Web-Based Simulation with OESjs

How to create and run simulations with the JavaScript-based simulation framework OESjs available on sim4edu.com

Gerd Wagner G.Wagner@b-tu.de

Thanks to Luis Gustavo Nardin and Jakub Lelo for valuable feedback that helped to improve this tutorial.

Published 2019-04-11

Abstract

This article shows how to create, and run, a *simulation model* with the JavaScript-based simulation framework OESjs available on Sim4edu.com. OESjs implements the *Object Event Simulation (OES)* paradigm, representing a general *Discrete Event Simulation* approach based on object-oriented modeling and event scheduling. In OES, a model normally defines various types of objects and events, but OES also supports models without explicit events, using 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 PDF

Table of Contents

List of Figures

List of Tables

Chapter 1. Introduction to Object Event Modeling

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 differential equations).

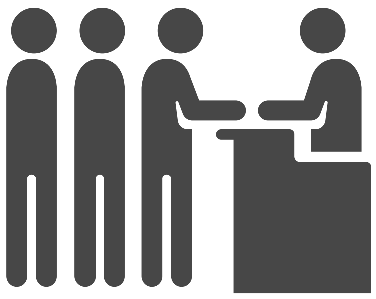
Many dynamic systems are examples of **discrete event systems** (or discrete dynamic systems), which consist of:

- **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:

- customers,
- service desks,
- waiting lines,
- service clerks, if the service is performed by (one or more) clerks.

The potentially relevant **event types** are:

- customer arrivals,
- service starts,
- service terminations,
- customer departures.

1.2. Making a Simulation 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 our example, the purpose of the simulation model is to compute the *maximum queue length* and possibly also the *service utilization*. So, we may abstract away from the following object types:

- *customers*: we don't need any information about individual customers.
- *waiting lines*: we don't need to know who is next, it's sufficient to know the length of the queue.
- *service clerks*: 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. In fact, for being able to compute the service utilization and the maximum queue length, the queue length is the only relevant state variable.

State variables can be modeled in the simple form of *global variables* or in the form of *attributes* of suitable object types. Consequently, the simplest model we can make for the given problem, called *ServiceDesk-0*, has only one global variable: *queueLength*. But, as an alternative, more explicit, model, called *ServiceDesk-1*, we will also model the system state in terms of (one or more) *ServiceDesk* objects having only one property:

queueLength. As opposed to the simpler model defining *queueLength* as a global variable, this model allows defining simulation scenarios with two or more service desks operating simultaneously.

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 *service terminations* and *customer departure* events. Consequently, we can drop the event type *service terminations*.

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 terminates. Therefore, any *service start* event immediately follows either a *customer arrival* or a *customer departure* event, and we may abstract away from *service start* events and drop the corresponding event type from the design model.

So we only need to consider *customer arrival* and *customer departure* events, modeled with the 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") operations, with a stereotype «rv», in the class to which they belong, as shown in the diagrams below.

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 operation declaration, following the UML syntax for property/method modifiers.

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}*.

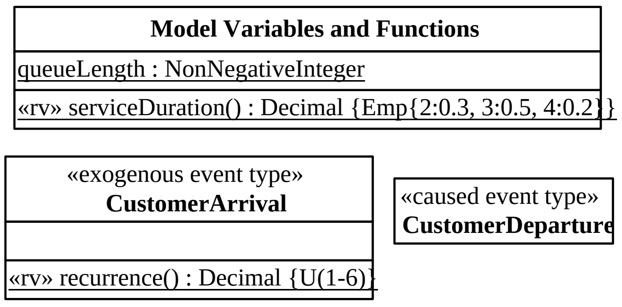
Computationally, object types and event types correspond to classes, either of an object-oriented information model, such as a UML class diagram, or of a computer program written in an object-oriented programming language, such as Java or JavaScript.

1.2.1. ServiceDesk-0: Modeling *queueLength* as a global variable

As discussed above, the simplest model for the service desk problem with maximum queue length statistics (available in the Sim4edu library as ServiceDesk-0) has only one global variable: *queueLength*, a non-negative integer, and a global function for computing the random service duration, but no object type.

An information model for ServiceDesk-0 consists of a special class for defining model variables and functions, and two classes for defining the event types *CustomerArrival* and *CustomerDeparture*, as shown in **Figure 1-1.** An information design model for ServiceDesk-0.

Figure 1-1. *An information design model for ServiceDesk-0*


Model Variables and Functions
queueLength : NonNegativeInteger
«rv» serviceDuration() : Decimal {Emp{2:0.3, 3:0.5, 4:0.2}}
«exogenous event type»
CustomerArrival
«rv» recurrence() : Decimal {U(1-6)}
«caused event type»
CustomerDeparture

In addition to an information model, which captures the system's state structure, we also need to make a process model that captures the dynamics of the service desk system. The dynamics of a system consists of events triggering state changes and follow-up events. 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)	DO (event routine)

CustomerArrival @ t	`INCREMENT queueLength`
	`IF queueLength = 1 THEN`
	`  sTime := serviceDuration()`
	`  SCHEDULE CustomerDeparture @ (t + sTime)`

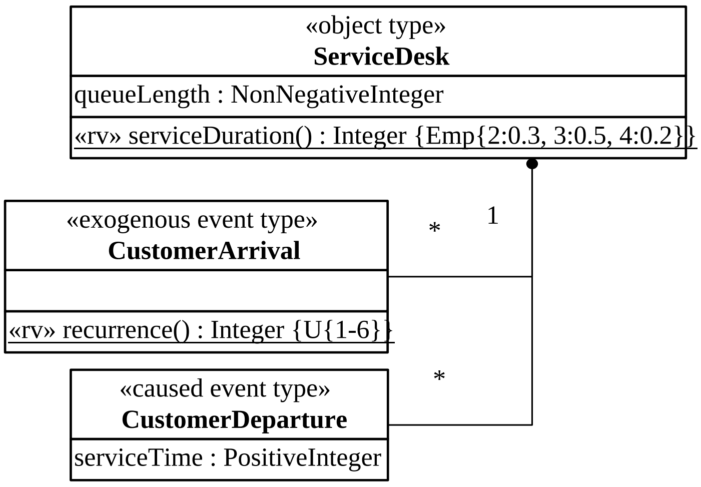
CustomerDeparture @ t	`DECREMENT queueLength`
	`IF queueLength > 0 THEN`
	`  sTime := serviceDuration()`
	`  SCHEDULE CustomerDeparture @ (t + sTime)`

1.2.2. ServiceDesk-1: Modeling *queueLength* as an attribute

In our extended model (ServiceDesk-1) we represent the state variable *queueLength* as an attribute of an object type *ServiceDesk*. This results in a model with three classes, the object class *ServiceDesk* with an attribute *queueLength*, and the event classes *CustomerArrival* and *CustomerDeparture*, both with a reference property *serviceDesk* for referencing the service desk at which an event occurs. When we also want to compute the service utilization statistics, we need to add an attribute *serviceTime* to the *CustomerDeparture* class for being able to update the service utilization statistics when a customer departs.

Both event types, *CustomerArrival* and *CustomerDeparture*, now have a many-to-one association with the object type *ServiceDesk*. This expresses the fact that any such event occurs at a particular 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, as shown in **Figure 1-2.** An information design model for ServiceDesk-1.

Figure 1-2. *An information design model for ServiceDesk-1*


«object type»
ServiceDesk
queueLength : NonNegativeInteger
«rv» serviceDuration() : Integer {Emp{2:0.3, 3:0.5, 4:0.2}}
«exogenous event type»
CustomerArrival
«rv» recurrence() : Integer {U{1-6}}
«caused event type»
CustomerDeparture
serviceTime : PositiveInteger
*
1
*

In addition to an 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.

The following table shows the two event rules, which now account for the fact that both types of events occur at a particular service desk that is referenced by the event expression parameter *sd*.

ON (event type)	DO (event routine)

CustomerArrival(sd) @ t
 	
with sd : ServiceDesk
	`INCREMENT sd.queueLength`
	`IF sd.queueLength = 1 THEN`
	`  sTime := ServiceDesk.serviceDuration()`
	`  SCHEDULE CustomerDeparture( sTime, sd) @(t + sTime)`

CustomerDeparture(sd) @ t
 	
with sd : ServiceDesk
	`DECREMENT sd.queueLength`
	`IF sd.queueLength > 0 THEN`
	`  sTime := ServiceDesk.serviceDuration()`
	`  SCHEDULE CustomerDeparture( sTime, sd) @(t + sTime)`

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

Chapter 2. Creating Object Event Simulations with OESjs

The Simulation for Education (Sim4edu) project website supports web-based simulation with open source technologies for science and education. It provides technologies, such as simulation libraries, frameworks, and simulators, as well as a collection of 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 JavaScript-based simulation framework *OESjs* implements the *Object Event Simulation (OES)* paradigm, representing a general *Discrete Event Simulation* approach based on *object-oriented* modeling and *event scheduling*. In OES, a model normally defines various types of objects and events, but OES also supports

1. models without objects, if they define state variables in the form of global *model 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).

OESjs supports two forms of simulations:

1. Standalone scenario simulations, which are good for getting a quick impression of a simulation model, e.g., by checking some simple statistics or by observing visualized (or sonified) simulation runs.
2. Simulation experiments, which are defined as a set of simulation scenarios by defining value sets for certain model variables, such that an experiment run consists of a set of scenario runs.

Using a simulation framework like OESjs means that only 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 present the basic concepts of the OESjs simulation framework, and show how to implement the service desk models described in the previous section.

2.1. Simulation Time

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

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

or *continuous time*, when setting

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

Choosing a discrete time model means that time is measured in steps (with equal durations), and all temporal random 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 (e.g., 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";
```

Typical examples of time models are:

1. An abstract discrete model of time where time runs in steps without any concrete meaning:

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

2. A concrete discrete model of time in number of days:

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

3. A concrete continuous model of time in number of seconds:

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

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 (e.g., to 2 for obtaining time values like 281.39), 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.

An OESjs 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.

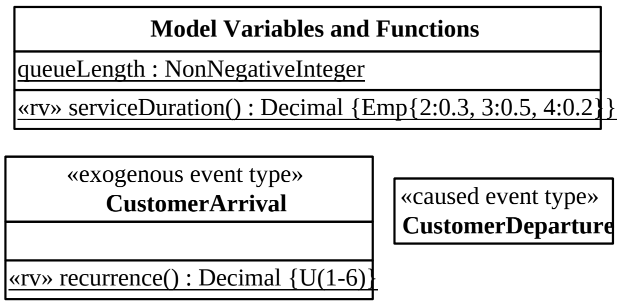
Real-time simulation requires fixed-increment time progression by setting the model parameter *timeIncrement*. In the case of a model with a *timeUnit* and real-time simulation turned on (by setting the scenario parameter *realtimeFactor* to 1), the simulator delays each simulation step such that its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

In the case of a model 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

2.2.1. Model Variables and Functions

In the simple model of a service desk discussed in the previous section, we define one (global) model variable, *queueLength*, one model function, *serviceDuration*(), and two event types, as shown in the following class diagram:


Model Variables and Functions
queueLength : NonNegativeInteger
«rv» serviceDuration() : Decimal {Emp{2:0.3, 3:0.5, 4:0.2}}
«exogenous event type»
CustomerArrival
«rv» recurrence() : Decimal {U(1-6)}
«caused event type»
CustomerDeparture

Notice that this model does not define any object type, which implies that the system state does not consist of any object, but only of one model variable, *queueLength*. The discrete random variable for modeling random service durations is implemented as a model function `serviceDuration` shown in the *Model Variables and Functions* class. It samples integers between 2 and 4 from the empirical probability distribution *Emp{2:0.3, 3:0.5, 4:0.2}*. The model can be coded with OESjs in the following way:

```
// (global) model variable
sim.model.v.queueLength = {
  range:"NonNegativeInteger",
  label:"Queue length",
  shortLabel:"qLen",
  initialValue: 0
};
// (global) model function
sim.model.f.serviceDuration = function () {
  return rand.frequency({"2":0.3, "3":0.5, "4":0.2});
};
```

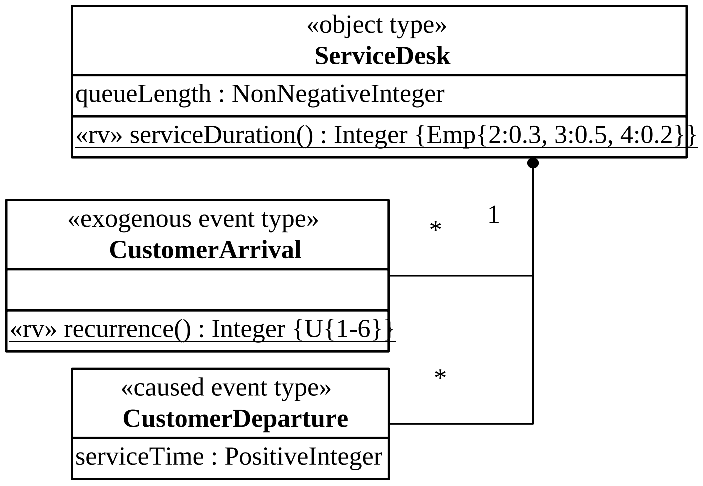
When a model variable, like `sim.model.v.queueLength`, is defined, its value can be accessed at simulation runtime with the expression `sim.v.queueLength`. A variable is shown in the user interface for model variables, whenever a `label` is defined for it. By default, the size of the variable's input field is 7. It can be changed by setting the property `inputFieldSize` in the variable definition.

A variable's value is shown in the simulation log, whenever a `shortLabel` is defined for it.

You can run this simulation model and download its code from the sim4edu.com website.

2.2.2. Object Types

Object types are defined in the form of classes. More precisely, they are defined as instances of the meta-class *cLASS*. Consider the object type *ServiceDesk* defined in the following model:


«object type»
ServiceDesk
queueLength : NonNegativeInteger
«rv» serviceDuration() : Integer {Emp{2:0.3, 3:0.5, 4:0.2}}
«exogenous event type»
CustomerArrival
«rv» recurrence() : Integer {U{1-6}}
«caused event type»
CustomerDeparture
serviceTime : PositiveInteger
*
1
*

The object type *ServiceDesk* is defined with an attribute `queueLength`:

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

Notice that, in OESjs, 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. A property may have both a `label` and a `shortLabel`. The `label` is used for user interface fields, while the `shortLabel` is used in the simulation log, which only logs those objects and properties that do have a *shortLabel*.

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 ("static") function `serviceDuration` in the *ServiceDesk* class:

```
ServiceDesk.serviceDuration = function () {
  return rand.frequency({"2":0.3, "3":0.5, "4":0.2});
};
```

You can run this simulation model and download its code from the sim4edu.com website.

2.2.3. 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",
  label: "Customer arrivals",
  shortLabel: "Arr",  // for the log
  supertypeName: "eVENT",
  properties: {
    "serviceDesk": {range: "ServiceDesk", label:"Service desk"}
  },
  methods: {
    "onEvent": function () {
      ...
    }
  }
});
```

Notice that this event type definition includes a reference property *serviceDesk*, which is used for referencing the service desk object at which an event occurs. In OESjs, event types are defined as subtypes of the pre-defined class `eVENT`, from which they inherit an attribute `occTime`, which holds the occurrence time of an event. As opposed to objects, events do normally not have an ID, nor a name.

Each event type needs to define an *onEvent* method that implements the event rule for events of the defined type. Event rules are discussed below. 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). The recurrence function is defined as a class-level method:

```
CustomerArrival.recurrence = function () {
  return rand.uniformInt( 1, 6);
};
```

Notice that the *recurrence* 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 OESjs simulator automatically creates the next *CustomerArrival* event by invoking the *recurrence* function for setting its *ocurrenceTime* and by copying all participant references (such as the *serviceDesk* reference). Only if an exogenous event type has additional properties, a *createNextEvent* method has to be defined for assigning all properties and returning the next event of that type. Whenever the simulator finds such a method, it will be invoked for creating corresponding exogenous events.

```
var CustomerDeparture = new cLASS({
  Name: "CustomerDeparture",
  label: "Customer departures",
  shortLabel: "Dep",  // for the log
  supertypeName: "eVENT",
  properties: {
    "serviceTime": {range: "NonNegativeInteger"},
    "serviceDesk": {range: "ServiceDesk", label:"Service desk"}
  },
  methods: {
    "onEvent": function () {
      ...
  }
});
```

2.2.4. 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 OESjs, 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, a set of 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;
}
```

2.2.5. Event Priorities

An OES model may imply the possibility of several events occurring at the same time. Consequently, a simulator (like OESjs) must be able to process simultaneous events. In particular, simulation models based on discrete time may create simulation states where two or more events occur at the same time, but the model's logic requires them to be processed in a certain order. Defining priorities for events of a certain type helps to control the processing order of simultaneous events.

Consider an example model based on discrete time with three exogenous event types *StartOfMonth*, *EachDay* and *EndOfMonth*, where the recurrence of StartOfMonth and EndOfMonth is 21, and the recurrence of EachDay is 1. In this example we want to control that on simulation time 1 + *i* * 21 both a StartOfMonth and an EachDay event occur simultaneously, but StartOfMonth should be processed before EachDay, and on simulation time 21 + *i* * 21 both an EndOfMonth and an EachDay event occur simultaneously, but EndOfMonth should be processed after EachDay. This can be achieved by defining a high priority, say 2, to StartOfMonth, a middle priority, say 1, to StartOfMonth, and a low priority, say 0, to EndOfMonth.

Event priorities are defined as class-level properties of event classes in the event type definition file. Thus, we would define in `StartOfMonth.js`:

```
StartOfMonth.priority = 2;
```

and in `EachDay.js`:

```
EachDay.priority = 1;
```

and finally in `EndOfMonth.js`:

```
EndOfMonth.priority = 0;
```

2.2.6. Library Methods for Sampling Probability Distribution Functions

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

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

Probability Distribution Function	OESjs Library Method	Example
Uniform	`uniform`(lowerBound, upperBound)	`rand.uniform( 0.5, 1.5)`
Discrete Uniform	`uniformInt`(lowerBound, upperBound)	`rand.uniformInt( 1, 6)`
Triangular	`triangular`(lowerBound, upperBound, mode)	`rand.triangular( 0.5, 1.5, 1.0)`
Frequency	`frequency`(frequencyMap)	`rand.frequency({"2":0.4, "3":0.6})`
Exponential	`exponential`(eventRate)	`rand.exponential( 0.5)`
Gamma	`gamma`(shape, scale)	`rand.gamma( 1.0, 2.0)`
Normal	`normal`(mean, stdDev)	`rand.normal( 1.5, 0.5)`
Pareto	`pareto`(shape)	`rand.pareto( 2.0)`
Weibull	`weibull`(scale, shape)	`rand.weibull( 1, 0.5)`

The OESjs library `rand.js` supports both unseeded and seeded random number streams. By default, its PDF sampling 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);
```

Warning: Avoid using JavaScript's built-in `Math.random` in simulation code. Always use `rand.uniform`, or one of the other probability distribution functions from the *`rand`* library described above, for generating random numbers.

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.

A *simulation scenario* consists of

1. simulation parameter settings, such as setting a value for `simulationEndTime` and `randomSeed`,
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
// ***** Simulation Model ******************
sim.model.time = "...";  // discrete or continuous
sim.model.timeIncrement = ...;   // optional
sim.model.timeUnit = "...";  // optional (ms|s|m|h|D|W|M|Y)
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. Pre-Defined Simulation Parameters

A few simulation parameters are predefined as attributes of the simulation scenario. The most important ones 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 slowing down simulation steps such that simulation runs can be observed.
- *randomSeed*: Setting this optional parameter to a positive integer allows to obtain a specific fixed random number sequence (generated by a Mersenne Twister random number generator). This can be used for performing simulation runs with the same (repeated) random number sequence, e.g., for testing a simulation model by checking if expected results are obtained.

2.3.2. Metadata

Both the model and the scenario can be documented and described by providing various metadata in a separate `metadata.js` file: 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.name = "...";
sim.model.title = "...";
sim.model.license = "CC BY-NC";
sim.model.shortDescription = "...";
sim.model.systemNarrative = "...";
```

It is recommended to use an *Attribution Share-Alike* Creative Commons 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 real system under investigation, as opposed to the solution-specific model description provided by *shortDescription*.

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 initial 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}
];
```

When initial events (or objects) are parametrized with the help of model variables, they can be defined by moving the `sim.scenario.initialState.events` definition to the `sim.scenario.setupInitialState` function because this function is executed after the model variables are assigned.

2.4. Simulation Configurations

A simulation scenario can be configured with various types of visualizations and various user interfaces (UI):

1. Turn on/off the simulation log by setting the configuration parameter `sim.config.createLog` to true/false.
2. Suppress or show the initial state UI, which allows to inspect/modify the initial values of model variables and the initial state of objects.
3. Turn on/off visualization, if there is one, by setting the configuration parameter *visualize* to true/false.
4. Turn on/off user interaction, if there is one, by setting the configuration parameter *userInteractive* to true/false. Since user interaction requires visualization, it is also turned off when `sim.config.visualize` is set to false.
5. Slow down a (standalone) scenario simulation run by setting the configuration parameter *stepDuration*, which defines the duration of a simulation step (in ms). This is typically used for being able to observe a simulation run.
6. Credit art work used in a visualization with the parameter *artworkCredits*.

In the simulation definition file, we could have settings like the following:

```
sim.config.createLog = true;
sim.config.suppressInitialStateUI = true;
sim.config.visualize = true;
sim.config.userInteractive = false;
sim.config.stepDuration = 200;   // 200 ms observation time per step
sim.config.artworkCredits = "Weather icons by https://icons8.com";
```

2.5. 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 *statistics 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 length"},
  "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 routines (*onEvent* 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.

2.6. Simulation Experiments

There are different types of simulation experiments. In a *simple experiment*, a simulation scenario is run repeatedly by defining a number of replications (iterations) for being able to compute average statistics.

In a *parameter variation experiment*, several variants of a simulation scenario (called *experiment scenarios*), are defined by defining value sets for certain model variables (the *experiment parameters*), such that a parameter variation experiment run consists of a set of experiment scenario runs, one for each combination of parameter values.

When running an experiment, the resulting statistics data are stored in a database, which allows looking them up later on or exporting them to data analysis tools (such as Microsoft Excel and RStudio)

2.6.1. *Simple Experiments*

A simple experiment is defined with a `sim.experiment` record on top of a scenario by defining (1) the number of *replications* and (2) possibly a list of *seed values*, one for each replication. The following code shows an example of a simple experiment definition:

```
sim.experiment.replications = 5;
sim.experiment.seeds = [1234, 2345, 3456, 4567, 5678];
```

Running this simple experiment means running the underlying scenario 5 times, each time with another random seed, as specified by the list of seeds. The resulting statistics is computed by averaging all statistics variables defined for the given model.

When no seeds are defined, the experiment is run with implicit random seeds using JavaScript's built-in random number generator, which implies that experiment runs are not reproducible.

2.6.2. *Parameter Variation Experiments*

A parameter variation experiment is defined with (1) a number of *replications*, (2) a list of *seed values* (one for each replication), and (3) one or more experiment parameters. The following code shows an example of a parameter variation experiment definition:

```
sim.experiment.replications = 5;
sim.experiment.seeds = [1234, 2345, 3456, 4567, 5678];
sim.experiment.parameterDefs = [
  {name:"arrivalEventRate", values:[0.4, 0.5, 0.6]}
];
```

Notice that this experiment definition defines three experiment scenarios: the 1st one with a value of 0.4 for the model variable *arrivalEventRate*, the 2nd one with a value of 0.5 and the 3rd one with a value of 0.6. Running this parameter variation experiment means running each of the 3 experiment scenarios 5 times (each time with another random seed, as specified by the list of seeds). The resulting statistics, as shown in the following table, is computed by averaging all statistics variables defined for the given model.

Experiment Log				
Experiment scenario	**Parameter values**	**Statistics**		
		Arrived customers	**Departed customers**	**Time in system [min]**
0	0.4	4,032	3,981	90.7
1	0.5	5,041	4,788	358.6
2	0.6	6,072	4,926	1,011.6

An experiment parameter must have the same name as the model variable to which it refers. It defines a set of values for this model variable, either using a `values` field or a combination of a `startValue` and `endValue` field (and `stepSize` for a non-default increment value) as in the following example:

```
sim.experiment.replications = 5;
sim.experiment.seeds = [1234, 2345, 3456, 4567, 5678];
sim.experiment.parameterDefs = [
  {name:"arrivalEventRate", startValue:0.4, endValue:0.9)}
];
```

There are a few further settings for controlling the storage of experiment statistics. An experiment should have an `id` value and a `title`. In addition, it should have a sequence number relative to the simulation scenario for which it is defined. These settings are shown in the following code listing:

```
sim.experiment.id = 1;
sim.experiment.title = "Test";
sim.experiment.experimentNo = 1;  // sequence number relative to simulation scenario
```

2.6.1. *Storage of Experiment Statistics Data*

An experiment's statistics data is stored in a browser-managed database using JavaScript's *IndexedDB* technology. The name of this database is the same as the name of the simulation model. It can be inspected with the help of the browser's developer tools, which are typically activated with the key combination [Shift]+[Ctrl]+[I]. For instance, in Google's Chrome browser, one has to go to Application/Storage/IndexedDB.

The experiment statistics database consists of three tables containing data about (1) experiment definitions, (2) experiment runs, and (3) experiment scenario runs, which can be exported to a CSV file. By default, the statistics data obtained from running all replications of an experiment scenario is stored in averaged form as one experiment scenario run record.

When the output statistics of each single experiment scenario run is to be stored (for later analysis), this can be achieved by setting the Boolean experiment attribute `storeEachExperimentScenarioRun` to *true*, as in

```
sim.experiment.storeEachExperimentScenarioRun = true;
```

When time series data is to be created and stored for certain statistics variables, this can be achieved by listing the names of the variables in the experiment attribute `timeSeriesStatisticsVariables`, as in

```
sim.experiment.timeSeriesStatisticsVariables = ["arrivedCustomers","departedCustomers"];
```

2.7. 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.

Simulation runs can be animated by *visualizing* objects and events, by *sonifying* events and by allowing human users to *interact* with the simulated world. OESjs allows adding 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, objects and events can be visualized by defining a *view* for them. An *object view* is defined by a 2D shape (like a *rectangle* or a *polygon*) or a 3D shape (like a *cuboid* or a *mesh*). An *event view* consists of an animation defined in the form of a *Web*

Animation (of one or more DOM elements using key frames). Events can also be *sonified* by attaching specific sounds to event occurrences in an event appearance definition.

2. A **user interaction UI** allows human users to interact with a running simulation by taking decisions on the values of decision variables or by taking actions that change the value of certain simulation variables.

3. A **participation UI** allows human users to participate in a multi-agent simulation scenario by receiving situational information and by performing in-world actions via the user interface. Any multi-agent simulation model can be turned into a user-interactive *participatory simulation* by adding a *participation model* and a corresponding UI.

2.7.1. Adding an Observation User Interface

For being able to observe a simulation run, some form of visualization has to be defined. OESjs supports both the visualization of *spatial models* and of non-spatial models. In a visualization of a non-spatial model, such as the ServiceDesk-1 model, all object views have to be explicitly positioned in an observation canvas. Rich two-dimensional visualizations can be obtained by using the web technology of *Scalable Vector Graphics (SVG)* in the definition of the observation UI.

In the case of our *ServiceDesk-1* model, we may, for instance, visualize the service desk using either an image or simply a fixed-size rectangle, and its queue in the form of a growing and shrinking bar.

For defining an observation UI with SVG-based visualization, the following settings have to be made:

```
sim.scenario.observationUI.type = "SVG";
sim.scenario.observationUI.canvas.width = 600;
sim.scenario.observationUI.canvas.height = 300;
```

Then we first define the fixed elements of the visualization, giving each one a name (here: "desk") and defining an SVG shape with attributes and a CSS style:

```
sim.scenario.observationUI.fixedElements = {
  "desk": {
    shapeName: "rect",
    shapeAttributes: { x: 350, y: 200, width: 50, height: 30},
    style: "fill:brown; stroke-width:0"
  }
};
```

For learning more about SVG shapes and their attributes, see the book chapter Basic Shapes & Paths by Joni Trythall. For learning more about CSS styling of SVG elements, see Styling And Animating SVGs With CSS by Sara Soueidan.

The main issue in visualization is to map the state variables of interest to suitable visual parameters such as colors, shape size, etc. For instance, we may want to map the *queueLength* attribute to the width of a rectangle, as in the following object view definition:

```
sim.scenario.observationUI.objectViews = {
  "serviceDesk1": [  // a view of the queue
    { shapeName: "rect",  // a rectangle defined by
      shapeAttributes: {  // left-upper corner (x,y) as well as width and height
        x: function (sd) {return Math.max( 0, 330 - sd.queueLength * 20);},
        width: function (sd) {return Math.min( 300, sd.queueLength * 20);},
        y: 150, height: 80
      },
      style:"fill:yellow; stroke-width:0"
    },
    { shapeName: "text",
      shapeAttributes: {x: 325, y: 250,
          textContent: function (sd) {return sd.queueLength;}},
      style:"font-size:14px; text-anchor:middle"
    }
  ]
};
```

2.8. Simulation Programming with OESjs

2.8.1. Using the Simulation Log

The OESjs simulator can generate a simulation log, which allows to inspect the evolving states of a simulation run. Inspecting the simulation log can help to understand the dynamics of a model, or it can be used for finding logical flaws in it.

The contents of the simulation log can be controlled by defining *short labels* for those objects and object properties as well as event types that we want to see in the log, using the `shortLabel` attribute. For instance, in the case of the service desk model, defining the short labels "sd1" for the service desk object, "qLen" for the `queueLength` property, "Arr" for the `CustomerArrival` event type and "Dep" for the `CustomerDeparture` event type leads to the following simulation log:

Simulation Log

Time	System State	Future Events
0	sd1{ qLen: 0}	Arr@1
1	sd1{ qLen: 1}	Arr@4, Dep@4
4	sd1{ qLen: 1}	Arr@5, Dep@8
5	sd1{ qLen: 2}	Dep@8, Arr@10
8	sd1{ qLen: 1}	Arr@10, Dep@12
10	sd1{ qLen: 2}	Dep@12, Arr@12
12	sd1{ qLen: 2}	Arr@15, Dep@16
15	sd1{ qLen: 3}	Dep@16, Arr@21
16	sd1{ qLen: 2}	Dep@18, Arr@21
18	sd1{ qLen: 1}	Dep@20, Arr@21
20	sd1{ qLen: 0}	Arr@21

2.823. Creating Objects and Events for the Initial State

In , we have shown how to create initial objects for the initial state of a simulation scenario using the map `sim.scenario.initialState.objects`. Whenever the initial state has to be populated with a larger set of objects, we can define a set-up procedure `sim.scenario.setupInitialState`, as in the following example where we create 100 `ServiceDesk` objects, each with an associated `CustomerArrival` event:

```
sim.scenario.setupInitialState = function () {
  var i=1;
  for (i=1; i <= 100; i++) {
    sim.addObject( new ServiceDesk({
      id: i,
      typeName: "ServiceDesk",
      queueLength: 0
    }));
    sim.scheduleEvent( new CustomerArrival( {
      occTime: 1,
      serviceDesk: i
    }));
  }
};
```

2.8.3. 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 one. 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 by ID 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 (objIdStr) {
  var obj = sim.objects[objIdStr];
  ...  // 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 (objIdStr) {
  var obj = cLASS["ServiceDesk"].instances[objIdStr];
  ...  // 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.

2.8.4. Defining and Using a History for an Attribute of an Object

There are use cases which require to construct a history of the changing values of a certain attribute for a specific object and evaluate or simply display this history. For example, we may define a history for the attribute `queueLength` of service desks using the `historySize` parameter:

```
var ServiceDesk = new cLASS({
  Name: "ServiceDesk",
  supertypeName: "oBJECT",
  properties: {
    "queueLength": { range: "NonNegativeInteger", historySize: 7,
        label: "Queue length", shortLabel: "qlen"}
  }
});

In such
```

In such a case, the OESjs simulator automatically constructs a history buffer of the specified size, which can, for instance, be converted to a string with the expression

```
sim.namedObjects["serviceDesk1"].history.queueLength.toString()
```

A *history buffer* is a ring buffer, having a limited size and an `add` operation for adding new items to the buffer as in:

```
sim.namedObjects["serviceDesk1"].history.queueLength.add( this.queueLength);
```

Notice that the oldest item may get lost when a (fixed-size) buffer is already full and a new item is added.

```
Object.keys( cLASS["ServiceDesk"].instances).forEach( function (objIdStr) {
  var obj = cLASS["ServiceDesk"].instances[objIdStr];
  ...  // 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.

Index