Tutorial: Discrete Event Simulation with OESjs-Core1

How to create and run simulations with the JavaScript-based simulation framework OESjs Core1 available from the OES GitHub repo

Gerd Wagner G.Wagner@b-tu.de

Published 2023-01-31

Available as HTML and PDF.

Abstract

This tutorial article explains how to use the OESjs Core1 simulation framework, which implements an architecture for Object Event Simulation (OES), extending the OESjs Core 0 framework by adding fixed-increment time progression, a seedable random number generator, a set of sampling functions from various probability distributions (uniform, triangular, normal, exponential, etc.), multiple scenarios per model, multiple experiment types per model, model parameters, parameter variation experiments, as well as persistent storage and export of experiment results.

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.

Both static systems/structures and dynamic systems can be modeled and simulated. Modeling and simulation (M&S) of *static structures*, such as the surface textures of materials, is only an issue in physics and computer graphics, while M&S of dynamic systems is an issue in all scientific and engineering disciplines, including management science, economics and other social sciences.

A *dynamic system* may be subject to discrete or continuous state changes. For simulating a dynamic system one has to model

1. the types of objects it is composed of,
2. the types of events that cause discrete state changes of objects,
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).

A (purely) continuous dynamic system does not include events and their causal effects (list items 2-4), but only objects that are subject to continuous state changes (list items 1 and 5). A (purely) discrete dynamic system does not include any continuous state changes of objects (list item 5). Many real-world systems include both discrete and continuous state changes, in which case they may be called *hybrid* dynamic systems.

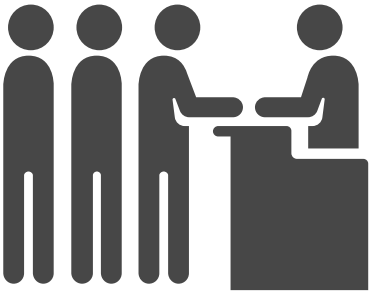
In this tutorial, we are only concerned with discrete state changes. Consequently, we only consider purely discrete dynamic systems, also called *discrete event systems*, which consist of:

- **objects** (of various types) whose states may be changed by
- **events** (of various types) occurring in a sequence of time points, causing state changes of affected objects and follow-up events.

This means that for modeling such a 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 of the System under Investigation



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 system under investigation 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 design based on the conceptual model of the system under investigation, wee need to abstract away from many items of the conceptual model for obtaining a sufficiently simple design. 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 *Service-Desk-0*, has only one global variable: *queueLength*. But, as an alternative, more explicit, model, called *Service-Desk-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 *Arrival* and *Departure*.

The event type *Arrival* 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 time* 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 *Freq{ 2:0.3, 3:0.5, 4:0.2}*.

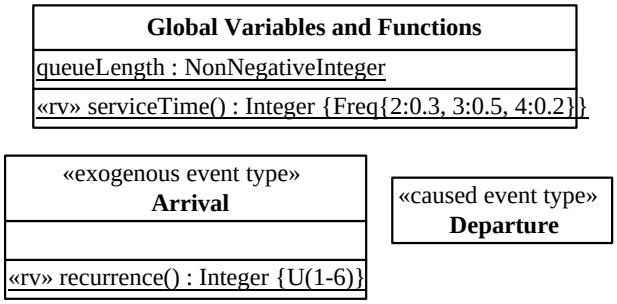
Computationally, object types and event types correspond to classes, either of an object-oriented information model, such as a UML class diagram, or of a computer program written in an object-oriented programming language, such as Java or JavaScript.

1.2.1. Service-Desk-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 Service-Desk-0) has only one global variable: *queueLength*, which is a non-negative integer, and a global function for computing the random service time, but no object type.

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

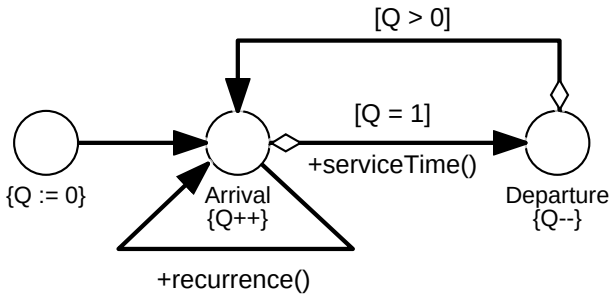
Figure 1-1. *An information design model for Service-Desk-0*



In addition to an information design model for defining the simulation system's state structure, we also need to make a process design model for defining the dynamics of the simulation 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 a process model diagram or in pseudo-code, or in a simulation or programming language. The following *Event Graph* provides a process design model for the Service-Desk-0 simulation scenario. Circles represent events (or, more precisely, event types) and arrows, which may be annotated with a delay expression, such as +serviceTime(), represent event scheduling relationships. An arrow with a mini-diamond at its source end represents a conditional event scheduling relationship where the condition is expressed in brackets below or above the arrow.

Figure 1-2. *A process design model in the form of an Event Graph, where the state variable Q stands for queueLength*



Event Graphs have originally been proposed by L. Schruben (1983). Their visual syntax has been improved and harmonized with the business process modeling language *BPMN* in the *Discrete Event Process Modeling Notation (DPMN)* proposed by Wagner (2018) and more thoroughly described in the book Discrete Event Simulation Engineering.

The following table shows the two event rules defined by the above Event Graph, expressed in pseudo-code.

ON (event type)	DO (event routine)

Arrival @ t	`INCREMENT queueLength`
	`IF queueLength = 1 THEN`
	`  sTime := serviceTime()`
	`  SCHEDULE Departure @ (t + sTime)`

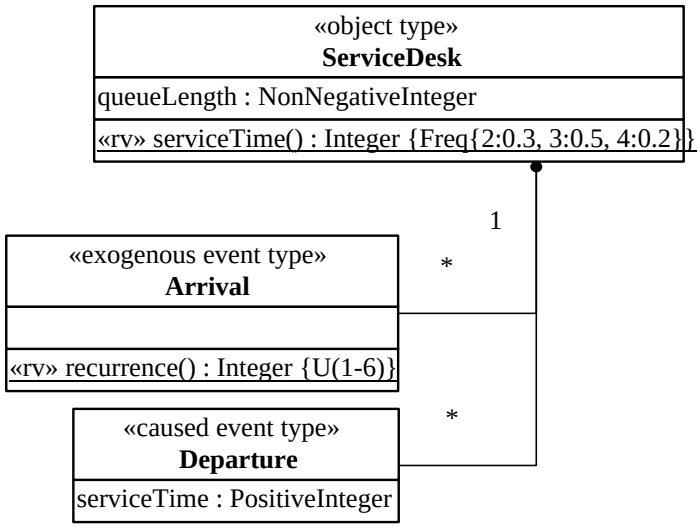
Departure @ t	`DECREMENT queueLength`
	`IF queueLength > 0 THEN`
	`  sTime := serviceTime()`
	`  SCHEDULE Departure @ (t + sTime)`

1.2.2. Service-Desk-1: Modeling *queueLength* as an attribute

In our extended model (Service-Desk-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 *Arrival* and *Departure*, 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 *Departure* class for being able to update the service utilization statistics when a customer departs.

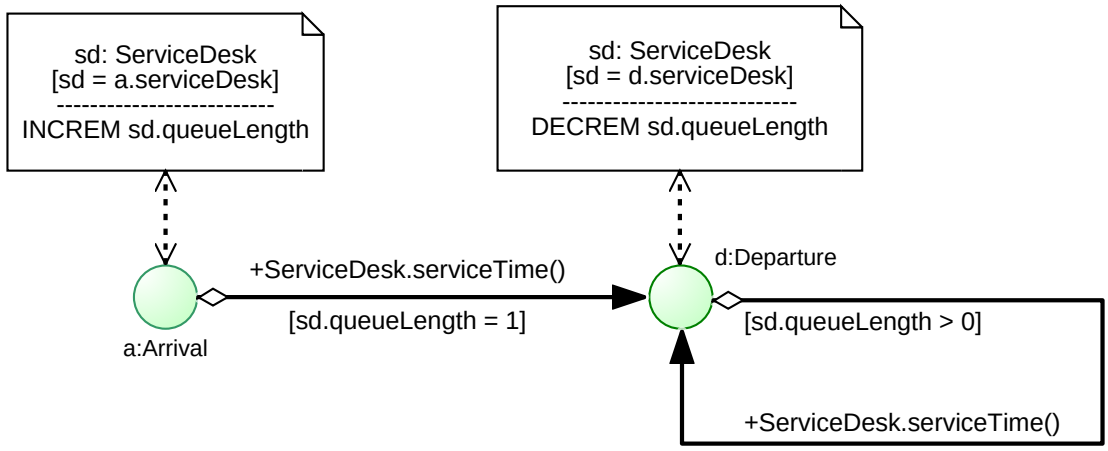
Both event types, *Arrival* and *Departure*, now have a many-to-one association with the object type *ServiceDesk*. This expresses the fact that any such event occurs at a particular service desk, which participates in the event. This association is implemented in the form of a reference property *serviceDesk* in each of the two event types, as shown in **Figure 1-3.** An information design model for Service-Desk-1.

Figure 1-3. *An information design model for Service-Desk-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 *DPMN Process Diagram* provides a process design model for the Service-Desk-1 simulation scenario. As in Event Graphs, circles represent event types and arrows represent event scheduling relationships. DPMN extends Event Graphs by adding object rectangles, attached to event circles, representing state change patterns for objects that are affected by events of that type.

Figure 1-4. *A process design model in the form of a DPMN Process Diagram*



The following table shows the two event rules defined by the DPMN diagram, 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)
Arrival(sd) @ t	`INCREMENT sd.queueLength`

ON (event type)	DO (event routine)

with sd : ServiceDesk	`IF sd.queueLength = 1 THEN`
	`  sTime := ServiceDesk.serviceTime()`
	`  SCHEDULE Departure( sTime, sd) @(t + sTime)`

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

Chapter 2. Creating Object Event Simulations with OESjs-Core1

The JavaScript-based simulation framework *OESjs Core1* implements the *Object Event Simulation (OES)* paradigm, representing a general *Discrete Event Simulation* approach based on *object-oriented* modeling and *event scheduling*.

The code of an OESjs Core1 simulation consists of (1) the OESjs Core1 framework files in the folder `framework`, (2) general library files in the `lib` folder and (3) the following files to be created by the simulation developer:

1. For each object type *ObjT*, a JS code file `ObjT.js`.
2. For each event type *EvtT*, a JS code file `EvtT.js`.
3. A `simulation.js` file defining further parts of the simulation, such as statistics variables and the initial state.

OESjs Core1 supports three 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.
2. Simple simulation experiments, which are defined as a set of replicated simulation scenario runs, providing summary statistics like mean, standard deviation, minimum/maximum and confidence intervals for each statistics variable defined in the underlying model.
3. Parameter variation experiments, for which a set of experiment parameters with value sets are defined such that each experiment parameter corresponds to a model parameter. When an experiment is run, each experiment parameter value combination defines an experiment scenario, which is run repeatedly, according to the specified number or replications for collecting statistics.

OESjs Core1 allows to define two or more simulation scenarios for a given model. While an experiment type is defined for a given model, an experiment of that type is run on top of a specific scenario.

Using a simulation library like OESjs Core1 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).

The following sections present the basic concepts of the OESjs Core1 simulation library.

Attention

You can download OESjs Core1 in the form of a ZIP archive file from the OES GitHub repo. After extracting the archive on your local disk, you can run any of its example models by going to its folder and loading its `index.html` file into your browser. You can create your own model by making a copy of one of the example model folders and using its code files a s starting point.

Since an OESjs simulation includes a JS worker file for running the simulator in its own thread separately from the main (user interface) thread, it cannot be run from the local file system without changing the browser's default configuration (due to the web security policy CORS).

For developing OESjs simulations on your computer, you should use *Firefox* because its security settings can be easily configured such that it allows loading JS worker files directly from the local file system by disabling the flag "strict_origin_policy" specifically for file URLs:

1. Enter "**about:config**" in the Firefox search bar.
2. Search for "**security.fileuri.strict_origin_policy**".
3. Disable this policy by changing its value from **true** to **false**.

This creates only a small security risk because the web security policy called "CORS" is only disabled for file URLs, but not for normal URLs.

For other browsers, like Chrome, you need to install a local HTTP server and load your simulation's *index.html* file from that local server, or run it via the JS development tool *WebStorm* (which has a built-in local server), because the only option for loading JS worker files from the local file system in Chrome would be to disable the CORS policy completely (see how to disable CORS in Chrome), but that would create a more severe security risk and is therefore not recommended.

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", "min", "hour", "day", "week", "month" or "year", as in

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

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 = "day";
   ```

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

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

2.1.1. Time Granularity

A model's *time granularity* is the time delay until the next moment, such that the model does not allow considering an earlier next moment. This is captured by the simulation parameter *nextMomentDeltaT* used by the simulator for scheduling immediate events with a minimal delay. When a simulation model is based on discrete time, *nextMomentDeltaT* is set to 1, referring to the next time point. When a simulation model is based on continuous time, *nextMomentDeltaT* is set to the default value 0.001, unless the model parameter `sim.model.nextMomentDeltaT` is explicitly assigned in the `simulation.js` file.

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 **next-event** time progression and **fixed-increment** time progression, as well as their combination.

An OESjs-Core1 model with fixed-increment time progression has to define a suitable periodic time event type, like `EachSecond` or `EachDay` in the form of an exogenous event type with a recurrence function returning the value 1. Such a model can be used for

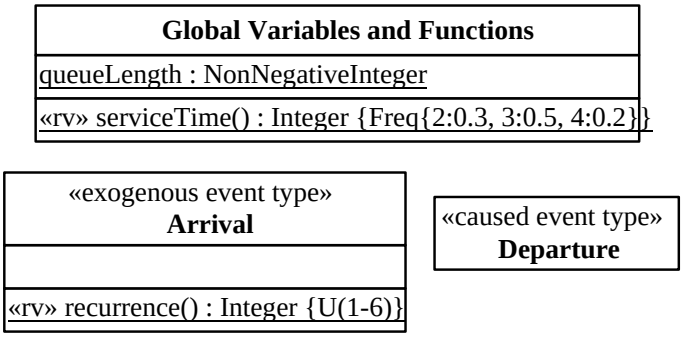
1. modeling continuous state changes (e.g., objects moving in a continuous space), or
2. making a discrete model that abstracts away from explicit events and uses only implicit periodic time events ("ticks"), which is a popular approach in social science simulation.

Examples of discrete event simulation models with fixed-increment time progression and no explicit events are the Schelling Segregation Model and the Susceptible-Infected-Recovered (SIR) Disease Model.

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, *serviceTime*(), and two event types, as shown in the following class diagram:



Notice that this model does not define any object type, which implies that the system state is not composed of the states of objects, but of the states of model variables, here it consists of the state of the model variable *queueLength*. The discrete random variable for modeling the random variation of service durations is implemented as a model function `serviceTime` shown in the *Global Variables and Functions* class. It samples integers between 2 and 4 from the empirical probability distribution *Frequency{ 2:0.3, 3:0.5, 4:0.2}*. The model can be coded with OESjs-Core1 in the following way:

```
// (global) model variable
sim.model.v.queueLength = 0;
// (global) model function
sim.model.f.serviceTime = function () {
  var r = math.getUniformRandomInteger( 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
};
```

You can run this Service-Desk-0 model from the project's GitHub website. An example of a run of this model is shown in the following simulation log:

Table 2-1. *Simulation Log*

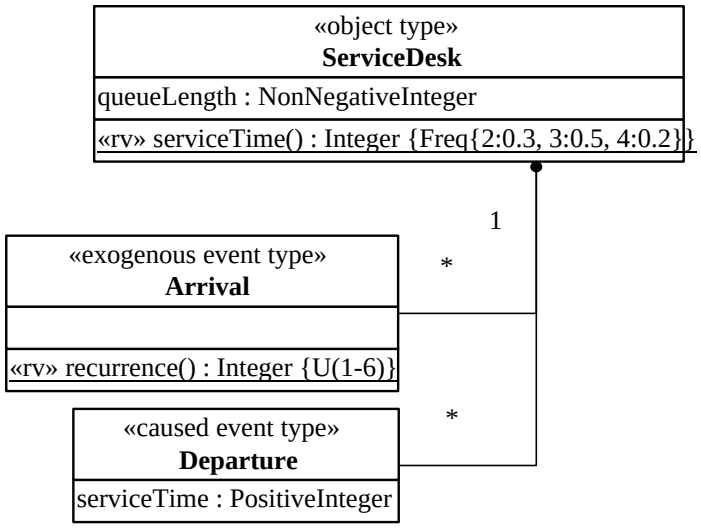
Step	Time	System State	Future Events
0	0	queueLength: 0	CustomerArrival@1
1	1	queueLength: 1	CustomerDeparture@4, CustomerArrival@4
2	4	queueLength: 1	CustomerDeparture@6, CustomerArrival@7
3	6	queueLength: 0	CustomerArrival@7

Step	Time	System State	Future Events
4	7	queueLength: 1	CustomerDeparture@11, CustomerArrival@13
5	11	queueLength: 0	CustomerArrival@13
6	13	queueLength: 1	CustomerDeparture@15, CustomerArrival@19
7	15	queueLength: 0	CustomerArrival@19
...	...	...	...
49	114	queueLength: 0	CustomerArrival@117
50	117	queueLength: 1	CustomerArrival@118, CustomerDeparture@119
51	118	queueLength: 2	CustomerDeparture@119, CustomerArrival@119
52	119	queueLength: 2	CustomerArrival@121, CustomerDeparture@123
53	121	queueLength: 3	CustomerDeparture@123, CustomerArrival@124
54	123	queueLength: 2	CustomerArrival@124, CustomerDeparture@126
55	124	queueLength: 3	CustomerArrival@125, CustomerDeparture@126
56	125	queueLength: 4	CustomerDeparture@126, CustomerArrival@128
57	126	queueLength: 3	CustomerArrival@128, CustomerDeparture@128
58	128	queueLength: 3	CustomerArrival@129, CustomerDeparture@131
59	129	queueLength: 4	CustomerDeparture@131, CustomerArrival@133
60	131	queueLength: 3	CustomerArrival@133, CustomerDeparture@135
61	133	queueLength: 4	CustomerDeparture@135, CustomerArrival@137
62	135	queueLength: 3	CustomerArrival@137, CustomerDeparture@137
63	137	queueLength: 3	CustomerArrival@139, CustomerDeparture@141
64	139	queueLength: 4	CustomerDeparture@141, CustomerArrival@142

Step	Time	System State	Future Events
65	141	queueLength: 3	CustomerArrival@142, CustomerDeparture@144
66	142	queueLength: 4	CustomerDeparture@144, CustomerArrival@147
67	144	queueLength: 3	CustomerArrival@147, CustomerDeparture@148
68	147	queueLength: 4	CustomerDeparture@148, CustomerArrival@148
69	148	queueLength: 4	CustomerArrival@149, CustomerDeparture@151
70	149	queueLength: 5	CustomerDeparture@151, CustomerArrival@151
...	...	...	...

2.2.2. Object Types

Object types are defined in the form of classes. Consider the object type *ServiceDesk* defined in the following *Service-Desk-1* model:



While `queueLength` was defined as a global variable in the *Service-Desk-0* model, it is now defined as an attribute of the object type *ServiceDesk*:

```
class ServiceDesk extends oBJECT {
  constructor({ id, name, queueLength}) {
    super( id, name);
    this.queueLength = queueLength;
  }
  static serviceTime() {
    var r = math.getUniformRandomInteger( 0, 99);
    if ( r < 30) return 2;         // probability 0.3
```

```
    else if ( r < 80) return 3;    // probability 0.5
    else return 4;                 // probability 0.2
  }
}
ServiceDesk.labels = {"queueLength":"qLen"};  // for the log
```

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. When a property has a `label` (defined by the class-level (map-valued) property `labels`), it is shown in the simulation log.

You can run this simulation model from the project's GitHub website.

2.2.3. Event Types

In OES, there is a distinction between two kinds of events:

1. events that are *caused* by other event occurrences during a simulation run;
2. *exogenous* events that 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 in OESjs-Core1:

```
class CustomerArrival extends eVENT {
  constructor({ occTime, serviceDesk}) {
    super( occTime);
    this.serviceDesk = serviceDesk;
  }
  onEvent() {
    ...
  }
  ...
}
```

The definition of the *CustomerArrival* event type includes a reference property *serviceDesk*, which is used for referencing the service desk object at which a customer arrival 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 ("static") method:

```
class CustomerArrival extends eVENT {
  ...
  static recurrence() {
    return math.getUniformRandomInteger( 1, 6);
  }
```

```
}
```

Notice that the *recurrence* function of *CustomerArrival* is coded with the library method `math.getUniformRandomInteger`, which allows sampling from discrete uniform probability distribution functions.

In the case of an exogenous event type definition, a *createNextEvent* method has to be defined for assigning event properties and returning the next event of that type, which is scheduled by invoking the *recurrence* function for setting its *ocurrenceTime* and by copying all participant references (such as the *serviceDesk* reference).

```
class CustomerArrival extends eVENT {
  ...
  createNextEvent() {
    return new CustomerArrival({
      occTime: this.occTime + CustomerArrival.recurrence(),
      serviceDesk: this.serviceDesk
    });
  }
  static recurrence() {...}
}
```

When an OE simulator processes an exogenous event *e* of type *E*, it automatically schedules the next event of type *E* by invoking the *createNextEvent* method on *e*, if it is defined, or, otherwise by duplicating *e* and resetting its occurrence time by invoking *E*.recurrence().

For an exogenous event type, it is an option to define a maximum number of event occurrences by setting the static attribute `maxNmrOfEvents`, as in the following example:

```
CustomerArrival.maxNmrOfEvents = 3;
```

The second event type of the *Service-Desk-1* model, *Departure*, is an example of a type of *caused* events:

```
class CustomerDeparture extends eVENT {
  constructor({ occTime, serviceDesk}) {
    super( occTime);
    this.serviceDesk = serviceDesk;
  }
  onEvent() {
    ...
  }
}
```

A caused event type does neither define a *recurrence* function nor a *createNextEvent* method.

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, event rules are coded as `onEvent` methods of the class

that implements the event type. These methods return a set of events (more precisely, a set of JS objects representing events).

Notice that in the DES literature, event rule methods are called *event routines*.

For instance, in the `CustomerArrival` class, the following event rule method is defined:

```
class CustomerArrival extends eVENT {
  ...
  onEvent() {
    var followupEvents=[];
    // increment queue length due to newly arrived customer
    this.serviceDesk.queueLength++;
    // update statistics
    sim.stat.arrivedCustomers++;
    if (this.serviceDesk.queueLength > sim.stat.maxQueueLength) {
      sim.stat.maxQueueLength = this.serviceDesk.queueLength;
    }
    // if the service desk is not busy
    if (this.serviceDesk.queueLength === 1) {
      followupEvents.push( new CustomerDeparture({
        occTime: this.occTime + ServiceDesk.serviceTime(),
        serviceDesk: this.serviceDesk
      }));
    }
    return followupEvents;
  }
}
```

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.

```
class CustomerDeparture extends eVENT {
  ...
  onEvent() {
    var followupEvents=[];
    // decrement queue length due to departure
    this.serviceDesk.queueLength--;
    // update statistics
    sim.stat.departedCustomers++;
    // if there are still customers waiting
    if (this.serviceDesk.queueLength > 0) {
      // start next service and schedule its end/departure
      followupEvents.push( new CustomerDeparture({
        occTime: this.occTime + ServiceDesk.serviceTime(),
        serviceDesk: this.serviceDesk
```

```
      }));
    }
    return followupEvents;
  }
}
```

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)`

Probability Distribution Function	OESjs Library Method	Example
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 David Bau's seedable random number generator seedrandom, 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 sampling functions from the `rand.js` library described above, for generating random numbers.

Otherwise, using a random seed does not guarantee reproducible simulation runs!

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.

An OESjs *simulation scenario* consists of

1. a simulation model;
2. simulation parameter settings, such as setting a value for `durationInSimTime` and `randomSeed`; and
3. an initial state definition.

An empty template for the `simulation.js` file has the following structure:

```
// ***** Simulation Model ******************
sim.model.time = "...";  // discrete or continuous
sim.model.timeIncrement = ...;   // optional
sim.model.timeUnit = "...";  // optional (ms|s|min|hour|day|week|month|year)
sim.model.v.aModelVariable = ...;  // (developer-defined) model variables
sim.model.f.aModelFunction = ...;  // (developer-defined) model functions
sim.model.p.aModelParameter = ...;  // (developer-defined) model parameters
sim.model.objectTypes = [...];  // (developer-defined) object types
sim.model.eventTypes = [...];  // (developer-defined) event types
// ***** Simulation Parameters **************
sim.scenario.durationInSimTime = ...;
sim.scenario.randomSeed = ...;    // optional
// ***** Initial State **********************
sim.scenario.setupInitialState = function () {
  // Initialize model variables
  ...
  // Create initial objects
  ...
  // Schedule initial events
  ...
};
// ***** Ex-Post Statistics *****************
sim.model.statistics = {...};
```

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

2.3.1. Model Parameters

While model variables are state variables whose values are changed as an effect of an event occurrence, *model parameters* are not part of the dynamic state of the simulated system, but are rather used for providing values that can only be read during a simulation run. The main purpose of model parameters is to allow *parameter variation experiments*.

2.3.2. Simulation Scenario Parameters

A few simulation parameters are predefined as attributes of the simulation scenario. The most important ones are:

- *durationInSimTime* - this attribute allows defining the duration of a simulation run; which runs forever when this attribute s not set;
- *randomSeed*: Setting this optional parameter to a positive integer allows to obtain a specific fixed random number sequence (generated by a 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.

2.3.3. Initial State

Defining an initial state means:

1. assigning initial values to global model 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 `setupInitialState` procedure takes care of these initial state definitions. A global model variable is initialized in the following way:

```
sim.scenario.setupInitialState = function () {
  // Initialize model variables
  sim.model.v.queueLength = 0;
  // Create initial objects
  ...
  // Schedule initial events
  ...
};
```

An initial state object is created by instantiating an object type of the simulation model with suitable initial property values, as shown in the following example:

```
sim.scenario.setupInitialState = function () {
  // Initialize model variables
  ...
  // Create initial objects
  const serviceDesk1 = new ServiceDesk({id: 1, queueLength: 0});
  // Schedule initial events
  ...
};
```

Notice that object IDs are positive integers.

Instead of assigning a **fixed value** to a property like `queueLength` for defining an object's initial state, as in `queueLength: 0`, we can also assign it an **expression**, as in `queueLength: Math.round(12/30)`.

An **initial event** is scheduled (or added to the *Future Events List)*, as shown in the following example:

```
sim.scenario.setupInitialState = function () {
  // Initialize model variables
```

```
  ...
  // Create initial objects
  const desk1 = new ServiceDesk({id: 1, queueLength: 0});
  // Schedule initial events
  sim.schedule( new CustomerArrival({occTime:1, serviceDesk: desk1}));
};
```

Initial objects or events can be parametrized with the help of model parameters.

2.3.4. Defining Alternative Scenarios with Different Initial States

For running a model on top of different initial states, one can define a list of scenarios, each with its own `setupInitialState` procedure:

```
sim.scenarios[1] = {
  scenarioNo: 1,
  title: "Scenario with two service desks",
  setupInitialState: function () {
    // Create initial objects
    var sD1 = new ServiceDesk({id: 1, queueLength: 0}),
        sD2 = new ServiceDesk({id: 2, queueLength: 0});
    // Schedule initial events
    sim.FEL.add( new CustomerArrival({occTime: 1, serviceDesk: sD1}));
    sim.FEL.add( new CustomerArrival({occTime: 2, serviceDesk: sD2}));
  }
};
sim.scenarios[2] = {...}
```

Before running a simulation, a specific scenario can be chosen in the user interface.

WARNING

Do not set model parameters in the `setupInitialState` procedure! This would interfere with parameter variation experiments in which the same parameter(s) are used.

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 *statistics variables* have to be defined, as in the following example:

```
sim.model.setupStatistics = function () {
  sim.stat.arrivedCustomers = 0;
  sim.stat.departedCustomers = 0;
```

```
  sim.stat.maxQueueLength = 0;
};
```

Statistics variables have to be updated in *onEvent* methods. For instance, the variables *arrivedCustomers* and *maxQueueLength* are updated in the onEvent method of the *CustomerArrival* event class:

```
class CustomerArrival extends eVENT {
  ...
  onEvent() {
    ...
    // update statistics
    sim.stat.arrivedCustomers++;
    if (this.serviceDesk.queueLength > sim.stat.maxQueueLength) {
      sim.stat.maxQueueLength = this.serviceDesk.queueLength;
    }
    ...
  }
}
```

In certain cases, a statistics variable can only be computed at the end of a simulation run. For this purpose, there is the option to define a *computeFinalStatistics* procedure:

```
sim.model.computeFinalStatistics = function () {
  // percentage of business days without stock-outs
  sim.stat.serviceLevel = (sim.time - sim.stat.nmrOfStockOuts) / sim.time * 100;
};
```

After running a simulation scenario, the statistics results are shown in a table:

Table 2-2. *Statistics*

arrivedCustomers	289
departedCustomers	288
maxQueueLength	4

2.5. Time Series

It is often desirable to observe the changes of a variable's value over time by looking at a temporal sequence of the values of a variable called a *time series*. Typically, a time series is plotted in a chart.

In OESjs, you can create a time series chart for a statistics variable or for an attribute of a specific object by assigning a corresponding definition to `sim.model.timeSeries` in `simulation.js` like so:

```
1 sim.model.timeSeries = {
2   "order quantity": {statisticsVariable: "orderQuantity"},
3   "liquidity": {objectId:1, attribute:"liquidity"}
4 };
```

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 parameters* (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.

An experiment type is defined for a given simulation model and an experiment of that type is run on top of a given simulation scenario for that model.

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)

Simple Experiments

A simple experiment type is defined with a `sim.experimentType` record on top of a model 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 type definition:

```
sim.experimentType = {
  title: "Simple Experiment with 10 replications, each running for 1000 time units (days)",
  nmrOfReplications: 10,
  seeds: [123, 234, 345, 456, 567, 678, 789, 890, 901, 1012]
};
```

Running this simple experiment means running the underlying scenario 10 times, each time with another random seed, as specified by the list of seeds. The resulting statistics are composed of the statistics for each replication complemented with summary statistics listing averages, standard deviations, min/max values and 95% confidence intervals, as shown in the following example:

Experiment Results			
Replication	**Statistics**		
	arrivedCustomers	**departedCustomers**	**maxQueueLength**
1	285	283	7
2	274	274	6
3	285	285	4
4	287	286	5
5	284	284	6
6	300	299	4
7	288	286	5
8	286	284	4
9	286	285	4

Experiment Results			
Replication	Statistics		
	arrivedCustomers	departedCustomers	maxQueueLength
10	295	293	6
Average	287	285.9	5.1
Std.dev.	6.848	6.506	1.101
Minimum	274	274	4
Maximum	300	299	7
CI Lower	282.9	281.9	4.4
CI Upper	291	289.6	5.7

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.

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.

An experiment parameter must have the same name as the model parameter to which it refers. It defines a set of values for this model parameter, 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.

The following code shows an example of a parameter variation experiment definition (on top of the Inventory-Management simulation model):

```
sim.experimentTypes[1] = {
  id: 1,
  title: "Parameter variation experiment for exploring reorderInterval and targetInventory"
  nmrOfReplications: 10,
  seeds: [123, 234, 345, 456, 567, 678, 789, 890, 901, 1012],
  parameterDefs: [
    {name:"reviewPolicy", values:["periodic"]},
    {name:"reorderInterval", values:[2,3,4]},
    {name:"targetInventory", startValue:80, endValue:100, stepSize:10},
  ]
};
```

Notice that this experiment definition defines 9 experiment scenarios resulting from the combinations of the values 2/3/4 and 80/90/100 for the parameters *reorderInterval* and *targetInventory*. Running this parameter variation experiment means running each of the 9 experiment scenarios 10 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 Results				
Experiment scenario	Parameter values	Statistics		
		nmrOfStockOuts	lostSales	serviceLevel
0	periodic,2,80	21.8	180.7	97.82
1	periodic,2,90	7.4	55.9	99.26
2	periodic,2,100	2.1	15.8	99.79
3	periodic,3,80	86.6	855.6	91.34
4	periodic,3,90	40.6	377.5	95.94
5	periodic,3,100	16.3	139.8	98.37
6	periodic,4,80	171.5	2067.5	82.85
7	periodic,4,90	110.6	1238.3	88.94
8	periodic,4,100	63.8	661.4	93.62

Storage and Export of Experiment Results

In OESjs-Core1, an experiment's output 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 runs, (2) experiment scenarios, and (3) experiment scenario runs, which can be exported to a CSV file.

2.7. Using the Simulation Log

The OESjs-Core1 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 *labels* for those object properties that are to be displayed in the log. For instance, in the case of the Service-Desk-1 model, a label "qLen" is defined for the `queueLength` property of *ServiceDesk* objects by setting

```
ServiceDesk.labels = {"queueLength":"qLen"};
```

This results in the following simulation log:

Step	Time	System State	Future Events
0	0	Service-Desk-1{ qLen: 0}	CustomerArrival@1
1	1	Service-Desk-1{ qLen: 1}	CustomerDeparture@5, CustomerArrival@6

Step	Time	System State	Future Events
2	5	Service-Desk-1{ qLen: 0}	CustomerArrival@6
3	6	Service-Desk-1{ qLen: 1}	CustomerArrival@7, CustomerDeparture@10
4	7	Service-Desk-1{ qLen: 2}	CustomerDeparture@10, CustomerArrival@10
5	10	Service-Desk-1{ qLen: 2}	CustomerArrival@12, CustomerDeparture@13
6	12	Service-Desk-1{ qLen: 3}	CustomerDeparture@13, CustomerArrival@16
7	13	Service-Desk-1{ qLen: 2}	CustomerArrival@16, CustomerDeparture@16
8	16	Service-Desk-1{ qLen: 2}	CustomerDeparture@19, CustomerArrival@21
9	19	Service-Desk-1{ qLen: 1}	CustomerArrival@21, CustomerDeparture@23

2.8. Observation and User Interaction

Being able to observe a simulation run with the help of visualization (and sonification) is important for educational simulations and games, but it can also be used as a general tool for testing, inspecting and validating simulations. Both objects and events by can be *visualized*, while events can also be *sonified*. A simulation can be turned into an even more immersive experience 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 *views* for them. An *object view* is defined by a 2D shape (like a *circle* or a *polygon*) or a 3D shape (like a *sphere* or a *mesh*). An *event view* consists 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 taking over an agent for receiving situational information and 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.

How to define an observation UI or a user interaction UI is described in the next chapter.

Chapter 3. Simulation Programming with OESjs

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 important simulation programming issues.

3.1. 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 with { id: 1, name:"serviceDesk1", ...} has the ID number 1 and the name "serviceDesk1". It can be retrieved by ID from the collection `sim.objects` (a JS Map) in the following way:

```
var object1 = sim.objects.get( 1);
```

It can also be retrieved by name from the collection `sim.namedObjects` (also a JS Map) in the following way:

```
var object1 = sim.namedObjects.get("serviceDesk1");
```

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

```
for (const obj of sim.objects.values()) {
  ...  // do something with obj
}
```

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

```
for (const objIdStr of Object.keys( sim.Classes["ServiceDesk"].instances)) {
  const obj = sim.Classes["ServiceDesk"].instances[objIdStr];
  ...  // do something with obj
}
```

Here, `sim.Classes` provides a map from class names to classes, which are special JS objects, such that for a class `C`, the collection `C.instances` (a JS map/object) provides a map from object IDs to JS objects. Thus, the expression `Object.keys( sim.Classes["ServiceDesk"].instances)` represents an array/list of object ID strings, namely the keys of the map `sim.Classes["ServiceDesk"].instances`.

3.2. Defining and Using a History Attribute

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 attribute `temperatureHistory` in addition to the attribute `temperature` for recording the history of average daily temperatures:

```
class LemonadeMarket extends DailyDemandMarket {
  constructor({id, name, temperature}) {
```

```
    super({id, name});
    this.temperature = temperature;
    this.temperatureHistory = new RingBuffer();
    this.temperatureHistory.add( temperature);
  }
```

The value of such a history attribute is a *ring buffer*, having a limited size and an `add` operation for adding new items to the buffer as in the last constructor statement above.

Notice that the oldest item of such a buffer may get lost when the buffer is already full and a new item is added.

The value of a history attribute can be converted to a string with the help of the expression

```
sim.namedObjects["lemonadeMarket"].temperatureHistory.toString()
```

Chapter 4. Defining User Interfaces

The OESjs simulation framework allows defining user interfaces for various purposes on top of a simulation model:

1. A *model parameter UI* allows the user to modify the values of parameters without changing the simulation code.
2. An *initial state UI* allows modifying the attribute values of initial objects and events.
3. An *observation UI* allows defining views for objects and events (and sounds for events) such that they can be visualized (and sonified) during a simulation run.
4. A *user interaction UI* allows defining user interactions bound to certain events, such that a simulation can be turned into a game.

4.1. Defining an Observation User Interface

An observation UI allows defining views for objects and events (and sounds for events) such that they can be visualized (and sonified) during a simulation run. Since OESjs is a framework for web-based simulation, an observation UI is based on the following Web technologies: CSS, SVG and Web Animations. 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.

Visualizing Objects

For being able to observe objects in a simulation run, they have to be visualized in some form. OESjs supports both the visualization of objects in space in *spatial models* and of objects in 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.

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.

Two-dimensional visualizations can be obtained by using the web technology of *Scalable Vector Graphics (SVG)* in the definition of the observation UI. For defining an observation UI with SVG-based visualization, the following settings have to be made:

```
1 sim.config.obs.ui.type = "SVG";
2 sim.config.obs.ui.canvas.width = 600;
3 sim.config.obs.ui.canvas.height = 300;
```

In addition, one can define a CSS style for the canvas element in the following way, e.g., for setting a background color or background image:

```
1 sim.config.obs.ui.canvas.style = "background-color: azure";
```

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

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

The main issue in visualization is to visualize simulation objects by defining suitable views for them and then map some of their attributes to suitable visual parameters such as color, shape width and height, etc. A ***view*** can be defined either for all instances of an object type or for specific instances only.

For instance, we may want to visualize the waiting line of the object "serviceDesk1" in the form of a rectangle and map the service desk's *queueLength* attribute to the width of that rectangle, as in the following object view definition:

```
sim.config.obs.ui.objectViews = {
  "serviceDesk1": {  // the name of the object
    visualizationAttributes: ["queueLength"],
    attributesViewItemsRecords: [
      { attributes:["queueLength"],
        viewItems: [  // a list of 2 view elements for the object "serviceDesk1"
          {shapeName: "rect",  // a rectangle
           shapeAttributes: {  // left-upper corner (x,y) as well as width and height
             x: sd => Math.max( 0, 330 - sd.queueLength * 20),
             width: sd => Math.min( 300, sd.queueLength * 20),
             y: 150, height: 80 },
           style:"fill:yellow; stroke-width:0"},
          {shapeName: "text",
           shapeAttributes: {x: 325, y: 250, textContent: sd => sd.queueLength},
           style: "font-size:14px; text-anchor:middle"}
    ]
  }
};
```

Notice that the view consists of two elements: a rectangle and an attached text displaying the queue length. In the view definition, certain attributes are assigned a fixed value, while others are assigned a JS function expression, which codes the mapping of object attributes to visual parameters.

Alternatively, instead of defining a view for a specific service desk object, we can also define a view for all service desk objects, like so

```
sim.config.obs.ui.objectViews = {
  "ServiceDesk": [  // the name of the object type/class
    { 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"
```

```
    }
};
```

Visualizing Events

4.2. Defining a User Interface for User Interactions

In an event-based simulation, implicit actions bound to events of a certain type can be replaced by *user actions*, such that when an event of that type occurs, the simulation halts and the user gets the opportunity to perform an action (or take a decision) that determines how the simulation is continued. This type of interaction requires a suitable user interface that allows the user to enter values for action/decision parameters or select one of a number of possible actions.

A *user interaction user interface (UI)* allows defining user actions bound to events of a certain type. Such a UI turns a simulation into a game, or a *human-in-the-loop* simulation.

For instance, in the simulation of a lemonade stand (as an example of a manufacturing company), there could be an event type *StartOfDay* representing the start of a business day at which replenishment decisions have to be made. While these decisions would be made algorithmically in a normal simulation, a user interaction UI would allow a human user to make these decisions in a user-interactive (human-in-the-loop) simulation.

A user interaction (UIA) is triggered by a simulation event (of some type, possibly satisfying some condition) leading to the creation of a modal UIA window and an interruption of the simulation loop by having the browser wait for user input/actions. The UIA window contains the following UI elements:

1. *output fields* for informing the user about the current values of critical state variables,
2. *input fields* allowing the user to enter values for decision variables, and
3. one or more *buttons* (typically, a "continue" button allows confirming the choices made).

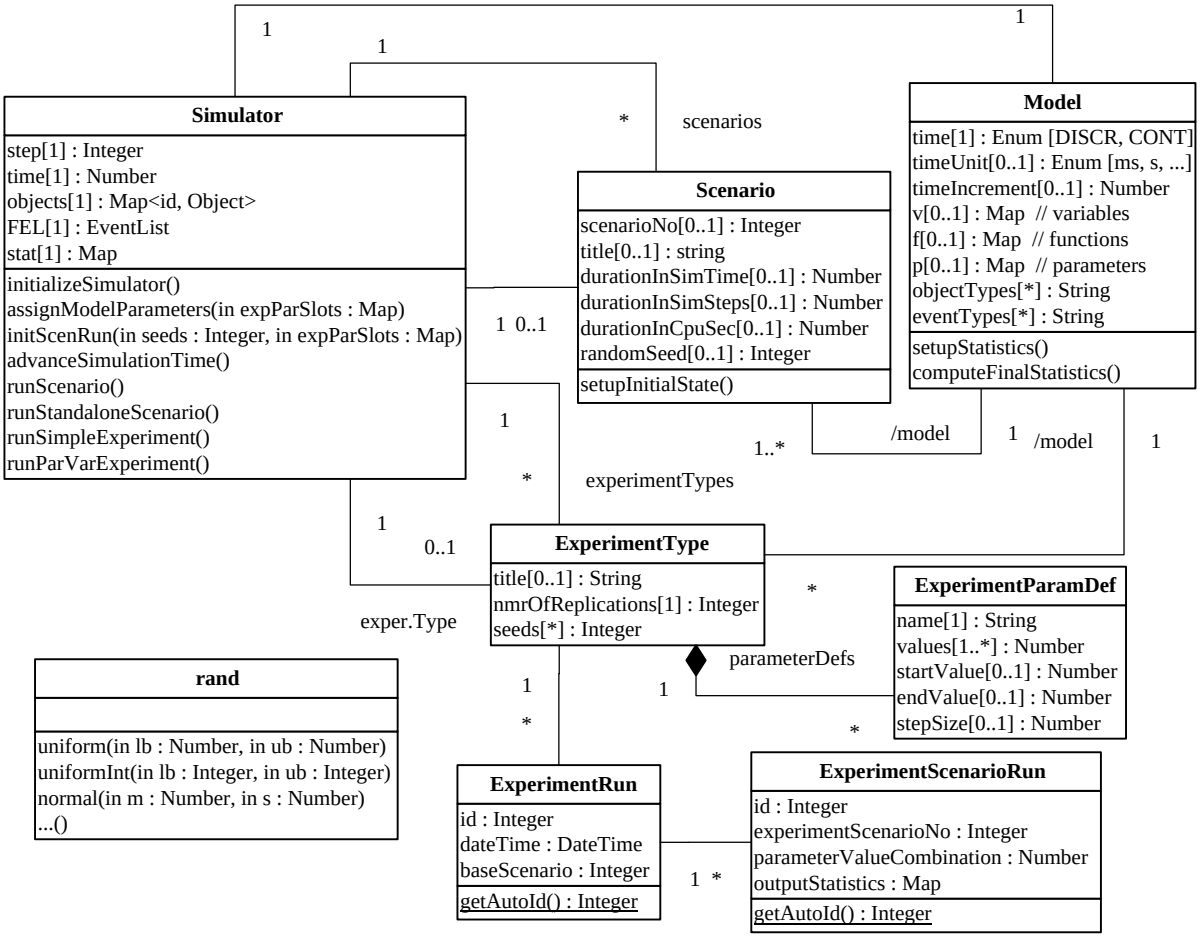
When the user confirms their choice(s) by activating the "continue" button, this triggers an event handler that restarts the simulator.

Appendix A. Simulator Architecture

OES Core 1 adds the following features to OES Core 0:

- fixed-increment time progression
- a seedable random number generator
- a set of sampling functions from various probability distributions (uniform, triangular, normal, exponential, etc.)
- multiple scenarios per model
- multiple experiment types per model
- model parameters
- parameter variation experiments
- persistent storage and export of experiment results

The OES Core 1 simulator's information architecture is described by the following class diagram, which defines the names of classes, properties and methods/functions:



Index

C

D

E

O

P

R

S