Proceedings of the 2016 Winter Simulation Conference
T. M. K. Roeder, P. I. Frazier, R. Szechtman, E. Zhou, T. Huschka, and S. E. Chick, eds.

INTRODUCTION TO SIMULATION USING JAVASCRIPT

Gerd Wagner

Department of Informatics
Brandenburg University of Technology
P. O. Box 101344
03013 Cottbus, GERMANY

ABSTRACT

JavaScript is a dynamic functional object-oriented programming language that can not only be used for enriching a web page, but also for implementing various kinds of web applications, including web-based simulations, which can be executed on front-end devices, such as mobile phones, tablets and desktop computers, as well as on powerful back-end computers, possibly in some cloud infrastructure. Although JavaScript cannot compete with strongly typed compiled languages (such as C++, Java and C#) on speed, it provides sufficient performance for many types of simulations and outperforms its competitors on ease of use and developer productivity, especially for web-based simulation. This tutorial provides a two-fold introduction: (1) to JavaScript programming using the topic of simulation, and (2) to simulation using the programming language JavaScript. It shows how to implement a Monte Carlo simulation, a continuous state change simulation and a discrete event simulation, using the power of JavaScript and the web.

1 INTRODUCTION TO SIMULATION

“Simulation” is an umbrella term subsuming a variety of use cases and approaches. Since we are only interested in computer simulation in this article, we can say that a simulation is provided by any computer program that imitates a static structure or a dynamic system of the real-world. This broad definition includes using

- a general-purpose programming language such as C++, Java, JavaScript, C#, etc., for implementing a simulation program, no matter if it is purely ad-hoc or based on a simulation paradigm;
- a simulation language such as Simula, SIMAN, Modelica, NetLogo, etc.;
- a simulation platform such as Arena, Simio, AnyLogic, etc.;
- an implementation of an abstract process formalism such as Petri Nets, State Charts, DEVS, Event Graphs, Activity Cycle Diagrams, etc.

Three main categories of simulations are often distinguished:

1. Monte Carlo simulation,
2. continuous simulation, and
3. discrete event simulation.

As explained in Wikipedia, *Monte Carlo methods* are “a broad class of computational algorithms that rely on repeated random sampling to obtain numerical results”. In principle, this approach can be used to solve any computational problem involving random variables. When Monte Carlo methods are used in a program for estimating certain performance indicators of real-world systems, such as in manufacturing or financial markets, this is often called *Monte Carlo simulation*, if the program does not involve the modeling of time and the simulation of state changes, but only a set of input random variables and a set of equations for computing the resulting output variables.

Continuous simulation, or more precisely: *continuous state change simulation*, is concerned with modeling a dynamic real-world system with the help of differential equations describing its state changes and simulating it by running a computational model based on a numerical solution of the differential equations. The computational model includes a discretization of the theoretically continuous time model by choosing a fixed time increment depending on a suitable granularity of time (e.g., 10 ms or 100 ms), and then running a fixed increment time progression loop that iteratively computes the next values of all state variables based on their current values.

Normally, continuous simulation is used for simulating *continuous dynamic systems*, in which continuous qualities, such as the spatial position or the velocity of a material object, or the temperature of an amount of matter, are subject to continuous changes. But continuous simulation can also be used for simulating discrete dynamic systems, such as biological populations or economic systems, by describing their discrete state changes approximatively with continuous state changes based on differential equations.

The term *Discrete Event Simulation (DES)* has been established as an umbrella term subsuming various kinds of computer simulation approaches, all based on the general idea of making a computational model of a *discrete dynamic system* by representing its state with the help of *state variables*, and modeling its dynamics by modeling the *events* that are responsible for its state changes. There is, however, no generally accepted definition of DES. Simulation textbooks and tutorials avoid defining the term "DES" in a precise way.

Pegden (2010) explains that the most fundamental DES approach, which he calls the *event worldview*, views a discrete dynamic system as a series of instantaneous events that change the state of the system over time, such that a DES model needs to define the events in the system and model the state changes that take place when those events occur. This is also the view adopted in DES textbooks, when they present implemented discrete event simulations based on the computational paradigm of *event scheduling* using a *future events list* and a *next-event time progression* loop that iteratively invokes the *event routines* of all imminent events.

As we have argued in (Guizzardi and Wagner 2010), *objects* and *events* are the most fundamental ontological categories for understanding and describing real-world systems, and therefore, the traditional *event worldview* should be extended to an *object-event worldview* where a system is viewed to consist of objects the state of which is changed by events. In this broader view, both objects and events are first-class citizens, the system state is the aggregation of all objects' states and the simulation model's state variables are provided by the attributes of the system's objects.

Notice that the *object-event worldview* is also supported, in some sense, by the success of the object-oriented modeling and programming paradigm in software engineeering, and by the remarkable fact that it is historically rooted in the concepts of objects, classes and inheritance originally pioneered by the simulation language *Simula*.

2 INTRODUCTION TO JAVASCRIPT

JavaScript was developed in 10 days in May 1995 by Brendan Eich, then working at Netscape, as the HTML scripting language for their browser Navigator 2. Originally, it was intended to be used for enriching web pages and web user interfaces. A web page could be enriched by (a) generating browser-specific HTML content or CSS styling, (b) inserting dynamic HTML content, or (c) producing special audio-visual effects (animations). A web user interface could be enriched by (a) implementing advanced user interface components, (b) validating user input on the client side, or (c) automatically pre-filling certain form fields.

Later, after browsers started supporting the *ECMAScript 5.1* standard defined in 2011, JavaScript mutated into a full-fledged programming language that could be used for programming all kinds of software applications and tools. At the same time, the back-end JavaScript platform *Node.js* was developed and adopted by an increasing number of web developers who appreciated the new possibility of using JavaScript not only for browser-based front-end development, but also for all kinds of back-end tasks.

Today, JavaScript is the most widely used programming language (according to certain measures, such as the statistics on *StackOverflow* and *GitHub)*, and a powerful platform, which offers many advantages over other platforms:

1. It's the only language that enjoys *native support* in web browsers.
2. It's the only *universal* language in the sense that it allows
   a. building web apps with just one programming language. All other languages (like Java and C#) can only be used on the back-end and need to be combined with front-end JavaScript, so developers need to master at least two programming languages.
   b. executing the same code (e.g., for constraint validation) on the back-end and the front-end.
3. It's the only language that allows *dynamic distribution*, that is, executing the same code (e.g., for business computations) either in the back-end or the front-end, depending on run-time conditions such as the availability of front-end resources.
4. It combines *object-oriented* with *functional* programming.
5. Its dynamism allows various forms of *meta-programming*, which means it enables developers to program their own programming concepts, like classes and enumerations.

JavaScript is *object-oriented* (OO), but in a different way than classical OO programming languages such as Java and C#. There is no explicit *class* concept in JavaScript. Rather, classes have to be defined in the form of special objects, either as *constructor* functions or as *factory* objects, using certain code patterns.

Objects can also be created without instantiating a class, in which case they are *untyped*, and properties as well as methods can be defined for specific objects independently of any class definition. At run time, properties and methods can be added to, or removed from, any object and class.

The version of JavaScript that is currently supported by web browsers is called “ECMAScript 5.1”, or simply “ES5”, but the next two versions, called “ES6” and “ES7”, with lots of added functionality and improved syntaxes, are around the corner and are already partially supported by current browsers and back-end JavaScript environments.

After a brief discussion of the most basic language elements of JavaScript, further concepts are explained on the fly when they are used for implementing a specific simulation. The following sub-sections are extracted from my *JavaScript Summary* article (Wagner 2015), which is recommended for readers who want to learn more about JavaScript.

2.1 Types and Data Literals

JavaScript has only three primitive data types: `string`, `number` and `boolean`, and we can test if a variable `v` holds a value of such a type with the help of `typeof(v)` as, for instance, in `typeof(v) === "number"`. There is no explicit type distinction between integers and floating point numbers (all numeric data values are internally represented in 64-bit floating point format). We can test if a number is an integer with the help of the built-in function `Number.isInteger`.

Since JavaScript has special forms of objects, arrays and functions, I normally say “JS object”, “JS array” or “JS function” whenever I refer to one of them. There are essentially three reference types in JavaScript: `Object`, `Array`, and `Function`. JS arrays and JS functions are special types of JS objects. The object concept of JavaScript is very versatile. For instance, JS objects can be used as *records* (attribute-value pairs), as in `{num:2, denom:3}`, or as *maps* (sets of key-value pairs), as in `{"one":1, "two":2, "three":3}`. A variable holding an *empty JS object literal* is defined by `var o = {}` with curly braces, while a variable holding an *empty JS array literal* is defined by `var a = []` with brackets. Since JS arrays do not have a fixed size, they rather correspond to *array lists*.

The types of variables, array elements, function parameters and return values are not declared and are normally not checked by JavaScript engines. Type conversion (casting) is performed automatically.

For testing the *equality* (or *inequality*) of two primitive data vales, we use the *strict equality* predicate expressed with the triple equality symbol === (and `!==`) instead of the double equality symbol == (and `!=`). Otherwise, for instance, the number 2 would be the same as the string "2", since the condition `(2 == "2")` evaluates to true in JavaScript.

2.2 Procedures and Functions

As in C/C++, procedures are called "functions", no matter if they return a value or not. Since JS functions are JS objects, they can be stored in variables, passed as arguments to functions, returned by functions, have properties and can be changed dynamically. Therefore, JS functions are first-class citizens, and JavaScript can be viewed as a *functional programming* language.

The general form of a *function definition* in JavaScript is an assignment of a function expression to a variable:

```
var myF = function theNameOfMyF () {...}
```

where `theNameOfMyF` is optional. When it is omitted, the function is *anonymous*. In any case, functions are invoked via a variable that references the function. In the above case, this means that the function is invoked with `myF()`, and not with `theNameOfMyF()`.

JS functions can have *inner functions*. The *closure* mechanism allows a JS function using variables from its outer scope, and a function created in a closure remembers the environment in which it was created.

2.3 Defining and Using Classes

The concept of a *class* is fundamental in *object-oriented* programming. Objects *instantiate* (or *are classified by*) a class. A class defines the properties and methods (as a blueprint) for the objects created with it. Having a class concept is essential for being able to implement a *data model* in the form of *model classes* within a *Model-View-Controller (MVC)* architecture.

There has been no explicit class concept in JavaScript before ES6, but only the concept of a constructor function, which can be used in combination with certain code patterns for defining an equivalent of classes. In ES6, a user-friendly syntax for defining classes and class hierarchies has been introduced, but it is still not supported in all browsers today. In ES5, we have to follow certain code patterns, recommended by Mozilla in their JavaScript Guide, for defining a class. The code pattern for defining an inheritance relationship between two classes requires seven steps, see (Wagner 2015). Because such a complex pattern is quite unwieldy, it can be preferable to use a library like cLASSjs for easily defining constructor-based classes and class hierarchies. This is also the approach taken in this tutorial.

2.4 JavaScript's Built-In Random Number Generator

A random number generator (RNG) is a deterministic algorithm that imitates the sampling of a random variable with a uniform distribution over the interval (0,1) by generating a corresponding sequence of floating point numbers upon repeated invocations. The RNGs originally built into JavaScript (more precisely, into the JavaScript engines of browser vendors) were not very good. In fact, at least in Firefox, Chrome and Safari, they have been replaced (in the beginning of 2016) with a high-quality RNG called *xorshift+*, based on Marsaglia's *xorshift* generator, see Vigna (2016).

JavaScript's built-in RNG is invoked with `Math.random()`, which returns a number in the range [0, 1) that is, from 0 (inclusive) up to but not including 1 (exclusive). Unfortunately, the built-in RNG cannot be seeded by the user, so when seeding is required, an external library, such as Mersenne Twister JS or seedrandom, has to be used. Also, since there is no built-in support for random variate generation, a library like jStat or probability-distributions, supporting all common probability distributions, has to be used.

3 MONTE CARLO SIMULATION

We start by building a Monte Carlo simulation, considering the case of manufacturing computer chips on wafers, which are discs that are typically made of purified silicon. Chips are built simultaneously in a grid formation on the wafer surface, as illustrated by Fig 1.

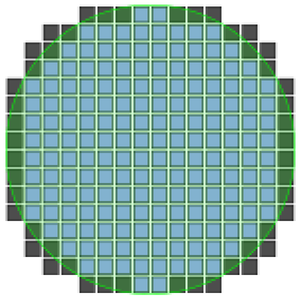

Figure 1: A wafer map showing cells on the wafer and cells that don't fully lie within the wafer (image by Moxfyre, CC BY-SA 3.0).

An important issue is the cost per chip resulting from the chip yield per wafer, which depends on wafer size, chip size and average number of defective chips. In our simplified example, we assume that chips may be defective due to random contamination of the wafer, for which we assume a uniform probability distribution over the wafer's circle area, obtained using polar coordinates.

A simple browser-based JavaScript program consists of an HTML file defining the user interface, and an embedded reference to an external JavaScript file, as shown in the following HTML code listing:

```
<!DOCTYPE html>
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en" lang="en">
<head>
  <meta charset="utf-8">
  <title>Wafer Defects Simulation</title>
  <meta name="viewport" content="width=device-width, initial-scale=1.0" />
</head>
<body>
 <h1>Simulating the Defects on a Wafer</h1>
 <table>
  <thead>
  <tr><th># Defects</th><th>Avg. # Good Chips</th><th>Avg. Chip Price</th></tr>
  </thead>
  <tbody id="tbody"></tbody>
 </table>
 <script src="WaferDefectsSimulation.js"></script>
</body>
</html>
```

In the external JavaScript file "WaferDefectsSimulation.js", we first define three utility procedures:

1. The function *getUniformDecimal* generates a uniformly distributed random variate between a lower bound and an upper bound number.
2. The function *generateRandomPointOnCircle* generates a random point on a circle based on uniformly distributed random polar coordinates.
3. The procedure createOutputTableRow creates a HTML table row in the user interface for showing simulation results to the user.

Then we define a class *Wafer* in the form of a constructor function with two parameters denoting the wafer *diameter* and the *chip size*:

```
var Wafer = function (d, s) {
  var i=0, j=0;
  // define attributes
  this.diameter = d;
  this.chipSize = s;
  this.gridWidthInNumberOfCells = parseInt( this.diameter / this.chipSize);
  this.nmrOfGridCellsOnWafer = 0;  // computed below
  this.gridCells = [];
  // initialize grid cells array
  for (i = 0; i < this.gridWidthInNumberOfCells; i++) {
    this.gridCells[i] = [];  // initialize row array
    for (j = 0; j < this.gridWidthInNumberOfCells; j++) {
      this.gridCells[i][j] = this.isGridCellOnWafer(i, j);
      if (this.gridCells[i][j]) this.nmrOfGridCellsOnWafer++;
    }
  }
};
```

The Wafer class encapsulates three methods:

1. The method *isGridCellOnWafer(i,j)* returns a Boolean value indicating if the grid indexes *i* and *j* identify a cell that fully lies within the wafer or not.
2. The method *generateDefect()* generates a random defect on the wafer.
3. The method *countGoodChips()* counts the good chips on the wafer, after the simulated defects have been generated.

The standard approach in JavaScript is to define the methods of a class as methods of a special "prototype" object associated with the constructor function via its built-in *prototype* property. This allows invoking the methods on any object created with the constructor and enables inheritance for subclasses. Consequently, we define, for instance, the *generateDefect* method in the following way

```
Wafer.prototype.generateDefect = function () {
  var p={}, i=0, j= 0,
      r = this.diameter / 2;
  p = generateRandomPointOnCircle( r);
  i = Math.floor( (p.x + r) / this.chipSize);
  j = Math.floor( (p.y + r) / this.chipSize);
  this.gridCells[i][j] = false;  // defective grid cell
};
```

The repeated random sampling for input variables, which is characteristic for Monte Carlo simulation, requires a corresponding loop, which is run within a nested for loop over chip sizes and defect frequencies in a *main* procedure:

```
(function main () {
  var s=0, k=0, i=0, wafer=null, p=null,
      chipSize = 0.0, nmrOfDefects = 0,
      totalNmrOfGoodChips = 0, avgNmrOfGoodChips = 0.0;
  var nmrOfExperiments = 50;  // # repeated simulation experiments
  var waferDiameter = 12;  // cm
  var waferCost = 5000;  // US Dollars
  // HTML element for tabular output
  var tableEl = document.getElementById("tbody");
  // loop over 3 chip sizes
  for (s=0; s < 3; s=s+1) {
    chipSize = 1 + s * 0.5;  // 1 cm, 1.5 cm, 2 cm
    // create wafer object for computing the # of cells on wafer
    wafer = new Wafer( waferDiameter, chipSize);
```

```
      createOutputTableRow( tableEl, "Chip size: " + chipSize +
          " / Number of potential chips: " + wafer.nmrOfGridCellsOnWafer);
      // loop over different defect frequencies
      for (nmrOfDefects = 10; nmrOfDefects <= 100; nmrOfDefects += 10) {
        totalNmrOfGoodChips = 0;
        avgNmrOfGoodChips = 0.0;
        // repeated experiments for computing averages
        for (i=1; i <= nmrOfExperiments; i=i+1) {
          // create new wafer object
          Wafer = new Wafer( waferDiameter, chipSize);
          // create defects on wafer
          for (k=1; k <= nmrOfDefects; k=k+1) wafer.generateDefect();
          totalNmrOfGoodChips += wafer.countGoodChips();
        }
        avgNmrOfGoodChips = totalNmrOfGoodChips / nmrOfExperiments;
        createOutputTableRow( tableEl, nmrOfDefects, avgNmrOfGoodChips,
            waferCost / avgNmrOfGoodChips);
      }
    }
  }());  // syntax of an immediately invoked function expression
```

Notice that the main procedure is expressed in the syntax of what is called an “immediately invoked function expression”. This is a JavaScript approach for creating a scope for variables that would otherwise pollute the global scope.

The *wafer defects* simulation can be executed, and its full code inspected, by visiting the web page: http://oxygen.informatik.tu-cottbus.de/modsim/ex/WaferDefectsSimulation.html.

4 CONTINUOUS SIMULATION

We consider a scenario where a car has to follow another car and maintain a “safe distance”, assuming that it knows its own velocity and the distance to the leading car, from sensor data. The scenario is illustrated in Fig. 2.

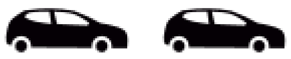

Figure 2: A car following another one (image by Delapouite under CC BY 3.0).

The leading car has its own independent driving behavior model. One or more following cars have a car-following behavior model based on two parameters: the security time distance between two cars, symbolically *sT*, which consists of the time needed to react and a security buffer, and can be set, e.g., to 2 seconds, and a reaction sensitivity coefficient β that defines how quickly a following car adapts its velocity *v* when the current distance *d* falls below, or exceeds, the security distance $v \cdot sT$. For adapting its velocity, a car either accelerates or decelerates, and an equation for computing the car’s acceleration *a* as a control action is the main issue of a car-following behavior model.

We use the simple linear equation $a = \beta\ (d - v \cdot sT)$, which implies a positive value (hence, acceleration), when the current distance is greater than the security distance, and a negative value (hence, deceleration), when it is less that the security distance.

4.1 Making a Simulation Design Model

For designing a general computational solution, we first make a conceptual information model describing the types of objects that populate the considered scenario, and their attributes, which form the system state. Cars are the only object type in the conceptual model shown on the left-hand side in Fig. 3. Cars are described in terms of their make and model, having the physical properties *position*, *velocity* and *acceleration*. These properties are essential for describing how one car, the following car, follows another one, the leading car, by adjusting its velocity via acceleration or deceleration in order to decrease or increase its distance to the leading car.

Notice how the names of both roles, *leading car* and *following car*, are attached to the two ends of the association between cars in the diagram on the left-hand side in Fig. 3. This association models the possibilities that a car may be a following car or a leading car in a sequence of two or more cars.

The general logic of continuous simulation requires iteratively computing the next values of all state variables based on their current values. In a general approach, we can use a two-element array for each state variable v, such that v[0] denotes the current value of v, and v[1] denotes its next value.

We can express this in a design model, as shown on the right-hand side in Fig. 3, by declaring the three attributes *position*, *velocity* and *acceleration* as arrays with two elements, each holding a decimal number (for simplicity, we choose a one-dimensional space model as part of our simulation design). Notice that we have allocated these three attributes, and the two corresponding update functions *computeNextVelocity* and *computeNextPosition*, to a more general class of *moving objects*, of which cars are a special case. In terms of object-oriented modeling, this means that the *Car* class inherits these attributes and functions from the *MovingObject* class.

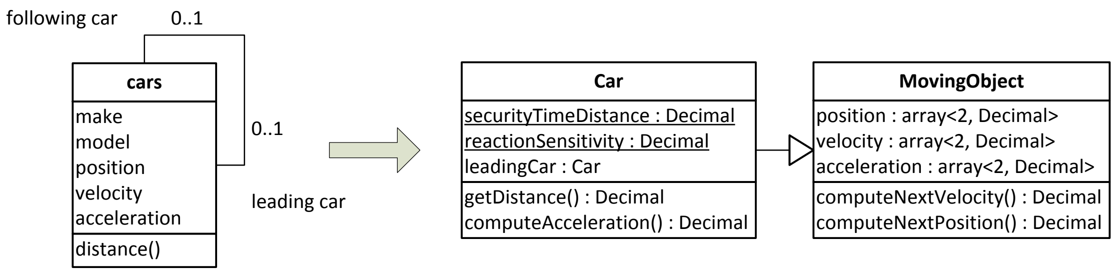
following car
0..1
cars
make
model
position
velocity
acceleration
distance()
0..1
leading car
Car
securityTimeDistance : Decimal
reactionSensitivity : Decimal
leadingCar : Car
getDistance() : Decimal
computeAcceleration() : Decimal
MovingObject
position : array<2, Decimal>
velocity : array<2, Decimal>
acceleration : array<2, Decimal>
computeNextVelocity() : Decimal
computeNextPosition() : Decimal

Figure 3: Deriving a computational design model from a conceptual model.

Notice also that the *computeAcceleration* update function, defining the acceleration behavior of a car, is encapsulated within the *Car* class, while *computeNextVelocity* and *computeNextPosition* simply implement Newtonian physics ($\Delta v = a \cdot \Delta t$ and $\Delta x = v \cdot \Delta t$, respectively) and are encapsulated within the *MovingObject* class.

4.2 Implementing the Simulation Design Model with JavaScript

We first show how to implement the two classes *MovingObject* and *Car* defined in the design model in Fig. 3 with the help of the cLASSjs library, which simplifies the definition of classes in JavaScript. A class is defined as an instance of the meta-class cLASS with a set of properties and a set of methods, where each property is defined by a name and a range, and each method is defined by a name and a method body in the form of a JS function expression (in terms of JavaScript syntax, both `properties` and `methods` are map-valued properties of the `cLASS` instance `MovingObject`).

```
var MovingObject = new cLASS({
  Name:"MovingObject",
  properties: {
    "pos": {range: ["Decimal","Decimal"]},
    "vel": {range: ["Decimal","Decimal"]},
    "acc": {range: ["Decimal","Decimal"]}
  },
  methods: {
    "computeNextVelocity": function () {
       return this.vel[0] + this.acc[0] * dt;
    },
    "computeNextPosition": function () {
       return this.pos[0] + this.vel[0] * dt;
    }
  }
});
```

Notice that *computeNextVelocity* and *computeNextPosition* do not have any parameters, but operate directly on the current acceleration attribute `this.acc[0]`, current velocity attribute `this.vel[0]` and current position attribute `this.pos[0]` of moving objects. The class *Car* is defined as a subclass of *MovingObject* by setting its attribute `supertypeName` to "MovingObject":

```
var Car = new cLASS({
  Name:"Car",
  supertypeName:"MovingObject",
  properties: {
    "leadingCar": {range: Car}
  },
  methods: {
    "getDistance": function () {
        return this.leadingCar.pos[0] - this.pos[0];
    },
    "computeAcceleration": function () {
        return Car.reactionSensitivity *
            (this.getDistance() - this.vel[0] * Car.securityTimeDistance);
    }
  }
});
```

In addition to these two class definitions, the simulation model also includes the following simulation parameter definitions in the form of class-level (static) attributes of *Car*:

```
Car.securityTimeDistance = 2.0;  // in seconds
Car.reactionSensitivity = 5.0;
```

After defining the *simulation model*, the next step is to initialize the simulation by initializing the simulation time variable *t* as well as the fixed time increment parameter *dt* and the simulation end time parameter *endTime*, and define the *initial state* of the simulation by defining initial attribute values for the leading *car1* and the following *car2*:

```
var t = 0.0, dt = 0.1, endTime = 20;  // in seconds
var car1 = new Car({pos:[30,0], vel:[17,0], acc:[5,0]}),
    car2 = new Car({pos:[0,0], vel:[17,0], acc:[0,0], leadingCar: car1});
```

Finally, we implement the ***fixed increment time progression*** loop that iteratively computes the next values of all state variables based on their current values:

```
for (i=0; i <= 200; i=i+1) {
  ...  // output current speed and distance
  // compute next state on basis of current state
  if (car1.vel[0] > 33.33) car1.acc[1] = -5.0;  // 120 km/h
  else if (car1.vel[0] < 16.66) car1.acc[1] = 5.0;
  else car1.acc[1] = car1.acc[0];
  car1.vel[1] = car1.computeNextVelocity();
  car1.pos[1] = car1.computeNextPosition();
  car2.acc[1] = car2.computeAcceleration();
  car2.vel[1] = car2.computeNextVelocity();
  car2.pos[1] = car2.computeNextPosition();
  // update current state
  car1.acc[0] = car1.acc[1]; car1.vel[0] = car1.vel[1];
  car1.pos[0] = car1.pos[1];
  car2.acc[0] = car2.acc[1]; car2.vel[0] = car2.vel[1];
  car2.pos[0] = car2.pos[1];
```

```
    // advance time
    t = t + dt;
  }
```

Notice that we model the acceleration behavior of the leading *car1* by starting with an acceleration of 5 m/s^2 and then periodically changing it to -5 m/s^2 when an upper limit velocity of 33.33 m/s is exceeded and raising it again to 5 m/s^2 when a lower limit of 16.66 m/s is reached.

The *car-following* simulation, extended by a simulation log, can be executed, and its full code inspected, by visiting the web page: http://oxygen.informatik.tu-cottbus.de/modsim/ex/CarFollowing-2cars.html.

5 DISCRETE EVENT SIMULATION

We consider a simple inventory management system: a shop selling one product type (e.g., one model of TVs), only, such that its in-house inventory only consists of items of that type. On each business day, customers come to the shop and place their orders. If the ordered product quantity is in stock, the customer pays for the order and the ordered products are provided. Otherwise, the order may still be partially fulfilled, if there are still some items in stock, else the customer has to leave the shop without any item and the shop has missed a business opportunity. If an order quantity is greater than the current stock level, the difference counts as a lost sale.

The purpose of the simulation project is to estimate lost sales under certain conditions.

5.1 Making a Simulation Design Model

We only model one object type: *SingleProductShop* with the following attributes: *quantityInStock*, *reorderLevel* and *orderUpToLevel*. As a design choice justified by the simulation purpose of estimating lost sales, we can simplify the model by aggregating all individual customer order events on a day in a single *DailyDemand* event.

Replenishment orders are placed whenever the *quantityInStock* falls below the *reorderLevel*. Assuming that each replenishment order causes a corresponding delivery with a random delay (*lead time*) allows abstracting away from replenishment order events and only considering the delivery events caused by them. Thus, we can make a model with only two event types:

1. An exogenous event type *DailyDemand* with a demand *quantity* attribute and a *recurrence* method defining the recurrence of DailyDemand events to be 1 (with the meaning that such an event occurs each day).
2. A caused event type *Delivery* with a *quantity* attribute.

Notice that both event types *DailyDemand* and *Delivery* are associated with *SingleProductShop*, since the shop participates in events of both types. These associations are implemented with the corresponding reference properties *DailyDemand::shop* and *Delivery::receiver*. It is a general *ontological pattern* that ***objects participate in events***. In our example, a shop participates in a *DailyDemand* event, and a shop participates in a *Delivery* event (as the receiver of delivered items). When making a simulation design model, these participations should be explicitly modeled with the help of corresponding associations.

We model the random variations of two variables: *demand quantity* and *delivery lead time*. In general, random (input) variables can be modeled in a UML class diagram as class-level ("static") methods that sample values from the underlying probability distribution. We model the random variable *demand quantity* with the *DailyDemand::sampleQuantity* method using a uniform integer distribution U(5,10), and the random variable *delivery lead time* with the method *Delivery::sampleLeadTime* using an empirical integer distribution Emp{1:0.2, 2:0.5, 3:0.3}. This leads to the information design model shown in Fig. 4.

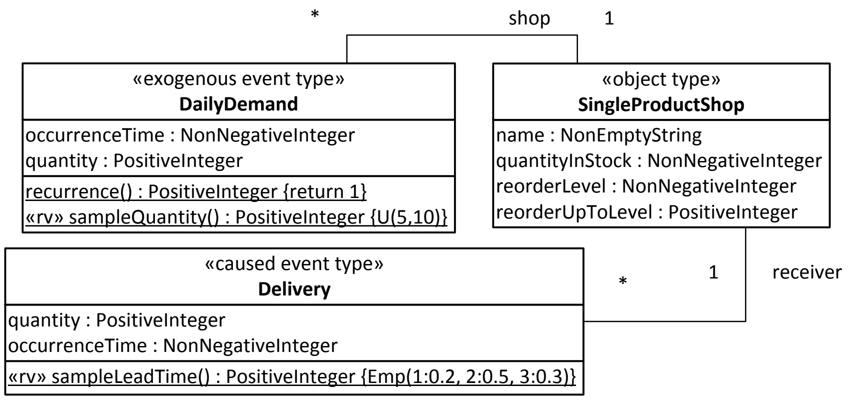
*
shop
1
«exogenous event type»
DailyDemand
occurrenceTime : NonNegativeInteger
quantity : PositiveInteger
recurrence() : PositiveInteger {return 1}
«rv» sampleQuantity() : PositiveInteger {U(5,10)}
«object type»
SingleProductShop
name : NonEmptyString
quantityInStock : NonNegativeInteger
reorderLevel : NonNegativeInteger
reorderUpToLevel : PositiveInteger
«caused event type»
Delivery
quantity : PositiveInteger
occurrenceTime : NonNegativeInteger
«rv» sampleLeadTime() : PositiveInteger {Emp(1:0.2, 2:0.5, 3:0.3)}
*
1
receiver

Figure 4: Information design model for the inventory management simulation.

We use the UML stereotypes «exogenous event type» and «caused event type» for categorizing classes that represent exogenous event types and caused event types, while we use «object type» for categorizing classes that represent object types.

As explained in (Wagner 2014), for obtaining a complete simulation design using the object-event worldview, the information design model of Fig. 4 is complemented with a process design model, e.g., in the form of an event rule design table as in Table 1. Notice that an event rule combines an event type expression with an event routine specifying immediate state changes of affected objects and (possibly delayed) follow-up events.

Table 1: Event rule design table for the inventory management simulation.

ON (event type)	DO (event routine)

DailyDemand(quantity) @ t	IF quantity <= shop.quantityInStock,
	  decrement shop.quantityInStock by quantity
	  IF shop.quantityInStock − quantity < reorderLevel AND
	      shop.quantityInStock > reorderLevel
	    schedule Delivery @ t + sampleLeadTime()
	      with quantity := reorderUpToLevel − quantityInStock
	ELSE (if quantity > shop.quantityInStock)
	  increment shop.lostSales by quantity − shop.quantityInStock
	    and set shop.quantityInStock := 0

Delivery(quantity) @ t	Increment shop.quantityInStock by quantity

Since all event types share an *occurrenceTime* attribute and an *applyRule* method, it is natural to define an abstract type *Event* that encapsulates these features shared by all event types, as shown in Fig. 5 below. This can be implemented as an abstract class with an abstract method *applyRule* in an object-oriented (OO) programming language. In this way, the OO mechanism of *method overriding* can be used in the implementation of the next-event time progression loop for invoking the *applyRule* method on the next event from the future events list. This means that, while in the code of the next-event time progression loop the abstract *Event::applyRule* method is used, when executing the loop, the runtime environment of the OO programming language (in our case, the JavaScript engine) will apply the specific *applyRule* method of the respective event type of the currently processed event (such as *Delivery::applyRule*).

In JavaScript, the *Event* hierarchy shown in Fig. 5 can be simplified by dropping the abstract method from the *Event* class. Due to its weakly typed nature, JavaScript does not need abstract methods for method overriding.

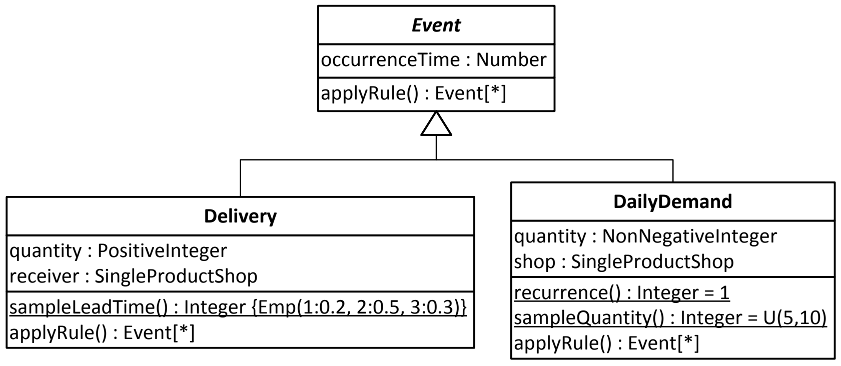
Event
occurrenceTime : Number
applyRule() : Event[*]
Delivery
quantity : PositiveInteger
receiver : SingleProductShop
sampleLeadTime() : Integer {Emp(1:0.2, 2:0.5, 3:0.3)}
applyRule() : Event[*]
DailyDemand
quantity : NonNegativeInteger
shop : SingleProductShop
recurrence() : Integer = 1
sampleQuantity() : Integer = U(5,10)
applyRule() : Event[*]

Figure 5: All event types inherit an *occurrenceTime* attribute from *Event*.

5.2 Implementing the Simulation Design Model with JavaScript

The simulation code has 4 parts:

1. A ***simulation model*** in the form of class definitions; while a continuous simulation model defines only object classes, a DES model also defines event classes.
2. An ***initial state*** definition, which defines a set of initial objects, and, in the case of a DES model, also initial events.
3. A set of ***simulation parameter definitions***, such as the simulation end time.
4. The ***simulation time progression loop***, which is based on next-event time advance in the case of DES, while it is fixed-increment time advance in the case of continuous simulation.

We first show the class definitions for implementing the simulation model: the object class *SingleProductShop* and the event classes *DailyDemand* and *Delivery*.

```
var SingleProductShop = new cLASS({
  Name: "SingleProductShop",
  properties: {
    "name": {range: "NonEmptyString"},
    "quantityInStock": {range:"NonNegativeInteger", label:"Stock"},
    "reorderLevel": {range:"NonNegativeInteger"},
    "reorderUpToLevel": {range:"PositiveInteger"},
    // output statistics
    "lostSales": {range:"NonNegativeInteger", label:"Lost"}
  }
});
```

Notice that, for simplicity, we have defined the lost sales statistics as an attribute of the object class *SingleProductShop*. In general, we would rather have special data structures and code for handling the output statistics variables.

Since event rules are implemented as methods with the pre-defined name *applyRule* in the event class concerned, we obtain the following code for the *DailyDemand* event class:

```
var DailyDemand = new cLASS({
  Name: "DailyDemand",
  supertypeName: "Event",
  properties: {
    "quantity": {range: "PositiveInteger", label:"quant"},
    "shop": {range: SingleProductShop}
  },
  methods: {
    "applyRule": function () {...}   // see below
  }
});
```

The *applyRule* method takes care of all state changes implied by a *DailyDemand* event, and of scheduling *Delivery* follow-up events. It is defined as follows:

```
"applyRule": function () {
  var q = this.quantity,
      prevStockLevel = this.shop.quantityInStock,
      followUpEvents = [];
  // update lostSales if demand quantity greater than stock level
  if (q > prevStockLevel) {
    this.shop.lostSales += q - prevStockLevel;
  }
  // update quantityInStock
  this.shop.quantityInStock = Math.max( prevStockLevel - q, 0);
  // schedule Delivery if stock level falls below reorder level
  if (prevStockLevel > this.shop.reorderLevel &&
      prevStockLevel - q <= this.shop.reorderLevel) {
    followUpEvents.push( new Delivery({
      occTime: this.occTime + Delivery.sampleLeadTime(),
      quantity: this.shop.reorderUpToLevel - this.shop.quantityInStock,
      receiver: this.shop
    }));
  }
  // since DailyDemand is exogeneous, schedule next DailyDemand event
  followUpEvents.push( new DailyDemand({
    occTime: this.occTime + DailyDemand.recurrence(),
    quantity: DailyDemand.sampleQuantity(),
    shop: this.shop
  }));
  return followUpEvents;
}
```

In addition to the property definitions and the *applyRule* method definition, we also have the definitions of the static *recurrence* method and the static method *sampleQuantity* implementing the random variable *demand quantity*:

```
DailyDemand.recurrence = function () {
  return 1;
};
DailyDemand.sampleQuantity = function () {
  return getUniformInteger( 5, 10);
};
```

The second event class, *Delivery*, has the following code:

```
var Delivery = new cLASS({
  Name: "Delivery",
  supertypeName: "Event",
  properties: {
    "quantity": {range: "PositiveInteger", label:"quant"},
    "receiver": {range: SingleProductShop}
  },
  methods: {
    "applyRule": function () {
      this.receiver.quantityInStock += this.quantity;
      return [];  // no follow-up events
    }
  }
});
```

In addition, the random variable *lead time* is implemented with a static method *sampleLeadTime*:

```
Delivery.sampleLeadTime = function () {
  var r = getRandomInt( 0, 99);
  if (r < 25) return 1;         // probability 0.25
```

```
    else if (r < 85) return 2;     // probability 0.60
    else return 3;                 // probability 0.15
  };
```

The initial state is defined in terms of initial objects and initial events. In our simple example, there is only one object, which represents a shop:

```
  var tvShop = new SingleProductShop({
    name:"TV",
    quantityInStock: 80,
    reorderLevel: 50,
    reorderUpToLevel: 100
  });
```

In addition, the initial future events list *FEL* is set up with just one initial *DailyDemand* event as follows:

```
  var FEL = new EventList();
  FEL.add( new DailyDemand({occTime:1, quantity:25, shop: tvShop}));
```

The future events list drives the ***next-event time progression loop***, the code of which is as follows:

```
  while (simTime < simEnd) {
    logSimulationStep( tvShop, FEL);
    nextTime = FEL.getNextOccurrenceTime();
    // extract and process next events
    nextEvents = FEL.removeNextEvents();
    for (i=0; i < nextEvents.length; i++) {
      e = nextEvents[i];
      // apply event rule
      followUpEvents = e.applyRule();
      // schedule follow-up events
      for (j=0; j < followUpEvents.length; j++) {
        FEL.add( followUpEvents[j]);
      };
    };
    simTime = nextTime;  // advance simulation time
  }
```

5.3 Implementing an Event List with JavaScript

Any DES algorithm based on *future events scheduling* needs a suitable complex datatype for the future events list. As we can see in the program code listing of the next-event time progression loop above, this complex datatype has to provide three operations:

1. A *getNextOccurrenceTime* function for retrieving the occurrence time of the next event.
2. A *removeNextEvents* method for retrieving and removing all events with an occurrence time equal to getNextOccurrenceTime() from the FEL.
3. An *add* method for adding an event to the FEL.

As in other programming languages, we have different options how to implement the complex datatype *EventList*. The simple implementation chosen in our Inventory Management simulation is based on using an unordered array as a store for the future events. While this choice is okay for all simulations where the future events list does not grow larger than a few hundred events, a better choice would be using a linked list. Notice that linked lists are not provided as a pre-defined class in JavaScript. However, they can be easily implemented.

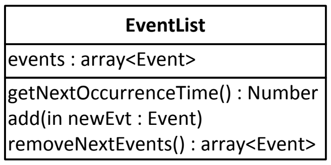
EventList
events : array<Event>
getNextOccurrenceTime() : Number
add(in newEvt : Event)
removeNextEvents() : array<Event>

Figure 5: An *EventList* class using an unordered JS array as the events store.

Visit the web page http://oxygen.informatik.tu-cottbus.de/modsim/ex/InventoryManagement.html for executing the Inventory Management simulation and for inspecting its code.

6 CONCLUSIONS

We have shown how different types of simulations can be implemented with JavaScript. However, for a lack of space, we were not able to show how to construct a user interface for our simulations. The web platform offers excellent technologies for building user interfaces with HTML and CSS, and enriching them with powerful visualizations using SVG and the Canvas API. For learning more about how to build a web-browser-based simulator supporting advanced concepts, user interfaces and visualization, using JavaScript, HTML, CSS and SVG, see my book Introduction to Simulation Using JavaScript (Wagner 2016), from which the material of this tutorial has been extracted.

REFERENCES

Guizzardi, G., and G. Wagner. 2010. “Towards an Ontological Foundation of Discrete Event Simulation.” In: Johansson B, Jain S, Montoya-Torres J, Hugan J, Yücesan E (Eds.), Proceedings of the 2010 Winter Simulation Conference, Baltimore (MD), USA, 652−664. Piscataway, New Jersey: Institute of Electrical and Electronics Engineers, Inc. Available from: http://www.informs-sim.org/wsc10papers/059.pdf

Pegden, C.D. 2010. “Advanced Tutorial: Overview of Simulation World Views.” In: Johansson B, Jain S, Montoya-Torres J, Hugan J, Yücesan E (Eds.), Proceedings of Winter Simulation Conference, Baltimore (MD), USA, 643−651. Piscataway, New Jersey: Institute of Electrical and Electronics Engineers, Inc.

Vigna, S. 2016. “PRNG Shootout.” Accessed April 29, 2016. http://xorshift.di.unimi.it/

Wagner, G. 2014. “Tutorial: Information and Process Modeling for Simulation.” In *Proceedings of the 2014 Winter Simulation Conference*, edited by A. Tolk, S. Y. Diallo, I. O. Ryzhov, L. Yilmaz, S. Buckley and J. A. Miller, 103–117. Piscataway, New Jersey: Institute of Electrical and Electronics Engineers, Inc. Available from: http://informs-sim.org/wsc14papers/includes/files/012.pdf

Wagner, G. 2015. “JavaScript Summary.” Accessed April 28, 2016. http://web-engineering.info/SummariesCheatsheetsPosters.

Wagner, G. 2016. “Introduction to Simulation Using JavaScript.” Accessed July 1, 2016. http://web-engineering.info/sim

AUTHOR BIOGRAPHIES

GERD WAGNER is Professor of Internet Technology at the Dep. of Informatics, Brandenburg University of Technology, Germany, and Adjunct Associate Professor at the Dep. of Modeling, Simulation and Visualization Engineering, Old Dominion University, Norfolk, VA, USA. His research interests include modeling and simulation, foundational ontologies, knowledge representation and web engineering. His email address is G.Wagner@b-tu.de.