36th European Modeling & Simulation Symposium
21th International Multidisciplinary Modeling & Simulation Multiconference

2724-0029 © 2024 Thomas Wiedemann and Michael Danzig
doi: 0009-0004-1994-6901

SimLuxJS – an optimized JavaScript-simulator

Thomas Wiedemann[1*] and Michael Danzig[1]

[1] University of Applied Science Dresden, F.-List-Platz 1, 01069 Dresden, Germany

*Corresponding author. Email address: wiedem@informatik.htw-dresden.de

Abstract

Today, JavaScript is very successful used in all environments – on web servers, in web-clients for building desktop like applications and in multi-core applications. Older simulation systems with JavaScript use a very complicated code structure ´, often called "callback-hell" and the first attempts of JavaScript based simulation were not very successful. In result of that situation, the authors did a restart with new JavaScript methods like promises and async/await-structures in the actual JavaScript versions The result is a JavaScript simulation system with an improved and better readable syntax. It is also very similar to the well known SimPy and SLX syntax, which is often used in simulation. So SimPy and SLX users can use SimLuxJS as a new option for discrete simulation without large migration efforts.

Keywords: JavaScript-based simulator, with improved syntax, SimPy, SLX

1. Introduction

With the improved V8 JavaScript-engine from Google from 2010 (Daniel, 2012), JavaScript (JS) made a very successful development. Now it is one of the most often used programing languages in the world. Nearly all web applications are built by using large and complex JavaScript frameworks like react or VueJS. In the last years JavaScript was extended with new syntax elements like async and await for better writing concurrent programming tasks. This paper extends an older JavaScript simulation approach by using these new syntax elements. The result is a JavaScript simulation system with an improved and better readable syntax. It is also very similar to the well known SimPy and SLX syntax. So SimPy- and SLX-users can use SimLuxJS as a new option for discrete simulation without large migration efforts.

2. State of the art of JS-simulation systems

A first JavaScript based simulation system was already presented by the first author at the EMSS conference in 2016 (Wiedemann, 2016).

The main structure of the simulation code consists of nested callbacks (see fig. 1). Especially in complex and large web-based applications such call back structures are hard to write and maintain. Often, this code is called "callback hell" and programmers try to avoid such code structures.

```
// simulation with nested operation steps
var simdis = require('./simdis');//sim modul
// advance to start time of process
simdis.advance( simStartTime,
  function()  // callback to next step
  {  // do the job
     simdis.waituntil(  simCond2(),
        function(err) //  next callback
        { console.log(simprocID + ":Step3");
                          // and so on …
}  }); }); }); });  // end of nested function calls
```

Figure 1: A typical call-back structure in old JavaScript

A similar approach was presented by Varshney with the SimJS-system (Varshney, 2011). Unfortunately, this system was not maintained further during the last five years. No other, real usable JS-discrete simulation-systems were found in the web with a normal web search by using Google.

Some other existing JS-simulators are also based on forks of the SimJS-project and must also be considered as inactive. All the projects used the old and ineffective code syntax. In result of this situation, the first author considered a restart with the state of JavaScript in 2023 in a master course on "Discrete simulation". The presented project is based on the final semester project of on master student in 2023. It could be seen also as a M&S teaching example.

3. New JavaScript-syntax options after 2017

The mentioned "callback-hell" was also acknowledged by the JavaScript core developers, and they offer now new methods like promises and async/await-structures in the JavaScript versions (ECMA, 2015) and (ECMA, 2017, Brandt, 2019).

```
// the new async / await-syntax
function resolveAfter2Seconds() {
  return new Promise((resolve) => {
    setTimeout(() => { resolve('resolved');
    }, 2000);   });
}
async function asyncCall() {
  const result = await resolveAfter2Seconds();
  console.log(result);
}
asyncCall(); // execute it
```

Figure 2: The new async/await syntax (MSDN24)

The new syntax is a combination of a call with the new option **await** inside a function, which is declared with the new prefix async as asynchronous. This syntax is more readable und avoids the nesting of callbacks. The code structure is less complex and allows a separation of the whole simulation code into a fixed simulation kernel and an application specific model description without complex algorithms.

This approach supports the well-known design scheme "separation of concern" (see Gudabayev T. (2021)). In the case of simulation models, it allows a very strong separation of the simulation core and the simulation model.

4. An optimized architecture and syntax for JavaScript based simulation

4.1. Main program structure of SimLuxJS

Like already mentioned, the **main advantage of the new approach** in JavaScript for asynchronous programing with async/await is the **separation of the asynchronous program parts of the simulation kernel and the application specific simulation model**. In result, a better structure of the whole simulation program is possible (see fig. 3)

The SimLuxJS -Library concentrates all necessary internal discrete simulation algorithms (for details and backgrounds of discrete simulators please see (Schriber, 2013)). **In general, the source code of the SimLuxJS -Library should not be changed by standard modelers** to avoid an inconsistent behavior of the simulation system. Otherwise, the whole system is available as an Open-source project at (GitHub, 2024) and all simulation experts are invited in collaboration.

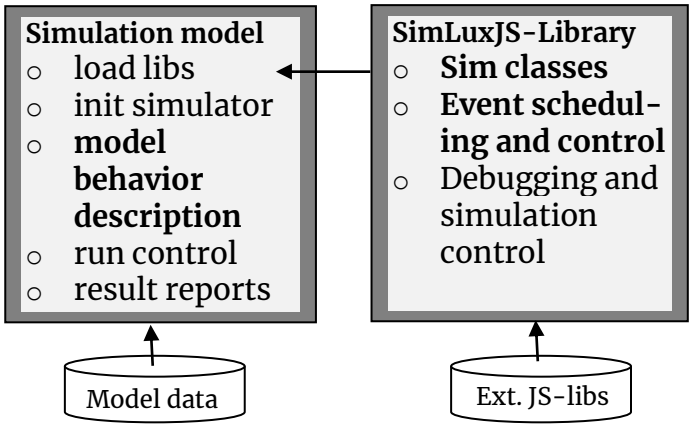

Figure 3: Main simulation program structure

The model data file(s) on the right side are optional, if there are any external data sets like work plans or order lists in a production model. The SimLuxJS -kernel-lib uses in the current version one external JavaScript module "async-mutex" for synchronizing asynchronous operations (Mutex,2024).

4.2. The SimLuxJS kernel library

The SimLuxJS kernel consists of two public classes.

The class **SimLuxJS** is the main simulator class and contains all simulation control data and methods. The class has a constructor that requires either no parameters or a Boolean flag for requesting debugging information on the console. An object of that class stands for one active simulation and schedules the simulation processes.

The class **SimEntity** is a minimalistic base class for simulated objects. They can be used by overriding their run method. That method must be asynchronous, to be executed in the correct order at the right simulated time. The execution process inside the SimEntity-class can be blocked by the waiting statements advance(time), waitUntil(condition, time-out) and waitForResource(resource). Especially the syntax elements **advance and waitUntil** are very similar statements like the counterparts in the well-known SLX-simulation system (SLX, 2018).

4.3. The SimLuxJS -Simulation control algorithms

The simulation control executes its main loop as long, as the loop body notices any change within the simulation or until an optionally given stop time is reached.

The loop body performs the following checks and always begins at the top again, after having found a positive condition, thus ensuring that earlier positive conditions have priority over later ones.

- Has a SimEntity recently reserved an awaited resource, and can it continue its work? If yes, then the simulation control continues with the next SimEntity.
- In case that a value of a control variable has recently changed, is there a waitUntil-condition in some SimEntity code, with that control variable inside? If yes, then check the condition again and if true, let that SimEntity resume operations!
- Have new SimEntities been added to the simulation with addSimEntity recently? If yes, then start their execution.
- Are there any other entries in the list of awaited times. If yes, then extract the event time of the SimEntity as the next simulated time. activate the operations on those entries again.

It is also possible to set a stop time for the simulation. If there was a change in the simulated time, then the new time is compared to the stop time and the simulation pauses, if the stop time is reached.

If there are no more SimEntities in the event list and none of these conditions apply, then the simulation ends.

4.4. An example SimLuxJS simulation model

The resulting model description in plain JavaScript-syntax is shown at figure 4. The syntax is very similar to well known Simply and SLX syntax.

At the first lines of source the SimLuxJS -kernel is imported. Then global model definitions are set. In the example there is a resource for a toll station created and the toll waiting line queue values are defined.

The asynchronous **`function car(…)`** defines the behavior of each car. In the beginning the car is parked for a certain time, then it starts driving. When it arrives at the toll station, it enters the toll queue line and then tries to enter the toll station by calling the SimLuxJS -**`waitForResource`** statement. If the toll station is available, the car enters immediately, otherwise it has to wait. After entering the toll station the tool payment is modelled by some time delay **`tolltime`**. After toll payment the car frees the toll station by calling **`releaseToll()`** function and continues driving.

```
// Import of SimLuxJS-kernel
const SimLuxJS =
require('./simLuxJS.js').simLuxJS;
const simEntity =
require('./simLuxJS.js').SimEntity;
const simLuxJS = new SimLuxJS();

// global model definitions
// of the car -model
const toll = simLuxJS.createResource(1);
let tollqueue =0, tollqueueMax = -1;

async function car(simid, carid,
parktime,drivetime1,tolltime,drivetime2)
{   await simLuxJS.advance(parktime);
    await simLuxJS.advance(drivetime1);
    tollqueue++;
    let releaseToll = await
        simLuxJS.waitForResource(toll);
    tollqueue--; // Charge now toll fee
    await simLuxJS.advance(tolltime);
    releaseToll();
    // continue after toll station
    await simLuxJS.advance(drivetime2);
… // generate SimLuxJs-object ( e.g. cars)
for (carid =1; i< number_of_Cars; i++)
{ // set other car model values and times
simLuxJS.addSimEntity(new simEntity(SimEntity =>
car(exp, carid, parktime, drivetime1, tolltime
,drivetime2)));
};
// Execute simulation(s)
async function DoSimulationExperiments ( )
{  await simLuxJS.run(until=200); }
```

Figure 4: An example model of a car toll station

Inside a for-loop a number of cars is instantiated and put into the simulation model by using the SimLuxJS -**`addSimEntity`**-method.

The shown source code elements await / advance are standard JavaScript syntax elements. If needed and if useful, the code could be packed into one single advance( ) -function.

With the last line of code, the simulation is started and executed until the given end time of 200 is reached.

4.5. Specific SimLuxJS kernel algorithms

4.5.1. Prevention of race conditions

SimLuxJS currently only uses single threading for its actual base functionality. The asynchronous functions of the SimEntities are handled by JavaScript's event queue. Because the order of events is clearly defined, no race conditions can occur.

If SimEntities are blocked by waiting statements, then JavaScript's event queue does nothing on them.

4.5.2. Internal implementation of waiting functions

The simulation uses semaphores and mutexes from the node.js package **async-mutex** for switching between the execution of the control algorithm in the **SimLuxJS** class and the end-user defined code in the **SimEntity** class (for details see (Mutex (2024))).

First simulation control is started. Whenever it starts or resumes SimEntities in its main loop, then it counts them and creates a locked semaphore (a resource without free capacity) and waits until that semaphore can be locked again with a weight that corresponds to the number of running SimEntities (until the resource has so much free capacity as there are running SimEntities).

Each SimEntity which calls a waiting statement or finishes its work unlocks the simulation control's semaphore by one (releases one capacity of the resource). When the waiting finishes, the SimEntity locks the semaphore by one again (reserves one capacity of the resource again). The simulation control only resumes execution when all SimEntities are waiting or finished.

In addition to the semaphore handling, a SimEntity, that calls a waiting statement, also triggers the creation of one waiting list entry with the parameters resource, duration/timeout and conditions. In result, the simulation control can check if and when waiting SimEntities can be resumed. These waiting list entries are grouped by equal awaited times and by identical conditions. The waiting itself is done by creating a locked mutex (a resource with capacity=1, that works like a red traffic light) for each group of waiting SimEntities. The group is then waiting for that mutex to unlock (until the traffic light switches to green). The unlocking is done by the simulation control, when it notices, that an awaited event occurred in its main loop.

4.5.3. Multicore executions

In general JavaScript can be executed in parallel on a multicore processor. The algorithms are already existing in the NodeJs-environment and can be reused also for simulation purposes.

An open question is the effective synchronization of the multicore instances, because there is no direct control available by a main processor, but the software must synchronize itself over the shared memory. But in overall, this way can help in lowering the run times of complex simulations, if any of the cores is executing the same model but with other random number sequences.

5. Distribution and performance measures

5.1. SimLuxJS-licensing and distribution

The main goal of the SimLuxJS-project is an open community project for discrete simulation at an innovative level of current JavaScript-technology.

In order to allow both commercial and free usage of the package, it is distributed under the MIT license model in the Github environment (see repo at https://github.com/htwddwiedem/SimLuxJS)

The SimLuxJS package is free for all users and especially universities and small and medium sized companies. Otherwise, the software can be included in commercial and paid simulation projects. **The authors invite all interested colleagues in a further joint development of the SimLuxJS -project.**

5.2. Discussion of actual performance issues

The performance of the developed JavaScript based simulator was compared to a very similar simulation model in SimPy. For getting a very similar semantic model, the model was first developed in SimLuxJS. Then the source code was copied to the Python environment **Thonny** and each line of JavaScript code was converted 1:1 to the Python or SimPy code.

First the performance was tested under debugging conditions with a large amount of log outputs. Under this condition the **SimLuxJS model was 20 to 30 times faster than the SimPy** model.

After switching off the log outputs, the **SimLuxJS - model execution was between 10 to 20 times slower than the SimPy model**.

In general, there was a larger variance of the run times of the SimLuxJS-models – sometime the difference between the mean and the minimal values was 50%, thus some executions were two times faster than all the others. Possible reasons could be the multitasking Windows environment with stochastically running background processes.

As a first conclusion about performance, we can state, that the overall comparison of the speed depends very heavily on the amount of text outputs during simulation. It seems, that the text visualization of Python is very ineffective and slow, compared to the speed of numeric calculations. Otherwise SimPy seems very fast in executing the model synchronization. A possible reason could be the compilation of the SimPy-environment to native code of the processor.

SimLuxJS depends on the execution of JavaScript-code inside Googles V8-engine and this will be slower in general than native code on the processor.

The performance behavior of both systems will be analyzed in the future in more detail and the authors will try to avoid ineffective JavaScript data structures by implementing optimized ones for handling large amount of active simulation objects.

6. Conclusions and outlook

In result of the new JavaScript options for asynchronous programming a new and optimized simulation system SimLuxJS was developed. The new syntax is very similar to the well known SimPy and SLX syntax, which are often used in simulation. Models, built with the new SimLuxJS -command set, are very readable like SimPy or SLX-models.

So experienced SimPy, SLX or Sim#.Net-users can use SimLuxJs as a new option of the same or very similar simulation semantics.

The performance behavior of SimLuxJS has to be analyzed in more detail, because the results depend very heavily on the amount text output during simulation in the log windows.

The future development of the SimLuxJS-package will continue in the following directions:

1. **Optimization of the simulation kernel** in terms of performance and memory usage
2. Adding an optional **graphical user interface (GUI)** for simplicity of usage also by non-computer scientists.
3. Adding an an optional **animation module** for 2D- and 3D-visualizations and animations by using existing JavaScript-based libraries.

The 2nd and 3rd development are already started by bachelor and diploma thesis of students and will continue to the end of the year 2024. First working examples will be shown at the conference.

Acknowledgements and Funding

The SimLuxJS -project was initiated by the first author based on an older version of a JavaScript based simulation (Wiedemann, 2016). The new SimLuxJS -simulation-kernel was developed by the second author of this paper in the context of the teaching module “Discrete simulation” at the University of Applied science Dresden. All performance measures were done and double checked by both authors.

The authors invite all interested colleagues in a further joint development of the SimLuxJS -project.

SimLuxJS uses the node.js library async-mutex which can be found at npmjs.com and in DirtyHairy’s Github repository at (Mutex,2024).

The project did not receive any specific grant from funding agencies in the public, commercial, or not-for-profit sectors.

References

Brandt, C., (2019) The History (and Future) of Asynchronous JavaScript https://developer.okta.com/blog/2019/01/16/history-and-future-of-async-javascript

ECMA, (2015) ECMAScript® 2015 Language Specification (ECMA-262 6th Edition / June 2015) https://262.ecma-international.org/6.0/

ECMA, (2017) ECMAScript® 2017 Language Specification (ECMA-262, 8th edition, June 2017) https://262.ecma-international.org/8.0/

MSDNASYNC (2024) MDN Web Docs (formerly Mozilla Developer Network) https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/async_function

Mutex (2024) https://www.npmjs.com/package/async-mutex https://github.com/DirtyHairy/async-mutex

Daniel C., (2012). Breaking the JavaScript Speed limit with V8 http://v8-io12.appspot.com/#2 2012

Gudabayev T. (2021). Separation of Concerns The Simple Way https://dev.to/tamerlang/separation-of-concerns-the-simple-way-4jp2

SimLuxJS-GitHub-Repo (2024). https://github.com/htwddwiedem/SimLuxJS

Schriber T.,Brunner D., Smith J., (2013). Inside Discrete Event Simulation Software: How It Works and Why It Matters. Proceedings of the 2013 Winter Simulation Conference, Pages 424-438

SLX, (2018) Homepage of the SLX-simulation language www.wolverinesoftware.com/SLXOverview.html

Varshney, M. (2011) . Sim.js-Homepage simjs.z5.web.core.windows.net/download.html

Wiedemann, T. (2016) Using the JavaScript ENGINE NodeJS for discrete simulation in a web-based environment. Proceedings of the 2016 European Modeling & Simulation Symposium, Pages 334-337