]

Table of Contents

1. Introduction to Object-Event Modeling and Simulation

Simulation is used widely today: in many scientific disciplines for investigating specific research questions, in engineering for testing the performance of designs, in education for providing interactive learning experiences, and in entertainment for making games.

For simulating a dynamic system one can model it in terms of

1. the types of objects it is composed of,

2. the types of events that are responsible for its dynamics,

3. the discrete state changes of objects caused by the occurrence of an event of some type,

4. the follow-up events caused by the occurrence of an event of some type,

5. the continuous state changes of objects (described with the help of mathematical functions).

Many dynamic systems are examples of **discrete event systems** (or *discrete dynamic systems*), which consist of:

- **objects** (of various types) whose states may be changed by
- **events** (of various types) occurring at some point in time.

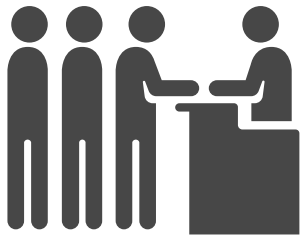
This means that in order to model a discrete event system, we have to

1. describe its **object types** and **event types** (in an information model);

2. specify, for any event type, the **state changes** of objects and the **follow-up events** caused by the occurrence of an event of that type (in a process model).

1.1. Making a Conceptual Model

Let's look at an example. We model a system of one or more *service desks*, each of them having its own queue, as a discrete event system:



- Customers arrive at a service desk at random times.
- If there is no other customer in front of them, and the service desk is available, they are served immediately, otherwise they have to queue up in a waiting line.
- The duration of services varies, depending on the individual case.
- When a service is completed, the customer departs and the next customer is served, if there is still any customer in the queue.

The potentially relevant **object types** of the problem domain are:

- *Customer*,
- *ServiceDesk*,
- *WaitingLine*,

- *ServiceClerk*, if the service is performed by (one or more) clerks.

The potentially relevant **event types** are:

- *CustomerArrival*,
- *StartOfService*,
- *EndOfService*,
- *CustomerDeparture*.

1.2. Making an Information Design Model

When making a simulation model, the right degree of abstraction depends on the purpose of the model. But abstracting away from too many things may make a model too unnatural and not sufficiently generic, implying that it cannot be easily extended to model additional features (such as more than one service desk).

In the case of our example, if the purpose of the simulation model is to compute the service utilization and the maximum queue length, only, then we may abstract away from the following object types:

- *Customer*: we don't need any information about individual customers.
- *WaitingLine*: we don't need to know who is next, it's sufficient to know the length of the queue.
- *ServiceClerk*: we don't need any information about the service clerk(s).

Notice that, for simplicity, we consider the customer that is currently being served to be part of the queue. In this way, in the simulation program, we can check if the service desk is busy by testing if the length of the queue is greater than 0. Consequently, we can model the system state in terms of (one or more) *ServiceDesk* objects having only one property: *queueLength* (a non-negative integer).

We also look for opportunities to simplify our event model by dropping event types that are not needed, e.g., because their events temporally coincide with events of another type. This is the case with *EndOfService* and *CustomerDeparture* events. Consequently, we can drop the event type *EndOfService*.

There are two situations when a new service can be started: either when the waiting line is empty and a new customer arrives, or when the waiting line is not empty and a service ends. Therefore, any *StartOfService* event immediately follows either a *CustomerArrival* or a *CustomerDeparture* event, and we may abstract away from the *StartOfService* event and drop it from the model.

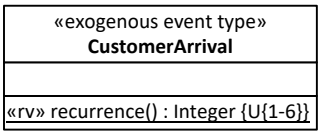
So we only need to model two event types: *CustomerArrival* and *CustomerDeparture*.

The event type *CustomerArrival* is an example of a type of **exogenous** events, which are not caused by any causal regularity of the system under investigation and, therefore, have to be modeled with a **recurrence** function that allows to compute the time of the next occurrence of an event of that type. In OES, exogenous event types are a built-in concept such that an OES simulator takes care of creating the next exogenous event whenever an event of that type is processed. This mechanism makes sure that there is a continuous stream of exogenous events throughout a simulation run.

We also have to model the random variations of two variables: (1) the recurrence of (that is, the time in-between two) customer arrival events and (2) the service duration. In a class model, such random

variables can be defined as special class-level ("static") methods, with stereotype «rv», in the class to which they belong.

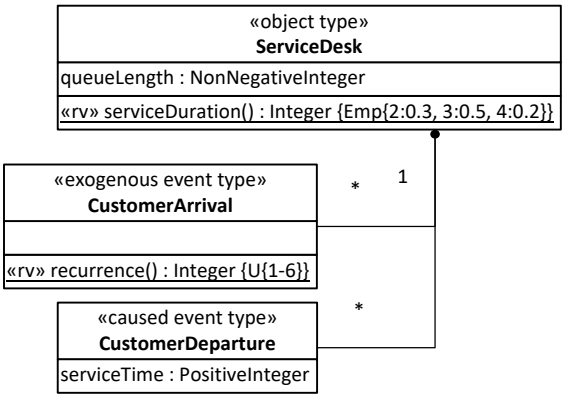
We model the recurrence of customer arrival events as a discrete random variable with a uniform distribution between 1 and 6 (minutes), which we express in the class diagram of the information design model by appending the symbolic expression *U{1-6}* within curly braces to the method declaration (following the UML syntax for property/method modifiers), as shown in the following diagram:



We model the service duration random variable with an empirical distribution of 2 (minutes) with probability 0.3, 3 (minutes) with probability 0.5 and 4 (minutes) with probability 0.2, using the symbolic expression *Emp{2:0.3, 3:0.5, 4:0.2}*. We

Computationally, object types and event types correspond to classes, either of an object-oriented information model, such as a UML class diagram, or of a computer program written in an object-oriented programming language, such as Java or JavaScript. In our example, the class *ServiceDesk* has a property *queueLength*, and the classes *CustomerArrival* and *CustomerDeparture* have a property *serviceDesk* referencing the service desk at which an event occurs.

Thus, we get the following UML class diagram visualizing the information design model, which is part of our simulation design model:



Notice that both event types, *CustomerArrival* and *CustomerDeparture*, have a many-to-one association with the object type *ServiceDesk*. This expresses the fact that any such event occurs at a service desk, which participates in the event. This association is implemented in the form of a reference property *serviceDesk* in each of the two event types.

In addition to this information model, we need to make a process model, which captures the dynamics of the service desk system consisting of arrival and departure events triggering state changes and follow-up events.

1.3. Making a Process Design Model

A process model can be expressed with the help of event rules, which define what happens when an event (of a certain type) occurs, or, more specifically, which state changes and which follow-up events are caused by an event of that type.

Event rules can be expressed with the help of pseudo-code or in process diagrams, or in a simulation or programming language. The following table shows the two event rules defining the transition logic of a service desk system, expressed in pseudo-code.

ON (event type)	**FOR (variable)**	**DO (event routine)**

CustomerArrival(sd) @ t
 	
with sd : ServiceDesk
	sTime: Integer	`INCREMENT sd.queueLength`
		`IF sd.queueLength = 1`
		`THEN`
		`  sTime = ServiceDesk.serviceDurat`
		`  SCHEDULE CustomerDeparture( sTi`

CustomerDeparture(sd) @ t
 	
with sd : ServiceDesk
	sTime: Integer	`DECREMENT sd.queueLength`
		`IF sd.queueLength > 0`
		`THEN`
		`  sTime = ServiceDesk.serviceDurat`
		`  SCHEDULE CustomerDeparture( sTi`

In the next section, we discuss how to implement this simple model of a service desk system with the Sim4edu Omega-Epsilon simulation framework.

2. Making Simulations with Sim4edu Omega-Epsilon

The Simulation for Education (Sim4edu) [http://sim4edu.com/] project website supports web-based simulation with open source technologies for science and education. It provides technologies, such as simulators and simulation programming libraries, and simulation examples. One important goal of Sim4edu is to facilitate building state-of-the-art user interfaces for simulations and simulation games without requiring simulation developers to learn all the recent web technologies involved (e.g., HTML5, CSS3, SVG and WebGL).

The Sim4edu *Omega-Epsilon* (ΩE) simulator implements the *Object-Event* Simulation Language OESL [http://sim4edu.com/OES], which represents a general *Discrete Event* simulation approach based on the object-event worldview. In OESL, a model normally defines various types of objects and events, but OESL also supports

1. models without objects, if they define state variables in the form of global variables, instead;
2. models without events, if they use pure fixed-increment time progression (by defining an `OnEachTimeStep` procedure and a `timeIncrement` parameter), instead; such a model can be used
   a. as a discrete model that abstracts away from explicit events and uses only implicit time events ("ticks"), which is a popular approach in social science simulation, or
   b. for modeling continuous state changes (e.g. objects moving in a continuous space).

Using a simulation framework like Sim4edu ΩE means that the model-specific logic has to be coded in the form of object types, event types, event routines and other functions for model-specific computations, but not the general simulator operations (e.g. time progression and statistics) and the environment handling (e.g. user interfaces for statistics output and visualization).

The following sections discuss the basic concepts of OESL and the Sim4edu ΩE simulator, and show how to implement the simple service desk model described in the previous section. You can run the

simulation [http://simurena.com/sims/1] and download the code [http://sim4edu.com/downloads] from the Sim4edu website.

2.1. Simulation Time

A simulation model has an underlying **time model**, which can be either the abstract model of discrete time, when setting

```
sim.model.time = "discrete";
```

or the concrete model of continuous (calendaric) time, when setting

```
sim.model.time = "continuous";
```

Choosing a discrete time model means that time is measured in steps (with equal durations), and all random time variables used in the model need to be discrete (i.e., based on discrete probability distributions). Choosing a continuous time model means that one has to define a *simulation time granularity*, as explained in the next sub-section.

In both cases, the underlying simulation **time unit** can be either left unspecified (in the case of an abstract time model), or it can be set to one of the time units "ms", "s", "m", "h", "D", "W", "M" or "Y", as in

```
sim.model.timeUnit = "h";
```

2.1.1. Time Granularity

When a simulation model is based on continuous time, it is possible to control the time granularity (the time delay until the next moment) in one of two ways:

1. through simulation time rounding by setting the model parameter *timeRoundingDecimalPlaces* to a suitable value, which implies a corresponding value of the model parameter *nextMomentDeltaT*;

2. by explicitly setting the model parameter *nextMomentDeltaT*.

The model parameter *nextMomentDeltaT* is used by the simulator for scheduling next events with a minimal delay.

2.1.2. Time Progression

An important issue in simulation is the question how the simulation time is advanced by the simulator. The OES paradigm supports ***fixed-increment*** *time progression* and ***next-event*** *time progression*, and their combination.

A Sim4edu ΩE model with pure fixed-increment time progression defines an `OnEachTimeStep` procedure and a `timeIncrement` parameter, but no event types. Such a model can be used

1. for modeling continuous state changes (e.g. objects moving in a continuous space), or

2. as a discrete model that abstracts away from explicit events and uses only implicit periodic time events ("ticks"), which is a popular approach in social science simulation.

A simulation model with pure next-event time progression, representing a classical DES model, defines event types and event rules, but no *timeIncrement* parameter.

It is also possible to combine both time progression mechanisms, e.g., in a "hybrid" model that supports both discrete and continuous state changes, or in a social science model based on "ticks" and explicit events.

2.1.3. Real-Time Simulation

Real-time simulation means to run an observable simulation model in such a way that the speed of its state changes is close to the speed of the state changes in the simulated real-world system. This is only possible if the simulator is able to run the simulation at least as fast as the real-world system is running. If this is the case, the running simulator can be slowed down to real-time speed.

In the case of fixed-increment time progression with a *timeUnit* and real-time simulation turned on (by setting the scenario parameter *realtimeFactor* to 1), the simulator delays each simulation step such that its real duration is equal to its simulation time, which is *timeIncrement* [timeUnit].

In the case of fixed-increment time progression without a *timeUnit* (that is, with abstract time), the simulator cannot automatically run in real-time, but the scenario parameter *stepDuration* (for specifying the real duration of a simulation step) can be set to a suitable value for making the simulation observable in real-time.

2.2. Simulation Models

A *simulation model* essentially defines the state structure and the dynamics of the simulated system. While the system's state structure is defined by the types of objects that populate it, its dynamics is defined by certain types of events and the state changes and follow-up events caused by them. We model the *state structure* of a simulated system with the help of global variables and object types, and we model its *dynamics* with the help of event types and event rules, such that, for any event type, an event rule specifies the *state changes* of affected objects and the *follow-up events* caused by the occurrence of an event of that type.

In the OES approach, a ***simulation model*** essentially consists of:

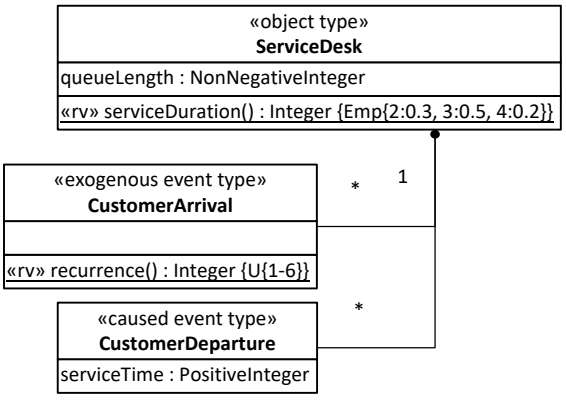
1. a choice of *time model* (either *discrete* or *continuous* time);

2. possibly a choice of *space model*, if the simulation is about objects located in some space;

3. a set of *object type* and *event type* definitions (and possibly other entity types for modeling *activities*, *agents*, *actions*, etc.);

4. a set of *event rules*, which capture causal regularities governing the causation of system state changes and follow-up events.

We now show how to implement the object and event types defined by the following information design model discussed in the previous section:



2.2.1. Object Types

Object types are defined in the form of classes (more precisely, as instances of the meta-class *cLASS*). As an example, we define an object type for service desks with the attribute `queueLength`:

```
var ServiceDesk = new cLASS({
  Name: "ServiceDesk",
  supertypeName: "oBJECT",
  properties: {
    "queueLength": { range: "NonNegativeInteger",
        initialValue: 0, label: "Queue length"}
  }
});
```

Notice that, in Sim4edu ΩE, object types are defined as subtypes of the pre-defined class `oBJECT`, from which they inherit an integer-valued `id` attribute and an optional `name` attribute.

The discrete random variable for modeling random service durations, which samples integers between 2 and 4 from the empirical probability distribution *Emp{2:0.3, 3:0.5, 4:0.2}*, is implemented as a class-level function `serviceDuration` in the *ServiceDesk* class:

```
ServiceDesk.serviceDuration = function () {
  var r = rand.uniformInt( 0, 99);
  if ( r < 30) return 2;          // probability 0.30
  else if ( r < 80) return 3;     // probability 0.50
  else return 4;                  // probability 0.20
};
```

2.2.2. Event Types

We distinguish between two kinds of events:

1. *caused* events are caused by other events occurring during a simulation run;

2. *exogenous* events seem to happen spontaneously, but may be caused by factors, which are external to the simulation model.

Here is an example of an exogenous event type definition:

```
var CustomerArrival = new cLASS({
  Name: "CustomerArrival",
  supertypeName: "eVENT",
  properties: {
    "serviceDesk": {range: "ServiceDesk"}
  },
  methods: {
    "onEvent": function () {
      ...
    }
  }
});
```

Notice that the event type includes a reference property `serviceDesk`, which is used for referencing the service desk object at which an event occurs. Each event type needs to define an *onEvent* method, which implements the event rule for events of the defined type. Event rules are discussed below.

Typically, exogenous events occur periodically. They are therefore defined with a *recurrence* function, which provides the time in-between two events (often in the form of a random variable), and with a *createNextEvent* function, which invokes the *recurrence* function. The *recurrence* function and a *createNextEvent* function (with a parameter *e* for the current event) are defined as class-level methods:

```
CustomerArrival.recurrence = function () {
  return rand.uniformInt( 1, 6);
};
CustomerArrival.createNextEvent = function (e) {
 return new CustomerArrival({
     occTime: e.occTime + CustomerArrival.recurrence(),
     serviceDesk: e.serviceDesk
  });
};
```

Notice that the *recurrence* random variable method of *CustomerArrival* is coded with the library method `rand.uniformInt`, which allows sampling discrete uniform probability distribution functions (the `rand` library provides several other PDF sampling methods as explained below). The *createNextEvent* function is invoked by the simulator for creating, and scheduling, the next event whenever an event of the given exogenous event type occurs.

In our example model of a service desk system, any *customer departure* event is caused, either by a customer arrival event or by a preceding *service start* event.

```
var CustomerDeparture = new cLASS({
  Name: "CustomerDeparture",
  supertypeName: "eVENT",
  properties: {
    "serviceTime": {range: "NonNegativeInteger"},
    "serviceDesk": {range: "ServiceDesk"}
  },
  methods: {
    "onEvent": function () {
      ...
  }
});
```

2.2.3. Event Rules

An event rule for an event type defines what happens when an event of that type occurs, by specifying the caused state changes and follow-up events. In Sim4edu ΩE, an event rule for an event type is defined as a method `onEvent` of the class that implements the event type. This method, which is also called *event routine*, returns a set of events (more precisely, JS objects representing events).

The following event rule method is defined in the `CustomerArrival` class.

```
// CustomerArrival event rule
"onEvent": function () {
  var srvTm=0, changes = [], events = [];
  this.serviceDesk.queueLength++;
  sim.stat.arrivedCustomers++;
  // if the service desk is not busy
  if (this.serviceDesk.queueLength === 1) {
    srvTm = ServiceDesk.serviceDuration();
```

```
    events.push( new CustomerDeparture({
      occTime: this.occTime + srvTm,
      serviceTime: srvTm,
      serviceDesk: this.serviceDesk
    }));
  }
  return events;
}
```

The context of this event rule method is the event that triggers the rule, that is, the variable `this` references a JS object that represents the triggering event. Thus, the expression `this.serviceDesk` refers to the service desk object associated with the current customer arrival event, and the statement `this.serviceDesk.queueLength++` increments the *queueLength* attribute of this service desk object (as an immediate state change).

The following event rule method is defined in the `CustomerDeparture` class.

```
// CustomerDeparture event rule
"onEvent": function () {
  var changes = [], events = [], srvTm=0;
  // remove customer from queue
  this.serviceDesk.queueLength--;
  // if there are still customers waiting
  if (this.serviceDesk.queueLength > 0) {
    // start next service and schedule its end/departure
    srvTm = ServiceDesk.serviceDuration();
    events.push( new CustomerDeparture({
      occTime: this.occTime + srvTm,
      serviceTime: srvTm,
      serviceDesk: this.serviceDesk
    }));
  }
  sim.stat.departedCustomers++;
  sim.stat.totalServiceTime += this.serviceTime;
  return events;
}
```

2.2.4. Library Methods for Sampling Probability Distribution Functions

Random variables are implemented as methods that sample specific probability distribution functions (PDFs). For facilitating the definition of specific PDF sampling methods, simulation frameworks typically provide a library of predefined parametrized PDF sampling methods, which can be used with one or several (possibly seeded) streams of random numbers (also called *pseudo-random numbers*).

The ΩE simulator provides the following predefined parametrized PDF sampling methods:

Probability Distribution Function	**ΩE Library Method**	**Example**
Uniform [https://en.wikipedia.org/wiki/Uniform_distribution_(continuous)]	`uniform(` *lowerBound*, *upperBound*`)`	`rand.uniform(0.5, 1.5)`

Probability Distribution Function	ΩE Library Method	Example
Discrete Uniform [https://en.wikipedia.org/wiki/Discrete_uniform_distribution]	`uniformInt(` *lowerBound*, *upperBound*)	`rand.uniformInt(1, 6)`
Triangular [https://en.wikipedia.org/wiki/Triangular_distribution]	`triangular(` *lowerBound*, *upperBound*, *mode*)	`rand.triangular(0.5, 1.5, 1.0)`
Exponential [http://en.wikipedia.org/wiki/Exponential_distribution]	`exponential(` *eventRate*)	`rand.exponential( 0.5)`
Gamma [https://en.wikipedia.org/wiki/Gamma_distribution]	`gamma(` *shape*, *scale*)	`rand.gamma( 1.0, 2.0)`
Normal [https://en.wikipedia.org/wiki/Normal_distribution]	`normal(` *mean*, *stdDev*)	`rand.normal( 1.5, 0.5)`
Pareto [https://en.wikipedia.org/wiki/Pareto_distribution]	`pareto(` *shape*)	`rand.pareto( 2.0)`
Weibull [https://en.wikipedia.org/wiki/Weibull_distribution]	`weibull(` *scale*, *shape*)	`rand.weibull( 1, 0.5)`

The ΩE `rand.js` library supports both unseeded and seeded random number streams. By default, the `rand.`*distr* methods are based on an unseeded stream using Marsaglia's high-performance random number generator *xorshift* that is built into the `Math.random` function of modern JavaScript engines.

A seeded random number stream, based on the slower Mersenne Twister algorithm, can be obtained by setting the scenario parameter `sim.scenario.randomSeed` to a positive integer value.

Additional streams can be defined and used in the following way:

```
var stream1 = new Random ( 1234);
var stream2 = new Random ( 6789);
var service1Duration = stream1.exponential( 0.5);
var service2Duration = stream2.exponential( 1.5);
```

2.3. Simulation Scenarios

For obtaining a complete executable *simulation scenario*, a simulation model has to be complemented with simulation parameter settings and an initial system state.

In general, we may have more than one simulation scenario for a simulation model. For instance, the same model could be used in two different scenarios with different initial states, or with different visualizations.

A ***simulation scenario*** consists of

1. simulation parameter settings, such as setting a value for `simulationEndTime`,

2. a *simulation model*,

3. an *initial state* definition, and

4. optional *user interface (UI)* definitions of, e.g., a *statistics* UI and an *observation* (or *visualization*) UI.

An empty template for a simulation scenario has the following structure:

```
// ***** Simulation Parameters **************
sim.scenario.simulationEndTime = ...;
sim.scenario.randomSeed = ...;  // optional
sim.scenario.createLog = ...;  // true/false
sim.scenario.visualize = ...;  // true/false
// ***** Simulation Model *******************
sim.model.name = "...";
sim.model.time = "...";  // discrete or continuous
sim.model.objectTypes = [...];
sim.model.eventTypes = [...];
// ***** Initial State **********************
sim.scenario.initialState.objects = {...};
sim.scenario.initialState.events = {...};
// Ex-Post Statistics
sim.model.statistics = {...};
```

We briefly discuss each group of scenario information items in the following sub-sections.

2.3.1. Simulation Parameters

Simulation parameters are defined as attributes of the simulation scenario. The most important parameters are:

- **simulationEndTime** - this mandatory attribute defines the duration of a simulation run;

- *stepDuration* - an optional attribute for specifying a minimum execution-time duration (in milliseconds) for each simulation step. This can be used for delaying simulation steps such that the simulation runs in real-time.

- *randomSeed*: Setting this optional parameter to a positive integer allows to obtain a specific fixed random number sequence generated by the random number generator. This can be useful for being able to test a simulation model by testing if expected results are obtained.

- *visualize*: A Boolean parameter that allows to turn on/off any visualization defined by the scenario.

- *createLog*: A Boolean parameter that allows to turn on/off the simulation log.

2.3.2. Documentation

Both the model and the scenario can be documented by providing a *name*, a *title* and a *shortDescription*, as well as meta data like *creator*, a *created* date, a (last) *modified* date and a copyright *license*, like so

```
sim.model.title = "...";
sim.model.shortDescription = "...";
sim.model.license = "CC BY-NC";
```

It is recommended to use an *attribution share-alike* Creative Commons [http://creativecommons.org/] license by specifying its abbreviated name "CC BY-SA" (or "CC BY-NC" for non-commercial use).

The mandatory model attribute *systemNarrative* has to be used for providing a brief description of the system under investigation, as opposed to the design-specific model description provided by *shortDescription*:

```
sim.model.systemNarrative = "...";
```

2.3.3. Initial State

Defining an initial state means:

1. assigning initial values to global variables, if there are any;

2. defining which objects exist initially, and assigning initial values to their properties;

3. defining which events are scheduled initially.

A scenario must include an initial state definition, which consists of a set of object definitions and a set of initial event definitions. An initial state object is defined as an entry in the map `initialState.objects` such that the object's `id` value is the map entry's key, and the map entry's value is a set of property-value slots, including a slot for the special attribute `typeName` defining the object's type, as shown in the following example:

```
sim.scenario.initialState.objects = {
  "1": {typeName: "ServiceDesk", name:"serviceDesk1", queueLength: 0}
};
```

Notice that object IDs are positive integers, but when used as keys in a map, they are converted to strings.

An initial event is defined as an element of the array list `initialState.events` in the form of a set of property-value slots, including a slot for the special attribute `typeName` defining the event's type, as shown in the following example:

```
sim.scenario.initialState.events = [
  {typeName: "CustomerArrival", occTime:1, serviceDesk:"1"}
];
```

2.4. Statistics

In scientific and engineering simulation projects the main goal is getting estimates of the values of certain variables or performance indicators with the help of statistical methods. In educational simulations, statistics can be used for observing simulation runs and for learning the dynamics of a simulation model.

For collecting statistics, suitable *statistic variables* have to be defined. The following code defines statistics variables for the service desk model.

```
sim.model.statistics = {
  "arrivedCustomers": {range:"NonNegativeInteger", label:"Arrived customers"},
  "departedCustomers": {range:"NonNegativeInteger", label:"Departed customers"},
  "totalServiceTime": {range:"NonNegativeInteger"},
  "serviceUtilization": {range:"Decimal", label:"Service utilization",
      computeOnlyAtEnd: true, decimalPlaces: 1, unit: "%",
      expression: function () {
```

```
        return sim.stat.totalServiceTime / sim.time * 100
      }
  },
  "maxQueueLength": {objectType:"ServiceDesk", objectIdRef: 1,
      property:"queueLength", aggregationFunction:"max", label:"Max. queue leng
  "averageQueueLength": {objectType:"ServiceDesk", objectIdRef: 1,
    property:"queueLength", aggregationFunction:"avg", label:"Avg. queue length
  "queueLength": {objectType:"ServiceDesk", objectIdRef: 1,
    property:"queueLength", showTimeSeries: true, label:"Queue length"}
};
```

The first three statistics variables (*arrivedCustomers*, *departedCustomers* and *totalServiceTime*) are simple variables that are updated in event rule methods.

The *serviceUtilization* variable is only computed at the end of a simulation run by evaluating the expression specified for it (dividing the total service time by the simulation time). In the case of the remaining three variables, the data source is the object property `queueLength` of the service desk object with id=1. For the variable `maxQueueLength` the built-in aggregation function `max` is applied to this data source, computing the maximum of all *queueLength* values, while for the variable `averageQueueLength` the aggregation function `avg` is applied. The last variable, *queueLength*, is defined for the purpose of getting a time series chart.

The statistics results are shown in a default view of the statistics output. It is an option to define a non-standard user interface for the statistics output.

2.5. Accessing Objects

The objects defined in the initial state, or created during a simulation run, can be accessed either by their ID number or by their name, if they have a name. For instance, the object {typeName:"ServiceDesk", id: 1, name:"serviceDesk1", queueLength: 0} defined above, has the ID number 1 and the name "serviceDesk1". It can be retrieved from the simulator map `sim.objects` in the following way:

```
var object1 = sim.objects["1"];
```

It can also be retrieved by name from the simulator map `sim.namedObjects` in the following way:

```
var object1 = sim.namedObjects["serviceDesk1"];
```

For looping over all simulation objects, we can loop over the simulator map `sim.objects` in the following way:

```
Object.keys( sim.objects).forEach( function (objId) {
  var obj = sim.objects[objId];
  ...  // do something with obj
});
```

We can loop over all simulation objects of a specific type, say `ServiceDesk`, in the following way:

```
Object.keys( cLASS["ServiceDesk"].instances).forEach( function (objId) {
  var obj = cLASS["ServiceDesk"].instances[objId];
  ...  // do something with obj
});
```

If a simulation has to deal with a large number of objects, using a `for` loop may be faster than a `forEach` loop.

2.6. Animation

Animation is important for educational simulations and games, but it can also be used as a general tool for testing, inspecting and validating simulations.

Simulations can be animated by visualizing objects and their state, by sonifying events and by allowing human users to interact with the simulated world. Accordingly, the simulation language OESL allows to add the following user interfaces (UI) to a simulation model:

1. An **observation UI** defines various kinds of visualizations (including 3D) for allowing the user to observe what is going on during a simulation run. Space models and objects are visualized by defining a *view* for them. A view is defined by a 2D shape (like a *rectangle* or a *polygon*) or a 3D shape (like a *cuboid* or a *mesh*).

2. A **sonification UI** allows attaching specific sounds and melodies to event occurrences by defining *event appearances* for event types.

3. A **participation UI** allows human users to interact with a simulated world by performing in-world actions via the user interface. Any simulation model can be turned into a user-interactive simulation by adding a participation model and a corresponding UI.

In the next version of this tutorial, we will show how an observation UI can be added to a simulation model as an incremental extension that does not affect the simulation model.