CLIENT SIDE SIMULATION TOOL JSSim

JAROSLAV SKLENAR

Department of Statistics and Operations Research
University of Malta
Msida MSD 06, Malta
Web: http://staff.um.edu.mt/jskl1/
E-mail: jaroslav.sklenar@um.edu.mt

Abstract: JavaScript is an interpreted language where the important techniques of Object Oriented Programming can be utilized. Some of them are not included directly, so they need additional support. For example inheritance has to be programmed explicitly. A JavaScript programmer is thus making use of a modern language that, together with HTML, supports creation of documents that can contain user-friendly input of validated data, any kind of data processing, and lucid presentation of results. Solutions based on JavaScript and HTML are typically placed on the web and made thus available literally to everybody who has a browser supporting particular versions of these two languages. These capabilities have been applied to create various web-hosted problem-solving tools. Such tools can contain simple and medium-scale simulation models. Several simulation models have already been implemented and placed on the web with very encouraging response. Routines used to create these models, including a simple event-oriented simulation engine together with a collection of classes for general use in discrete simulation, have been collected into a tool that we call JSSim. The paper describes the capabilities of this tool by using examples oriented to simulation of queueing systems. The tool also supports direct links between JavaScript objects and parts of the corresponding HTML documents in order to simplify programming as much as possible. A queueing network has been simulated to compare JSSim with Extend™ and Arena™ from several points of view.

keywords: Web based simulation, JavaScript, Discrete event simulation, Queueing models.

1 INTRODUCTION

The book [Flanagan, 1998] describes the JavaScript prototype oriented paradigm. The papers [Sklenar, 2001, 2002] explain how to use this paradigm in order to be able to utilize all important techniques of Object Oriented Programming (OOP) in JavaScript. Some new techniques not available in strongly typed compiled Object Oriented Languages (OOL) are also introduced. In particular the programmed inheritance described in the paper [Sklenar, 2001a] enables creation of "subclasses" that inherit only selected methods of the superclass. Thus we can create simplified versions of general superclasses. All these techniques can be used to create a reusable code open to future expansion and modification. In other words in an interpreted JavaScript environment we can use the techniques typical for classical compiled strongly typed OOLs like for example Simula or Java together with the flexibility and simplicity typical for interpreted languages with loose typing. All this of course can be done at the expense of security, but as JavaScript is not intended as a language for large software projects, it is not considered as a big problem. The paper [Sklenar, 2001b] deals with the implementation of a simulation engine that was written entirely in JavaScript and that together with appropriate HTML documents supports user-friendly development of web hosted tools that contain simple and medium scale simulation models. The engine is based on the classical event-oriented approach with two primitives: *schedule an event at a certain time* and *cancel a scheduled event*. These primitives are implemented as calls to routines with appropriate parameters. Other simulation supporting facilities are also available, for example generation of random numbers, working with queues, and transparent collection and computation of statistics. All these facilities have now been collected into a tool called JSSim (JavaScript Simulation). The purpose of this paper is to describe the capabilities of this tool. Though JSSim is a general tool for event-oriented discrete simulation, examples oriented to simulation of queueing systems will be used.

2 SIMULATION FACILITIES OF JSSim

Facilities found in languages and tools for programming discrete simulation models can be classified into the following main groups:

- *Time control, synchronization and communication of processes*
- *Generation of random numbers*
- *Transparent collection of statistical data*
- *Statistical analysis*
- *Advanced data structures*
- *User-friendly Input and Output*

 ISBN 3-936150-25-7

Next chapters will summarize the implementation of these facilities in JSSim.

2.1 Time Control

For time control we consider only the two commonly used approaches. While the process-oriented discrete simulation represents the most advanced way of modeling the dynamics of complex systems, the classical event-oriented approach is simpler and easier to learn and to implement. That's why it has been chosen for a JavaScript based tool that is not intended for large simulation studies. Assuming that the reader is familiar with the event-oriented principle, these are the basic facts: During (re)loading of the document the engine creates two global variables: the *time* and the empty *sequencing set* (SQS). Events are represented by *event notices* created by the user and stored in the SQS. Each event notice has the occurence time of the event and any other user-defined data. The engine assigns the time when the event is scheduled. From the user's point of view, the SQS is a list of event notices ordered by the time of occurence in increasing way. After activation, the engine repeatedly removes the first event notice from SQS, updates the model time, and activates a user routine that is given the reference to the event notice. Simulation ends by the empty SQS or by any user supplied condition. These are the engine routines that are called from the user's part of the simulation model:

`initialize_run()` is a routine that clears the SQS (the previous experiment may have finished with nonempty SQS) and sets the model time to zero. It should be called at the beginning of the model initialization.

`evnotice()` is the event notice constructor. It returns an object with the time property, that is used later by the engine and should not be accessed by the user. The user can add any other properties to distinguish between types of events and to store other model dependent data.

`schedule(event,t)` schedules the event whose notice is the first parameter at the time given by the second parameter.

`modeltime()` is the current time of the model. So scheduling an event `e` after a delay `d` is performed as follows:
`schedule(e,modeltime()+d).`

`cancel(event)` cancels a scheduled event. The function returns a boolean value that reports whether canceling was successful.

`simulation_run(stats,length)` starts the simulation experiment. This routine should be called after the model initialization that has to schedule at least the first event. The two parameters just affect the progress reporting in the status bar. The routine ends by reaching the empty SQS or by the user supplied terminating condition - the user's routine `finish_run()`.

The above routines are common to all simulation models. Model specific behavior is implemented by two routines that have to be supplied together with the code (preferably also a routine) that starts the simulation. These are the routines (together with examples) that represent the user's part of the simulation control:

`finish_run()` tests whether simulation should be terminated. It is called by the engine after updating the model time just before activating the next user event. It can just test the time against the experiment duration or it can implement a more complicated terminating condition, like for example serving a given number of customers. The following is the function of a model where the experiment is finished by reaching its duration `runlength`:

```
function finish_run() {
  return (modeltime()>runlength)
};
```

`eventroutine(event)` is activated by the engine. The routine is given the reference to the event notice that has been removed from the SQS. The rest is the user's responsibility. Typically there will be some properties created by the user used to switch between various types of events. It might be a good idea to keep this routine short and simple and to write routines for various types of events similarly as they are written in event oriented simulation languages. The following is the function of a model with two types of events:

```
function eventroutine(event) {
 // The event routine switches
 // between types of events
  switch (event.eventtype) {
  case 1: next_arrival(); break;
  case 2:
   end_of_service(event.servnum);
   break;
  default:alert(
      "Wrong eventtype: "
      + event.eventtype);};
};
```

The start of simulation has also to be programmed. For example it can be a function activated by pressing a button "Run". This function is supposed to perform the following activities in this order:

- Initialization of the engine by `initialize_run()`
- Model specific initialization
- Starting simulation by `simulation_run()`
- Model specific experiment evaluation.

The following is an example of a function activated by pressing the button "Run" and its link to HTML. Some model specific tests have been removed.

```
<INPUT TYPE="button" VALUE="Run"
 onClick="simulation()">

function simulation() {
   // Tests whether simulation can
   // start (not shown here)
 initialize_run();
   // This prepares the engine
 initialization();
   // Initiates model & statistics
 var ev = new evnotice();
   // Scheduling the first arrival
 ev.eventtype = 1;
   // User defined property
 var x = arrival.generate();
   // Generation of first interval
 intstat.update(x);
   // Interval statistics update
 schedule(ev,modeltime() + x);
   // Scheduling the first event
 simulation_run(showstatus,
                runlength);
   // This starts the experiment
 evaluation();
   // Experiment evaluation };
```

2.2 Generation of random numbers

JSSim contains a rather complex class used to generate instances (objects) that represent random numbers. These can have either a theoretical distribution (so far only few are available), but primarily they are supposed to contain tables used to generate values with a general (for example experimentally obtained) distribution. Methods are available for entering and editing such tables. Working with empirical tables is user-friendly; table entries can be modified, inserted and deleted. Large tables can be saved and loaded (provided cookies are enabled in the browser). Figure 1 shows a table created by HTML used to enter parameters of a random variable. The controls are self-explaining. Figure 1 shows the situation just before confirmation of an empirical CDF table by pressing the button "Check & Confirm". Inversion is used for generation that can be either discrete or interpolated. The technique of restricted inheritance (simplification) mentioned earlier was used do declare a simplified version of this class for generation of discrete random numbers with empirical distribution only. Its instances have been used for example to represent random movements of customers in queueing networks. Work with random numbers is very simple. Instances are first created by statements similar to the following one located typically in the so-called head code that is interpreted during loading of the document:

```
var arrival=new Distribution("a1");
```

The method `generate()` returns the random values, so during simulation statements similar to the next one are used:

```
var x = arrival.generate();
```

So far the standard JavaScript random generator `Math.random()` is used.

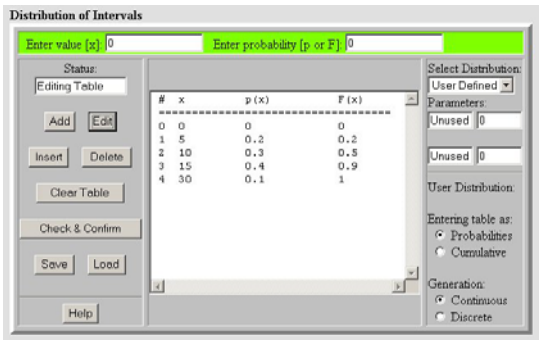


Figure 1: Entering parameters of a random variable

2.3 Transparent statistics

Transparent collection of statistical data and simple statistical analysis are implemented by the classes *Accumulator* and *Tally* (Simscript II^{TM} terminology is used). They differ in time treatment. *Tally* ignores the time; the statistics is based on the collection of the assigned values only. *Accumulator* statistics is based on time integrals. Basically they are both real variables with transparent collection of statistics. The consequence for the user is the difference in the form of the assignment statement. The usual `a = x` has to be replaced by a method call `a.updateto(x)`. Simscript II^{TM} calls this mechanism *left monitoring*. It is based on the idea suggested by [McNeley, 1968] who used the name *Store Association*. Both *Tally* and *Accumulator* objects have methods that return standard statistical figures like `a.average()` that are used during experiment evaluation without any further programming.

2.4 Advanced data structures

To implement the sequencing set that is conceptually an ordered list of event notices the *heap* class has

been implemented. Heap (not to be mixed with the dynamically allocated memory of some languages) is a perfectly balanced binary tree stored in an array with the following properties assuming ascending ordering of items by a certain key:

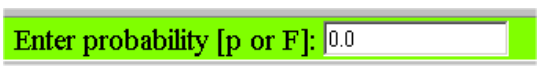
- The root with minimum key is at the position 1
- The two children (if any) of a node at the position i are at the positions $2i$ and $2i+1$
- Both children have bigger (or equal) key than the parent.

Heap supports two basic operations *adding an item* and *removing the first item*. These operations have the performance of $O(\log_2 n)$ where n is the number of items in the heap. Heap is also intended to be used as a priority queue. For more details see [Sklenar, 2001b].
JSSim also contains classes that implement a linked list and a generic statistically monitored queue without any specific ordering. Using these two classes as superclasses, programmed multiple inheritance was used to define classes for FIFO and LIFO queues. Due to the superclasses used in multiple inheritance, the queues can have practically unlimited length and methods are available that return typical statistical results like average, standard deviation, and maximum of the queue length. Methods that perform the basic operations have the same name. Loosly typed JavaScript is polymorphic, so the same code is used to work with various types of queues.

2.5 User-friendly Input and Output

Validated input is easily implemented by JavaScript code associated with text areas in the HTML document. JSSim contains various validation routines to check for example that the user has entered a syntactically correct non-negative number. The technique is known to everyone who has filled in any on-line form. In addition to validation it is also possible to update model parameters accordingly. This can simplify model initialization when simulation is started. The following HTML fragment together with the associated page contents represents a validated input of a probability value. The routine checks non-negativity and whether the value is not bigger than 1. Note that 0 is restored in case of wrong input.

```
Enter probability [p or F]:
<INPUT TYPE="text"  NAME="GIpx"
SIZE=15  VALUE="0.0" ONCHANGE="if
   (!testNonnegLE1Value(GIpx.value))
   {GIpx.value = 0}">
```



Model parameters can be updated directly when the user enters the values or alternatively it is possible to link objects to HTML text fields and to write methods that read the validated data before simulation starts. This direct link has so far been utilized for outputs. The idea is as follows. The link is done by common names. So assume that a queue instance has been created by calling its constructor:

```
queue = new FifoQueue("Q1");
```

The constructor creates and initializes the queue, the name is stored to the property `qname`. The instance has two output methods inherited from statistically monitored queue. The first method is used to update the contents of the host HTML document:

```
StatQueue.prototype.scrupdate =
function(dname) { with (this) {
  eval(dname + qname +
       "av.value = average()");
  eval(dname + qname +
       "ma.value = maxqlength");
  eval(dname + qname +
       "sd.value = stdDev()");
}};
```

Note that the method `scrupdate()` updates three text fields (typically in a table with results) that would contain the average length, the maximum length, and the standard deviation of the length of the queue. Assume that the method is called as follows:

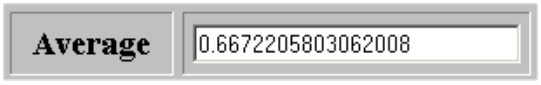
```
queue.scrupdate("document.form1.");
```

So for the average and with respect to the above example the procedure `eval` is given and evaluates the parameter:

```
document.form1.Q1av.value=average()
```

This updates the text field called `Q1av` on the screen. The following is the HTML fragment together with the associated page contents:

```
<TH> Average </TH>
<TD><INPUT TYPE="text" NAME="Q1av"
     SIZE=25></TD>
```



So far it is the user's responsibility to keep the compatibility of names. Here it is the name of the JavaScript object `Q1` that is linked to the HTML text field called `Q1av`. This can be achieved by using standard HTML templates processed by the "Replace All" operation available in practically all

text editors. In this case a template displaying typical queue statistics would be used. Another method `winupdate()` generates an HTML fragment that displays four lines with results:

```
StatQueue.prototype.winupdate =
function(stitle,w) { with (this) {
 w.writeln(stitle +
    " length statistics:" +
    "<BR><UL>");
 w.writeln("<LI> Average: " +
   average());
 w.writeln("<LI> Maximum: " +
   maxqlength);
 w.writeln("<LI> Std Dev: " +
   stdDev() + "</UL>");
}};
```

The method `winupdate()` is used for generation of results in textual format in a separate window. Assuming that there is an open window `resw` the method is activated as follows:

```
var d = resw.document; ...
queue.winupdate("Queue",d);
```

The generated output can then be copied and pasted into other documents as it has been done here:

Queue length statistics:

- Average: 0.6672205803062008
- Maximum: 10
- Std Dev: 1.4089986068350546

3 EXAMPLE SIMULATION

There are several simulation models created by using JSSim that are available on the web. One of them is a general simulator of queuing networks whose last version is available at: *http://staff.um.edu.mt/jskl1/simweb/net2/netmain.html*. This model has been used to compare capabilities of JSSim with two professional simulators that both can be characterized as Visual Interactive Modelling Systems (VIMS) [Pidd, 1998]. Academic version of Arena™ (Rockwell Software Inc.) is distributed with the book [Kelton et al., 2002]. It is a general discrete simulation tool oriented to simulation of queueing systems. Extend™ (Imagine That Inc.) is a general tool for both continuous and discrete simulation. Its demo version can be downloaded from *http://www.imaginethatinc.com/*.

The simulated system is a network made of two generators of customers and four network stations. Figure 2 made of self-explaining blocks is a network created by Extend, Arena chart is similar. The network works as follows: after generation the customers enter randomly any of the four service stations, all with the same probability. After being served the customers either leave the network or move to any of the four stations, all five options have the same probability.

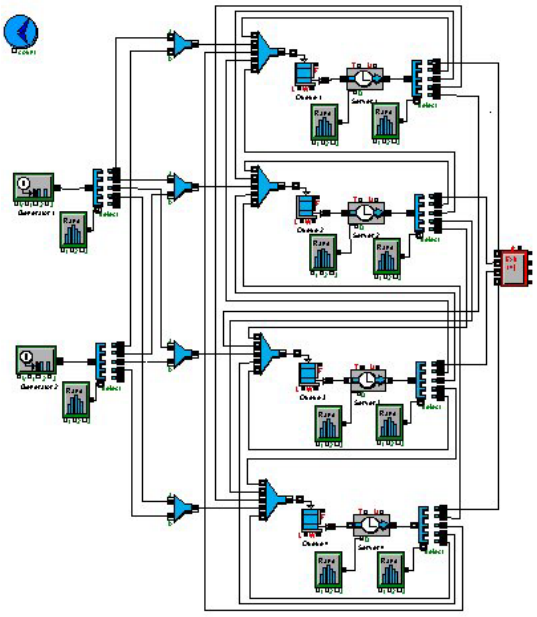


Figure 2: Example queueing network in Extend™

The two generators are for simplicity equal with exponential distribution of intervals with the mean value 10 time units (let's assume minutes). The service stations are also equal, all made of one FIFO queue of unlimited capacity and one server with exponential distribution with the mean value 2 minutes. All these assumptions can be easily modified. Having these input data, the first task was to create the models. In Arena and Extend it means drawing the networks and entering parameters of blocks. Describing all details is out of the scope of this paper, but the work is easy and can be done with just the basic training. In JSSim simulator there is no drawing. The distributions are entered in tables like the one in Figure 1. Server parameters are also entered in tables, this model uses defaults, so actually nothing is entered. The first criterion is the ease of creating the model. While drawing a network is certainly a good way for beginners and for education, it is a real nuisance in case of routine work with non-trivial models. Creating the model in JSSim simulator by entering data into tables was much faster than drawing a network like the one in Figure 2. Moreover equal distributions are entered only once, then saved to cookies, and loaded for other blocks – see the "Save" and "Load" buttons in Figure 1. JSSim simulator and Arena provide all typical results for the generators, the queues, the servers and the whole network as such, in particular the average time spent by a custonmer in the network. In Extend measuring this time has to be incorporated in the model, that is not included in Figure 2. The following text has been copied from a

report generated by the JSSim simulator in a separate window. These are system results for one particular 60000 minutes long experiment rounded to 3 decimal places.

Number of arrivals : 11787
Number of lost customers : 0 (0%)
Number of departures : 11778
Average time in network : 19.855
Network time standard deviation : 21.367
Minimum time in network : 0.001
Maximum time in network : 234.360

The following are the results for the first service station with the CDF table removed:

Server # 1

Exponential service duration, Mean=2
Routing of departures: (removed)
Number of channels : 1
Unlimited queue, FIFO organization
Number of arrivals : 14809
Number of not waiting arrivals : 5948 (40.16%)
Number of lost customers : 0 (0%)
Number of services : 14808
Average service duration : 1.997
Minimum service duration : 0.00007
Maximum service duration : 19.646
Average waiting time : 1.902
Waiting time standard deviation : 3.352
Average non zero waiting time : 3.178
Maximum waiting time : 35.340
Average time in server : 3.899
Time in server standard deviation : 3.890
Minimum time in server : 0.00007
Maximum time in server : 38.558
Average queue length : 0.469
Queue length standard deviation : 1.065
Maximum queue length : 13
Utilization of server(s) : 0.493

Results from Extend and Arena models are very similar with variations given by different sequences of random numbers. The second and the most important criterion is the speed. Table 1 shows the typical duration of an experiment of the length 60000 minutes for various simulators on the same computer (PII, 350MHz, 128MB, Windows/Me) in single task mode. JSSim engine measures the time exactly, duration for the other two was measured by stop watches. There is some variation, figures in Table 1 are averages taken from several runs. Though the precision of the measurements is not very high, for the comparison they are sufficient. It is no surprise that JSSim's interpreted code is slower than the other two compiled and optimized simulators. The speed of the JSSim simulator in Internet Explorer 6 is in fact a pleasant surprise. The results show clearly that using JSSim it is possible to create at least medium size models that run fast enough to enable long experiments or repetitions to reduce the variance of the results.

Table 1: Speed of queueing networks simulators

Simulator	Duration [s]
Extend 4.1	32
Arena 5.0	55
JSSim (IE 6.0)	78
JSSim (NC 4.75)	135
JSSim (NC 7.02)	340

CONCLUSION

JSSim is a result of experimentation with concrete models. Its facilities were added gradually according to concrete problems that had to be solved. In addition to facilities listed so far there are also various utilities like displaying a help window, work with cookies, etc. Its next development will be oriented to enhancement of random numbers and especially to the definition of more complex standard classes like complete multichannel servers. The simulators implemented so far are used mainly for education, but they generated also interest from professional organizations. Simulators are freely available for direct use and for download at *http://staff.um.edu.mt/jskl1/simweb/*.

REFERENCES

Darnell R. et al. 1998, "HTML 4 Unleashed. Professional Reference Edition", Sams.net Publishing.

Eckel B. 1998, "Thinking in Java", Prentice Hall. Inc.

Flanagan D. 1998, "JavaScript - The Definitive Guide", O'Reilly & Associates, Inc.

Kelton W.D. et al. 2002, "Simulation with Arena", McGraw-Hill.

McNeley J.L. 1968, "Compound Declarations". In: *Proceedings of IFIP Working Conference on Simulation Languages*, Oslo, May 1967. North Holland, p.292-303.

Pidd J. 1998, "Computer Simulation in Management Science", John Wiley & Sons.

Sklenar J. 2001, "Interactive Simulators in JavaScript". In: *Proceedings of 15th European Simulation Multiconference ESM2001*, Prague, p.247-254.

Sklenar J. 2001, "Client Side Web Simulation Engine". In: *Proceedings of 27th ASU Conference Model Oriented Programming and Simulation*, Rättvik, Sweden, p.1-13.

Sklenar J. 2002, "Discrete Event Simulation in JavaScript". In: *Proceedings of 28th ASU Conference: The Simulation Languages*, Brno, Czech Republic, p.115-121.