CONVERTING THE CELL-DEVS WEBVIEWER TO A GENERIC DEVS WEBVIEWER

SYSC5900 F – System Engineering Project

Department of Systems and Computer Engineering

Carleton University

Project Submission to:

Prof. Gabriel Wainer

Project Report Submission by:

Navneet Kaushal

Student Id: 101094963

Master of Engineering, Electrical and Computer Engineering,

Department of System and Computer Engineering,

Carleton University

Table of Contents

1. INTRODUCTION

The absence of a generic visualizing tool has been one of the major setbacks for limiting the usability of DEVS among non-expert users. Most simulation tools are built for domain-specific models and thus their visualization tools are also domain-specific. Though the visualization tools help users easily analyze their models by hiding complex simulation results, it reduces the reachability of the tool. On the other hand, we have few visualization tools that are not restricted to a certain domain, however, they are not compatible with all kinds of DEVS models. Cell-DEVS WebViewer is an existing example of that type of tool.

The aim of the project is to develop a generic DEVS WebViewer that is not restricted to a certain domain and should also be able to visualize the simulation results of all kinds of DEVS models. It has been observed that modifying the existing Cell-DEVS WebViewer to a generic WebViewer would be a wiser step rather than building a new application from scratch. We already have few versions of the Cell-DEVS WebViewer; the later version is a complete rebuild of the tool that followed a more modular object-oriented approach aiming the extension of the tool in the future. Currently, Cell-DEVS WebViewer can parse simulation results of Cell-DEVS models from CD++ and Lopez simulators. To make it a generic application, a new parser for simulation results of Classic DEVS models needs to be included. In the future, simulation results from the Cadmium tool would also be possible to parse easily because of the new design of the WebViewer.

The Cell-DEVS WebViewer was originally developed by the Advanced Real-time Simulation Laboratory at Carleton University for academic use in courses and publications (Van Schyndel et al. 2016). As the second version of Cell-DEVS WebViewer (St-Aubin et al. 2018), the generic DEVS version is also a web-based platform for visualization and analysis of DEVS simulation results. The viewer is coded entirely in HTML5 and JavaScript, it can be run locally on a user's computer as a client-application and does not have external dependencies.

To achieve the objective of making a generic WebViewer, this project contributes to the redesign of the data structures with a new data storage strategy, along with little changes in the architecture. In a nutshell, the paper explains how the objective of changing a grid-based web viewer into a model-based web viewer is achieved.

2. BACKGROUND AND RELATED WORK

2.1 DEVS Overview

DEVS models have been used for experimentation in plenty of cases from different domains. Discrete Event Systems Specifications (DEVS), a well-recognized technique for efficiently modeling real-life systems, was first introduced by Bernard Zeigler in 1976 (Zeigler, Praehofer, Kim 1976). It provides a discrete event-based method to visualize systems into models so that they can be used for experimentation in cases where working on the real system is not possible. (Vangheluwe 2000 and St-Aubin, et al. 2018). Initially, DEVS was introduced for modeling and simulating discrete-event dynamic systems (DEDS). Therefore, it provided a way to define systems that change their state on the reception of an input event or due to the expiration of a time delay.

DEVS Model is organized hierarchically, also, the higher-level components are broken down into simpler elements to deal with the complexity of the systems. DEVS, therefore, supports hierarchical and modular models that can be easily reused. It defines a rigorous formalism which fits the general structure of deterministic systems in classical systems theory. In the DEVS system, behavior can be described at two levels. The lowest level is atomic DEVS which describes the autonomous behavior of a discrete-event system and at the higher level, a coupled DEVS which describes a system as a network of the coupled component. The connections denote how the components influence each other. Output events of one component may become an input event of another component via a network connection. The DEVS formalism was conceived by Zeigler to provide a rigorous common basis for discrete-event modeling and simulation (Hans Vangheluwe, 2005).

DEVS is a timed event system used for modeling and analyzing discrete event dynamic systems. A real system modeled using DEVS is made up of atomic and coupled models. DEVS is a methodology to specify systems whose state can change upon reception of an external input event or due to the expiration of a time delay (Wainer, 2009).

Any real system modeled using DEVS is composed of atomic and coupled models. Atomic models are defined as:

$$M = < X, Y, S, \delta int, \delta ext, \lambda, ta >$$

Where X is the set of input events; Y is the set of output events; S is the set of sequential states; δext is the external state transition function; δint is the internal state transition function; λ is the output function; and ta is the time advance function as shown in Figure 1. A DEVS model stays in a state S for a time ta in absence of an external event. On the expiration of ta the model outputs the value λ through a port Y and changes to a new state given by δint. This transition is called an internal transition. If there is a reception of an external event, δext determines its new state. If ta is infinite, then s is said to be in a passive state (Wainer, 2009).

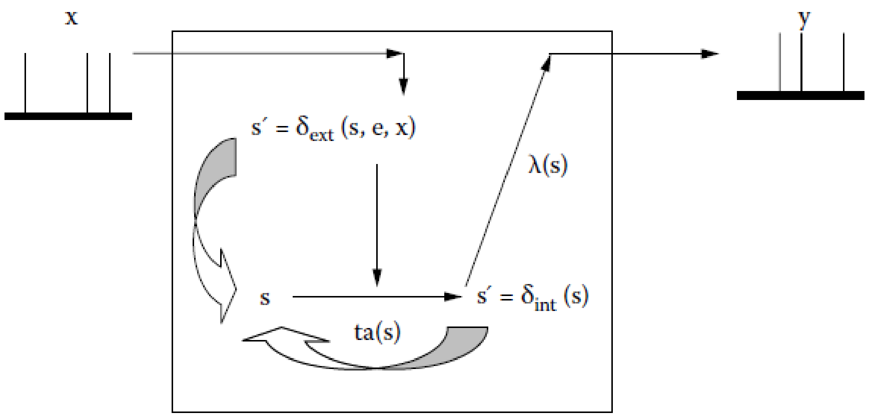


Figure 1. DEVS Atomic Model Semantics. Image Courtesy (Wainer, 2009)

The value of the time advance function may range from 0 to ∞. When the value of time advance function is 0, then *s* is said to be in a transient state. This transient state triggers an instantaneous internal transition. On the other hand, if time advance function is ∞ then *s* is said to be a passive state. In this state, the system will remain as such until and unless an external event is received.

A DEVS coupled model is composed of several atomic or coupled submodels. It is formally defined by-

$$CM \; = < X, Y, D, \{M_d | d \in D\}, EIC, EOC, IC, select >$$

Where X is the set of input events; Y is the set of output events; D is the set of the component names and for each d ∈ D; M_d is a DEVS basic (i.e., atomic or coupled) model; EIC is the set of external input couplings; EOC is the set of external output couplings; IC is the set of internal couplings, and select is the tiebreaker function.

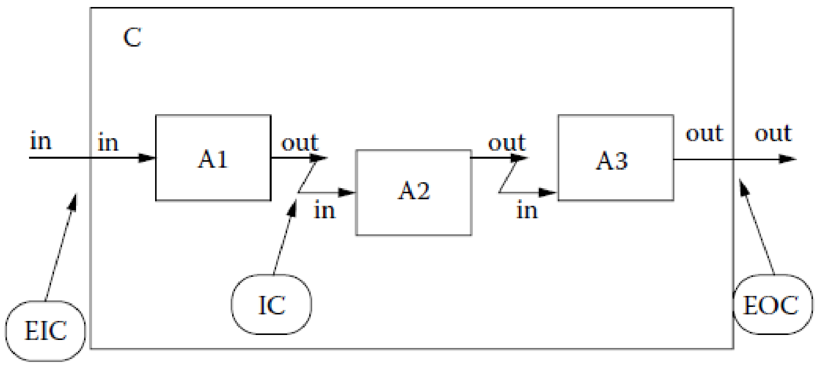


Figure 2. DEVS Coupled Model. Image Courtesy: (Wainer, 2009)

The coupled models are defined as a set of basic components (atomic or coupled) interconnected through the model's interfaces. The translation function is in charge of converting the outputs of a model into inputs for the others. An example of a DEVS coupled model is shown above with three subcomponents, A1‑A3. These basic models are interconnected through the corresponding I/O ports presented in Figure 2. The models are connected to the external coupled model through the EIC and EOC connectors. The models A1‑A3 are basic models, they can be atomic or coupled models (Wainer, 2009).

The Cell-DEVS formalism was introduced by Wainer and Giambiasi, it is a combination of cellular automata and the discrete event system specification (DEVS) with explicit timing delays (Wainer and Giambiasi 2002). A cell DEVS model is defined as the lattice of cells. Each cell has a value and a local rule which determines how the new value will be obtained based on the current value of the cell and the value of its neighbors. Cell-DEVS defines a cell as a DEVS model and a cell space as a coupled model. A cell uses a set of input values to compute its future state. The future state is obtained by applying the local computation function τ. A delay function d is also associated with each cell. It defers the output of the new state value. After the basic behavior for a cell is defined, the complete cell space can be constructed by building a coupled Cell-DEVS model (Wainer, 2014).

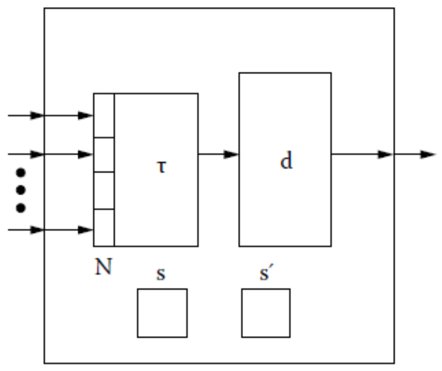


Figure 3. Cell-DEVS atomic model definition with transport delay. Image Courtesy: (Wainer, 2009)

This model can be formally described as-

$$TDC = < X, Y, S, N, type, d, \tau, \delta int, \delta ext, \lambda, D >$$

Where X is the set of input external events; Y is the set of output external events; S is the state set; N is the set of input values; d is the delay for the cell; type is the kind of delay (transport/inertial/other); τ is the local computing function; δint is the internal transition function; δext is the external transition function; λ is the output function; and ta is the state's lifetime function (Wainer, 2009).

Similar to classic DEVS, every cell has functions for the internal transition, an external transition, output and time advance. As long as a cell receives external events or it has an internal event scheduled the cell remains active. The cell passivates when there are no more events. When a cell changes state, its new state will be transmitted to neighboring cells only after a delay (transport or inertial). In transport delays, the new state is guaranteed to be communicated to neighboring cells once the delay expires, whereas, the inertial delays provide a mechanism to pre-empt the state change and prevent the cell from transmitting to neighboring cells. This usually happens when a cell that was scheduled for a change state receives another event that overrides the cell state before the completion of another transition.

There are various advantages of using DEVS for modeling and simulation. Firstly, it gives better performance, as compared to other modeling methods, because of its faster computation ability. DEVS also simplifies the complex model definition by using a hierarchical approach. The major advantage of using DEVS models for the visualization tool is that it is open to all domains and it doesn't restrict users to simulate models from a specific domain.

2.2 The Role of Visualization in DEVS and WebDevelopment Overview

A good visual analysis tool offers detailed information about the simulation, the capability to identify invisible errors in the model and to identify different patterns in the state transition process. Developing user-friendly, dedicated analysis tools, is also a way to make modeling and simulation accessible to a wider public (St-Aubin et. al, 2018). Collins and Knowles Ball (2013) even argue that what matters most is what that outside of the modeling and simulation community can see. The presentation of the simulation's results is more important than the simulation itself. Visualization and analysis do not provide a substantial amount of research. However, efficient visualization and analysis of the simulation and its results are key to decision-making for stakeholders other than the simulator itself.

One of the drawbacks of DEVS is that its approachability is limited among non-expert users, therefore, researchers have tried to solve this issue by making user-friendly interfaces that abstract DEVS theory (Goldstein et. al. 2016-2018, Bergeron, Kofman 2011, Sarjoughian and Elamvazhuthi 2010). Most of the above researches are focused on modeling and simulation aspects and some of them propose visualization and analysis tools. DEVS software development merely aims at the analysis of the results instead it usually more dedicated towards the performance, model debugging, verification and validation or the modeling process. For the visualization and analysis of the detailed assessment of results usually third party software such as MatLab, R or Tableau are referred. This gap can be filled by developing a visualization tool for DEVS that would provide greater usability of DEVS in the future. It will also contribute to a higher adoption rate of DEVS among the community of simulation and modeling users. Data visualization has begun to generate a more significant amount of research in recent years. Data visualization is focused on providing an accurate representation of large volumes of data so that it can be interpreted by users and there can be deep insights into the data. Since a high volume of data can be generated by simulations, data visualizations concepts can be applied to the analysis of the simulation outputs (St-Aubin et. al, 2019).

JavaScript, a dynamically typed interpreted language, is one of the most popular languages that provides a number of frameworks and libraries. Almost 99% of all web visualization examples are in JavaScript. JSON (JavaScript Object Notation), a de facto web visualization standard, plays very nicely with JavaScript (Dale 2016). ES6 specification, the standard governing JavaScript, provides a really strong feature of JS becoming an object-oriented compatible language. ES6 also added some other features such as promises for asynchronous actions get and set accessors for classes, anonymous functions, default values for parameters passed to functions, etc. Initially, JavaScript was a prototype-based, an uncommon form of object-oriented programming, language. Another important feature added by ES6 is to provide the possibility to import or export modules within other modules. Earlier, the dependent files were included in the HTML files, that too in a definite order. Doing so may restrict the software developers to restrict the number of files and push a larger size of code in a single that was difficult to maintain. All these features from the new ES6 specifications were used to refactor the WebViewer to be able to follow an object-oriented architecture to build the software (St-Aubin et. al, 2019).

2.3 Evolution of the Cell-DEVS WebViewer

The WebViewer was originally built as an alternative to the desktop-only simulation viewer that was included in the CD++ development environment (Chidisiuc, Wainer 2007). The application was developed with a plan to have more control over the visualization, offer a better loading process for simulation results and more user-friendly visualization of the results. The initial version of the Cell-DEVS WebViewer application has been developed as a standalone, lightweight, web-based visualization and analysis platform as a replacement to expensive proprietary software. The application offered a user-friendly way to post-process Cell-DEVS simulation results. The software was entirely written in HTML5 and JavaScript though there have been dependencies on the Whammy.js framework for recording videos of canvas animation in .webm format and Data-Driven Documents (D3) library for dynamic charts (St-Aubin, 2018).

This application was developed to be run entirely on the client-side, however, if needed it could be hosted on the server as well. Therefore, no installation of software was required because of its accessibility on the web. To use the application all the cell-DEVS simulation files (.log, .val, .pal, and .ma) should be loaded from the upload section.

- *.val file:* the initial value file; contains the cell-space state for time step 0
- *.ma file:* the model file; a text file that contains the simulation configuration parameters required to visualize the results
- *.pal file:* the color palette file; defines the colors associated with each cell state for cell-space rendering
- *.log file:* the result log file; contains all the transitions that were made by each cell in the cell-space for the duration of the simulation

Since the size of the .log file is sometimes very huge, it was not possible to directly parse these large files in one go at the browser-side. Due to new developments in HTML5 and JavaScript, it has been possible to get the task done. The HTML5 FileReader object has been used to quickly load a very large .log file. The file reader object divides the file content into chunks and loads it in memory for the WebViewer to animate and analyze.

The animated results have been made available for the visual representation of the simulation. Visualization of the cell-DEVS simulation results is basically a rendering of cell-space to the screen based on the result data starting from the values of the cell-space at time 0 till the end time of the simulation .log file. For each time step, the transition of each cell state has been stored. Based on the transitions read from the log file, the cache was built to let the user progress through the simulation playback. To render the cell-grid. the earlier versions of the viewer used HTML divs and SVG (Scalable Vector Graphics) elements. However, it was not efficient to use these DOM (Document Object Model) elements. Therefore, Canvas elements were used to draw the cell grid because Canvas is suitable with a very large number of elements that need to be drawn, removed or redrawn frequently (MSDN 2017). Moreover, the Canvas element is also compatible with 2D and 3D rendering. Once the initial cell-space was rendered, the reconstructed simulation could be run using the playback widget of the WebViewer. Navigating to different time steps, recording videos of their simulation, inspecting the state of individual cells, and exporting the raw data in JSON for further processing in other external programs; these have been the wonderful features of this application (St-Aubin, 2018).

The tool has also provided statistical analysis capabilities in the form of charts (bar chart and heat map chart). These charts were implemented using the Data-Driven Document (D3), a JavaScript API, built by Mike Bostock in 2011. It is currently one of the most widespread

JavaScript data visualization libraries on the web and serves as a base for other higher-level data visualization libraries that are derived from it. It provides an easy way to bind data to document elements (Bostock, Ogievetsky, Heer 2011). The visualization of the result in the form of animated charts was capable of displaying data derived from the simulation as it was being executed.

The second version of the web viewer consists of many improvements from the first version. There have been significant changes in all sections of the previous version of the WebViewer; from converting it to a more cohesive architecture, providing a more flexible interface for the user, adding new analytical and interaction features. With the help of the new specifications refactoring the web viewer has become convenient. Since the first version of the WebViewer used a very weak object-oriented approach while developing the application, therefore, this second version was a complete rebuild of the viewer that followed a more modular, object-oriented approach. This way it limited the development complexity involved in reconstructing the simulation and made the application easy to maintain and extend in the future. One of the key features added in this version was to automatically detect the parser as soon as the simulation log file is uploaded. However, this step had the main issue of reading the file through the HTML5 file reader's event-based asynchronous process because all the parser classes have to be tested against the log file sequentially until a valid parser type is found. Promise class of the ES6 specification became helpful to tackle this issue. A Promise is an object that is instantiated with an Executor parameter that provides Resolve and Reject callback functions to be executed upon successful or failed resolution of the deferred call. In cases where a parser class cannot be identified, the system will throw an error with proper feedback to the end-user. Once the parsing strategy is identified, it is passed to the Simulation object (St-Aubin et al. 2019).

The object-oriented approach of the second version of Cell-DEVS WebViewer would be easier to understand through the class diagram of the *Simulation* Class and how it is associated with other classes (See Figure 4). Due to the importance of the *Simulation* class, it is considered at the heart of the CD-WebViewer. The simulation object holds all the data required to visually reconstruct the simulation. It has several methods to load the simulation files, to advance or rollback the time frames during visualizing the simulation and to retrieve the cell-space at a given time step. This class is a subclass of *Evented* class, this class dispatches events that can be listened to externally. It is also important to decouple the simulation class from different user interfaces. As soon as the simulation files are loaded, the *Parser* class detects the parser through its subclasses CDpp and Lopez. Depending on the parser type is detected, one of these subclasses starts parsing the simulation files (.log, .pal, .val). The parsed information from .pal file is stored into an object of *Parse* class. The data of .val file is stored in the message object and further, the output data of the .log file is also passed chunk by chunk into the same object. The message object stores the information in the form of three-dimensional coordinates and its value for each timestamp. *Message* class is used for data storage from the simulation log file. *Frame* class associated with *Simulation* class is involved in creating the frame objects. These frames are basically a collection of the output value of each model at each timestep. The frame object is an ordered data structure of message object data. *Transition* class stores the difference of the value of the between the current transition state and previous transition state.

Once the entire log file is parsed, the simulation object builds the *Cache* object of cell-space states. With all data filtered, the *Simulation* object builds a *Cache* object of cell-space states. The aim of caching is to achieve the fast processing of simulations with a very large number of frames. It consists of storing the entire cell-space state every n frame from the first till last cell-space states. The cached cell-space state is the clone of the current cell-space state so that while applying the transition doesn't modify the actual content. It helps in applying the transition for each frame between the cached frame and the frame selected by the user through the frame slider. The Cell-DEVS WebViewer also displays a set of information about the simulation in the info widget. In this version, all this information such as the size of the cell-space, the number of transitions, the number of frames, etc. are now determined from the .log file which used to be rendered from the model file.

A new analytics feature, *Cell Track Chart,* was also introduced in this version of the viewer. This analysis feature gives the users frequently track the state of selected cells across the simulation. One of the other major contributions of this second version of Cell-DEVS WebViewer has been the introduction of RISE parser, that can parse the simulation result files coming from the Lopez simulator, aiming the integration of this web-application with the RISE server in future. RISE is a RESTful distributed simulation platform that supports DEVS and other formalisms (Wainer and Wang 2017, Al-Zoubi and Wainer 2015).

3. NEW DEVELOPMENTS

The previous version of WebViewer (CD-WebViewer) handled the shortcomings of the initial version, while this new version of WebViewer (DEVS WebViewer) tries to extend the previous application to make it a generic DEVS WebViewer. It doesn't show any significant change at the user interface level, however, it certainly makes the application more flexible towards accommodating results of other models than just Cell-DEVS.

3.1 The Transition from a Cell-based to a Generic DEVS Viewer

The transition of this version of generic DEVS WebViewer application from the previous version of WebViewer did not face major challenges because of the modular object-oriented approach adopted in the previous version. The code of each component was separated as a module and it has become easier to twist each module separately as per the need.

However, the complex structure of the application has been a concern. Due to the interdependency of each component with other components, the code change was complicated. Understanding the architecture of the application is key to making any progress in the direction of modifying the application. To achieve the objective of converting the previous version into a generic DEVS visualization tool, the transition in the application has to happen majorly at two levels:

a) Transition at the level of architecture
b) Transition at the level of data structures

During the development, the changes were made at the data structure level first and then slowly moved to the architectural level update. However, it was essential to understand the architecture before any change in the data structure because of the interdependency. Thus, the discussion on the transition at the level of architecture is preferred over that on the other transition.

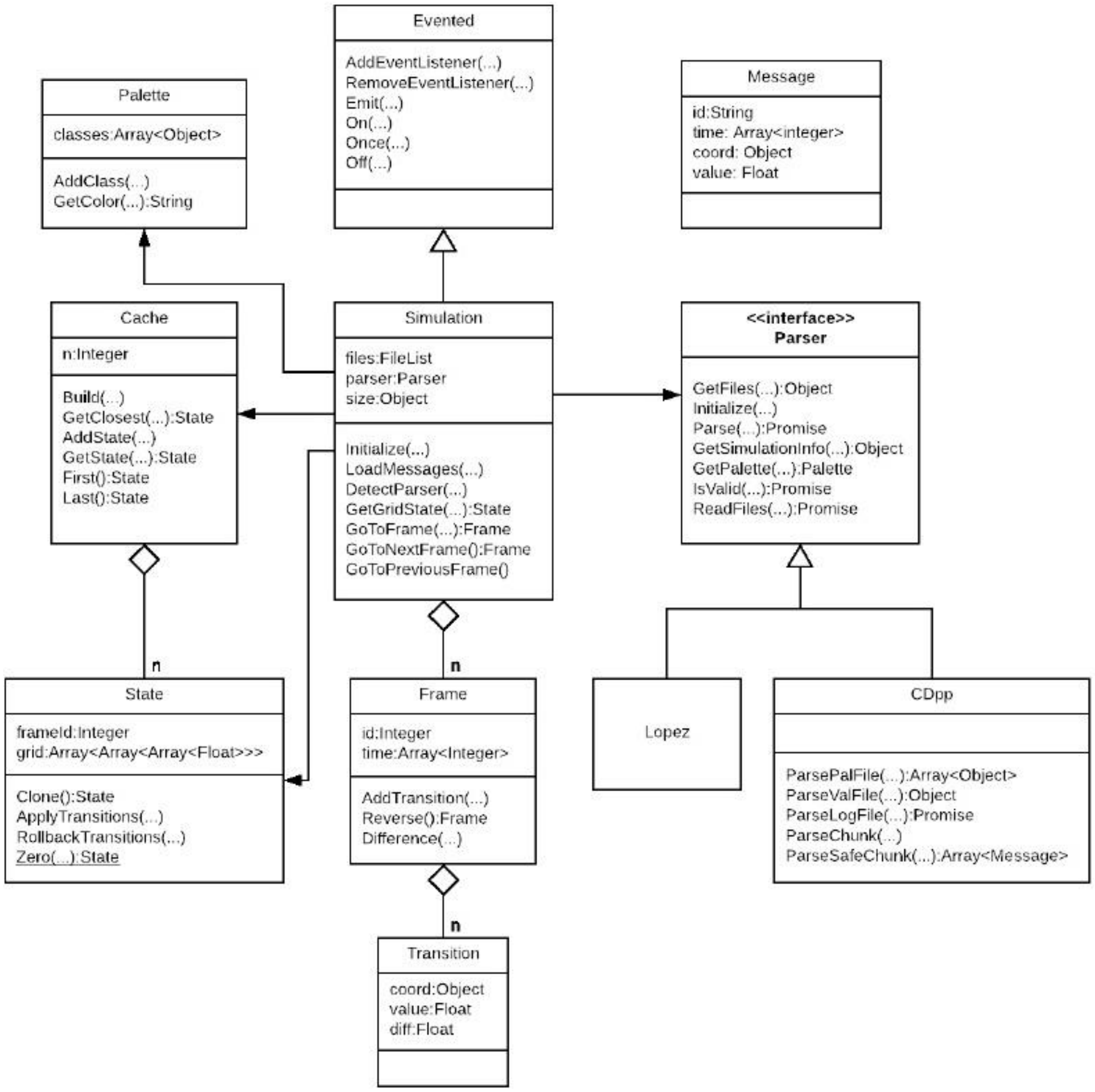


Figure 4. Class Diagram of Cell-DEVS WebViewer. Image Courtesy:

One of the aims of changing architecture has also been to simplify it by eliminating any unnecessary class in the application. It has been observed that *Message* class (See Figure 4) can be avoided as it is just holding the data until *Frames* are created. It would be a clever approach to discard the *Message* object and directly parse the .log and store the data into a *Frame* object for each time step. Thus, the *Message* class is not seen in the new architecture of the application (See Figure 5).

Another significant change in the architecture has been to include a dedicated parser for Classic DEVS models, named as *DEVS* parser. This *DEVS* parser also inherits *Parser* class like any other parser type class. The discussion on *DEVS* parser has been done in detail in the latter part of the report.

It has been already discussed in the background section that the second version of the Cell-DEVS WebViewer has reconstructed the simulation and other classes considering the future extension and maintenance of the application. The *Simulation* class being the main class of the application communicates with all other classes. All the parsed data is stored as part of the simulation object and is available for use by any other objects or the visualization objects.

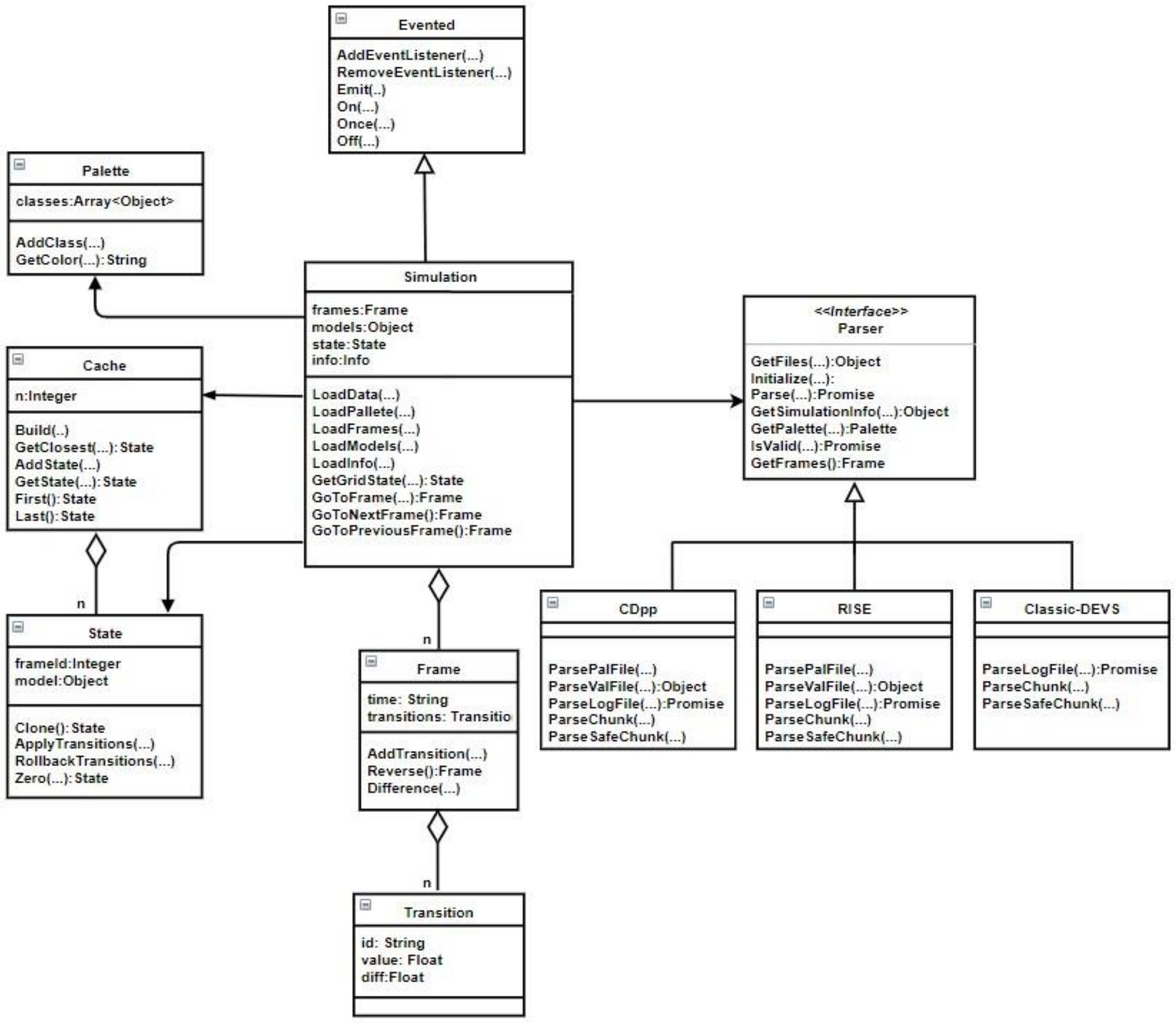


Figure 5. Class Diagram of DEVS WebViewer.

Instead of *LoadMessage()*, now *LoadFrames()* is storing the frame object data whereas *LoadModels()* method is now been used to load the models as a separate object. Model ids are separately stored inside this object. This *Simulation* class is adherent to the data structure modifications done in the application. All the grid type data handling has been changed to a model type data handling approach so that adding a new parser, like one for Classic DEVS, does not require any change in the *Simulation* class or in its associated classes. It's just an easy plugin to introduce a new parser as long as its parser subclass file is able to create the frames in the current format.

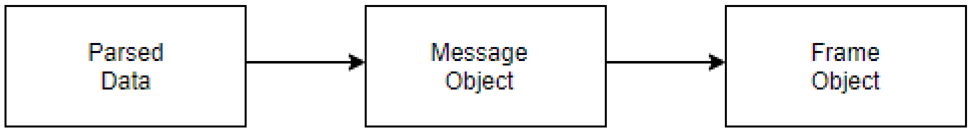


Figure 6. Data Storage Strategy in CD-WebViewer

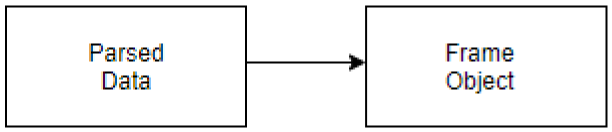


Figure 7. Data Storage Strategy in DEVS-WebViewer

The data storage technique has also been changed a little due to the elimination of the *Message* class (See Figures 6 and 7). Instead of storing the data into message objects and then creating frames at a later stage, the frames are created at the same time as reading the chunks of data (See Figures 8 and 9). In simple words, the parsers are now able to directly parse the data and directly create *Frame* and *Model* objects in the same class. The parsing strategy, however, is more or less similar to the previous version CD-WebViewer.

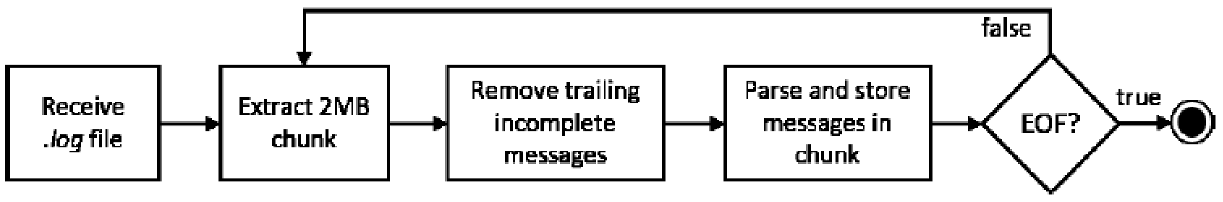

Receive .*log* file
Extract 2MB chunk
Remove trailing incomplete messages
Parse and store messages in chunk
EOF?
false
true

Figure 8. Data Parsing Strategy in CD-WebViewer

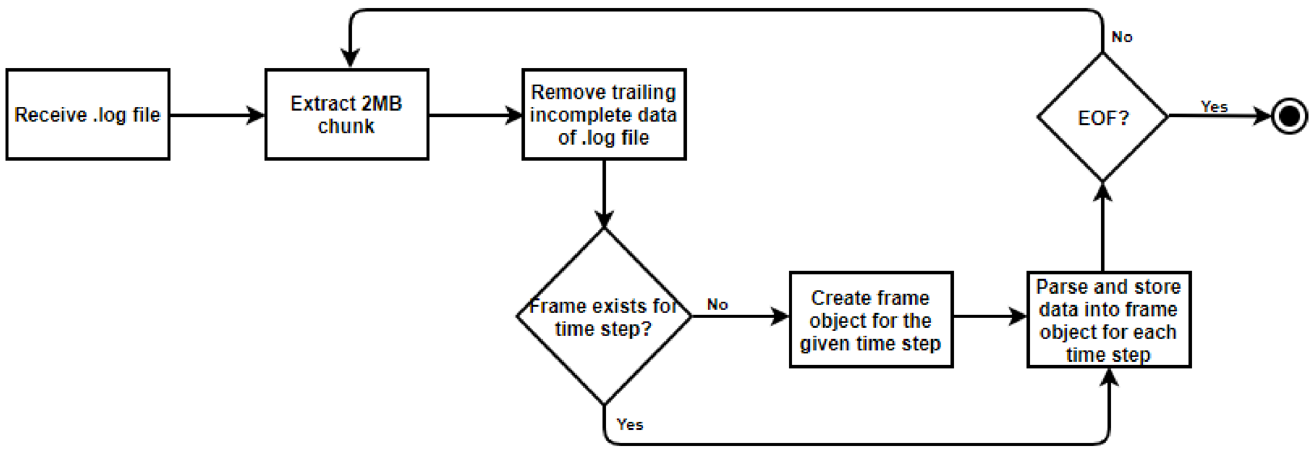

Receive .log file
Extract 2MB chunk
Remove trailing incomplete data of .log file
Frame exists for time step?
No
Yes
Create frame object for the given time step
Parse and store data into frame object for each time step
EOF?
No
Yes

Figure 9. Data Parsing Strategy in DEVS-WebViewer

Models (model ids) are stored in a *Model* object along with the storage of data into a *Frame* object. Storage of models is done in an object data structure to keep the model id unique. It stores a new model in the object because of the key-value feature of the object, it is treated as a dictionary whenever there is a need for the model value to be matched. These data parsing and data storage strategies are adopted for both simulator parser types: CD++ and Lopez (from RISE server) parsers.

The major transition has happened for the data structures. Almost all of them including various objects of almost all the classes are now changed into a model-based data structure that was grid-based (or with coordinates) earlier.

```
Mensaje Y / 00:00:00:100 / lug(19,16,0)(3835) / out /   102.00000 para lug(02)
```

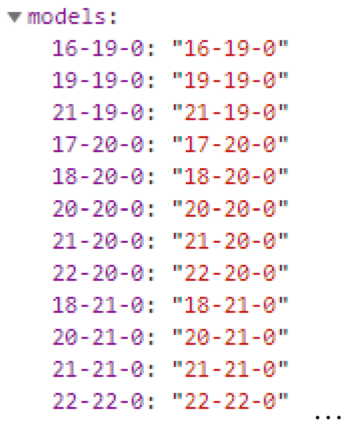
The previous message object, an array of objects, used to parse the above data line from .log file into id, time, coordinates, and value as its attributes in below format-

id: "00:00:00:100"
time: [0, 0, 0, 100]
coord: [19, 16, 0]
value: 102

In the new architecture, where the *Message* class is removed and a new *Models* object is used to store the model information by following the logic given below:

```
model = X coord + "-" + Y coord + "-" + Z coord (if no Z, put 0)
this.models[model] = model;
```

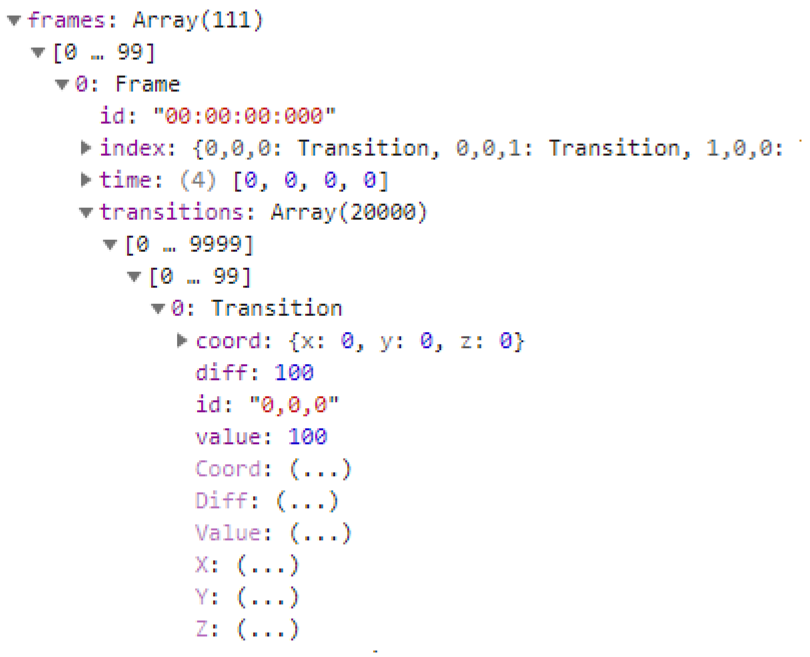
This will keep pushing the different coordinates of Cell-DEVS models in the format shown in Figure 10. For a given object id (model id), it can store only one value of the model. Therefore, there won't be a repetition of model objects. It is one of the advantages of using this data structure in an object type over using an array. Another advantage of using this data structure is that it can be treated as a dictionary that works as a key-value pair.



```
▾models:
   16-19-0: "16-19-0"
   19-19-0: "19-19-0"
   21-19-0: "21-19-0"
   17-20-0: "17-20-0"
   18-20-0: "18-20-0"
   20-20-0: "20-20-0"
   21-20-0: "21-20-0"
   22-20-0: "22-20-0"
   18-21-0: "18-21-0"
   20-21-0: "20-21-0"
   21-21-0: "21-21-0"
   22-22-0: "22-22-0"   …
```

Figure 10. Data Structure of Model Object in DEVS-WebViewer

In the previous version of WebViewer, *Frames* were data structures of an ordered array where each *Frame* contains a time value and an array of *Transitions* for a given time step. A *Transition* consists of a grid coordinate (x, y, z) and a value representing the new state of the cell at that time step (See Figure 11). The Frame data structure was having id, index, time, transitions as its components, in which each id and time components were used to denote a particular time step in different data structure types- id being a string in timestamp format, whereas time as an array of length four denoting hour, minute, second, millisecond in different indices. However, it was not required to have both in the Frame data structure since both presenting the time step.



```
▼frames: Array(111)
  ▼[0 … 99]
    ▼0: Frame
        id: "00:00:00:000"
      ▶index: {0,0,0: Transition, 0,0,1: Transition, 1,0,0:
      ▶time: (4) [0, 0, 0, 0]
      ▼transitions: Array(20000)
        ▼[0 … 9999]
          ▼[0 … 99]
            ▼0: Transition
              ▶coord: {x: 0, y: 0, z: 0}
                diff: 100
                id: "0,0,0"
                value: 100
                Coord: (...)
                Diff: (...)
                Value: (...)
                X: (...)
                Y: (...)
                Z: (...)
```

Figure 11. Data Structure of Frame & Transition Object in CD-WebViewer

Considering the unnecessary additional space in the memory, in the new version of WebViewer, one of them is removed. Eventually, the id attribute of the Frame data structure is removed but its value in the string, a simpler form of data structure, is assigned to the time component.

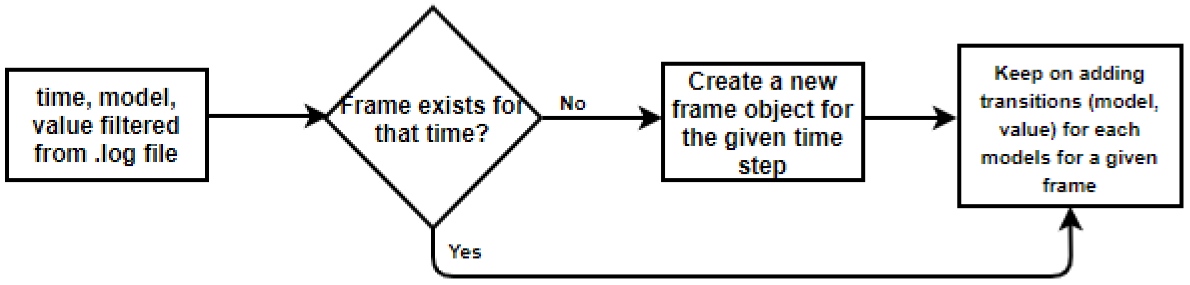
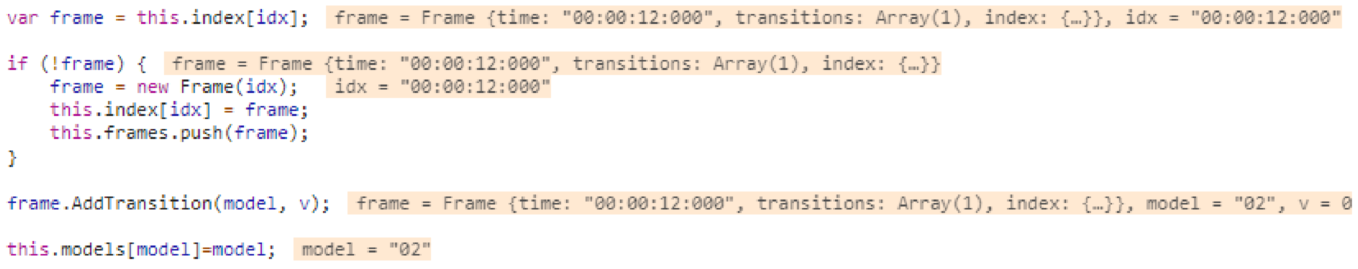

time, model, value filtered from .log file
Frame exists for that time?
No
Create a new frame object for the given time step
Keep on adding transitions (model, value) for each models for a given frame
Yes

Figure 12. Algorithm of creating the Frame objects by adding transitions in DEVS-WebViewer

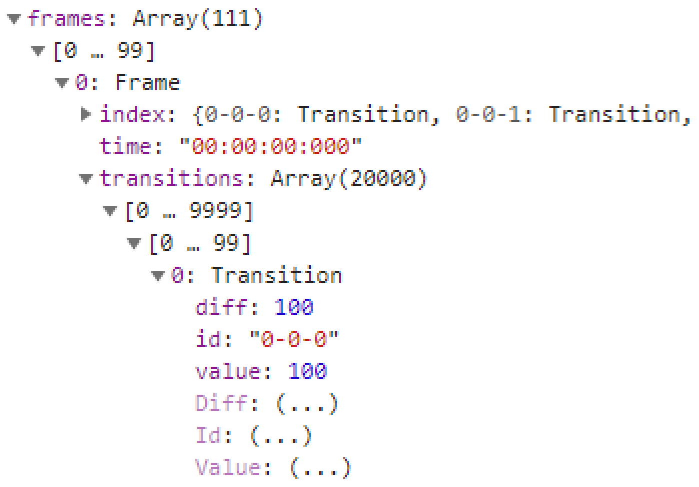
In this newer version, the algorithm (See Figure 12) to create frame objects is implemented in the *Parser* class itself. For each time step, a frame object is created and all the models and its values are stored by adding the transition into the frame object. How the creation of *Frame* objects and *Model* object storage been done is illustrated in Figure13 with .log data flow for a particular moment. This snap is taken from the Chrome Developer tool.



```
var frame = this.index[idx];  frame = Frame {time: "00:00:12:000", transitions: Array(1), index: {…}}, idx = "00:00:12:000"

if (!frame) {  frame = Frame {time: "00:00:12:000", transitions: Array(1), index: {…}}
    frame = new Frame(idx);   idx = "00:00:12:000"
    this.index[idx] = frame;
    this.frames.push(frame);
}

frame.AddTransition(model, v);  frame = Frame {time: "00:00:12:000", transitions: Array(1), index: {…}}, model = "02", v = 0

this.models[model]=model;  model = "02"
```

Figure 13. Frame Creation and Model Storage Code (with an example of data flowing through)

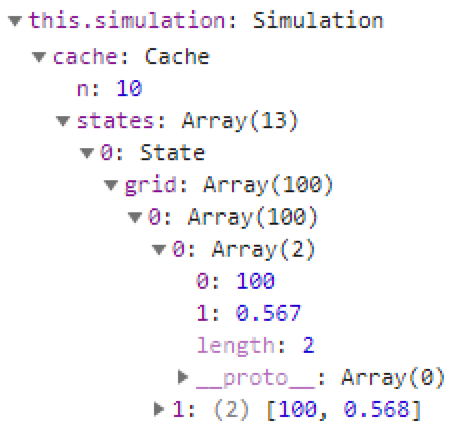
As the data structure of the *Frame* object, the data structure of the *Transition* object also has two attributes id and coordinates holding similar model data (coordinates of cell space, each cell in a cell space is considered as an atomic model). Also, there are get methods returning the X, Y, Z coordinates of the models, in the version of CD-WebViewer. Whereas in the DEVS-WebViewer, the coordinates have been completely removed from the common architecture of the application so it is not considered in the *Transition* class too. It only contains id storing the coordinates of the model as a string in the “x-y-z” format, along with the value and difference attributes for that model (See Figure 14). This id is similar to the model id stored in the *Model* object. Instead of returning different coordinates now a getter method is there to retrieve the transition id.



```
▼frames: Array(111)
  ▼[0 … 99]
    ▼0: Frame
      ▶index: {0-0-0: Transition, 0-0-1: Transition,
       time: "00:00:00:000"
      ▼transitions: Array(20000)
        ▼[0 … 9999]
          ▼[0 … 99]
            ▼0: Transition
                diff: 100
                id: "0-0-0"
                value: 100
                Diff: (...)
                Id: (...)
                Value: (...)
```

Figure 14. Data Structure of Frame & Transition Object in DEVS-WebViewer

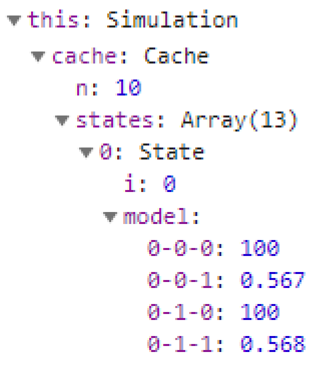
The process of caching is no different from the previous version so there is no change in the data structure of a *Cache* object. In the current version also, the cache consists of two attributes: number of frames and state object of each frame. However, there is a complete transformation of the data structure of the *State* object. Earlier each *State* object used to consist of an array of grids, and the value for that grid model like in Figure 15 for grid [0,0,0] the value is 100, and for grid [0,0,1] it is 0.567. Every time, to retrieve the value of any grid, the entire array is to be traversed in multiple loops.



```
▼this.simulation: Simulation
  ▼cache: Cache
     n: 10
    ▼states: Array(13)
      ▼0: State
        ▼grid: Array(100)
          ▼0: Array(100)
            ▼0: Array(2)
               0: 100
               1: 0.567
               length: 2
              ▶__proto__: Array(0)
            ▶1: (2) [100, 0.568]
```

Figure 15. Data Structure of Cache & State Object in CD-WebViewer

Whereas, in the new version, the data structure of the *State* object is pretty straight forward in a key-value format. Each value is stored for the given model-id as a *Model* object like the state value of model “0-0-0” is 100 and that of model “0-0-1” is 0.567 as seen in Figure 16.



```
▼this: Simulation
  ▼cache: Cache
     n: 10
    ▼states: Array(13)
      ▼0: State
         i: 0
        ▼model:
           0-0-0: 100
           0-0-1: 0.567
           0-1-0: 100
           0-1-1: 0.568
```

Figure 16. Data Structure of Cache & State Object in DEVS-WebViewer

One of the major advancements has been in reducing the time complexities for the algorithms used in the application. For instance, the *Zero()* method of *State* class returns a state object with 0 values of each grid. It takes the size with (x,y,z) components and pushes the number of arrays in a three dimensional way using three for loops resulting in time complexity of O[n^3], it is evident in the below code snippet from *State* class of the previous version.

```
    static Zero(size) {
        var grid = [];
        for (var i = 0; i < size.x; i++) {
            grid.push([]);
            for (var j = 0; j < size.y; j++) {
                grid[i].push([]);
                for (var k = 0; k < size.z; k++) {
                    grid[i][j].push(0);
                }
            }
        }
        return new State(-1, grid);
    }
```

After the changes made in the current version of the application, now the value 0 can be directly assigned to the model ids of the *Model* object that makes the same *Zero()* method of *State* class (see the below code snippet from *State* class) so simpler with a significant decrement in time complexity from O[n^3] to O[n].

```
    static Zero(models) {
        var model = models;
        for (var id in model) {
            model[id] = 0;
        }
        return new State(-1, model);
    }
```

The improvement in the data structure of the *State* object as one of the components of the *Cache* object can be seen clearly in Figure 16 compared to its previous version in Figure 15. The *State* made up of three-dimensional arrays of grids is now reduced to a simple *model* object.

Adding any analytical feature to the application was not part of the scope of this project. But few changes have been done to intact the analytic feature of the viewer for Cell-DEVS models.

To keep the visual presentation the same as before the auto classes of all kinds of widgets require modification to be compatible with the code changes done in *Simulation* and its associated classes. Now the single model object has to be split into three dimensions wherever required in the cell-DEVS class. But these are just the model-specific (cell-space CD models or classic DEVS models) changes and applicable only where necessary. It was convenient to use the *split()* method of JavaScript that converts any string into an array of substrings. It is a method of *String* class and just takes a delimiter as its argument and based on that delimiter it divides the string into an array of substrings (See Figure 17).

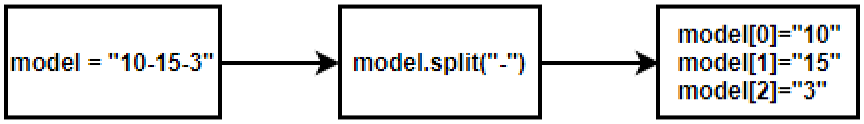


Figure 17. Split method of JavaScript

A string from an array of substrings can be achieved by simple concatenation. Thus, from a grid-based data handling to a model-based data handling has been a smooth transition in the classes related to visualization and analytics. *Simulation (Grid) Layer* and *Transition Heatmap Chart* are specific to the Cell-DEVS models whereas *Model Track Chart* and *State Frequency Chart* are applicable for both Cell-DEVS and Classic DEVS models. Auto Classes of Cell-DEVS specific models undergone to the three-dimensional changes using split and concatenation of String as discussed above. However, *Auto.CellTrackChart* class has not been changed except a small modification in the *UpdateSelcted()* method where the function now returns the selected cell in terms of model id in place of a coordinate value as it was done in the previous version.

The *State Frequency Chart* shows the frequency of each selected state value in the form of a bar chart. The class related to this module, *Auto.StateChart* is modified for a method *Data()* in which the value of each grid used to be calculated through an algorithm with time complexity of O[n^3] that has been now changed to O[n] time complexity because of the change in the data structure of the *State* object. The nested For loop used for traversing three-dimensional arrays is now eliminated by a single For loop where the value of *States* can be traversed using the model id.

```
    Select(x, y, z) {
        this.selected.push({ x:x, y:y, z:z });
        this.Emit("Change", { x:x, y:y, z:z, selected:true });
    }
```

Methods of *Selection* class, deeply associated with most of the visualization and analytics classes through *Simulation* class, has been changed to transact model ids (mod variable for model id in the below code) instead of grid coordinates (see above code-snippet).

```
    Select(mod) {
        this.selected.push( mod );
        this.Emit("Change", { mod, selected:true });
    }
```

Apart from these transitions discussed above, few additional but small changes, required to make it a generic DEVS Viewer, have also been done in other parts of the previous code of the WebViewer.

3.2 New parser for Classic-DEVS

Introducing a Classic-DEVS parser and visualizing its simulation are important contributions towards this project. Starting from 'developing a new parser subclass for Classic DEVS simulation results' to 'making its data available to render from the *Simulation* class' have been taken care of during the course of this project development. However, visualization of the results was out of the scope and has been handled by another colleague.

Cell-DEVS WebViewer, a web-based application, is there as a visualization tool for Cell-DEVS models from CD++ and Lopez simulators. Whereas there was no visualization tool offered to visualize and analyze the simulation results of classic DEVS models. Thus, in this version of the WebViewer, a parser for classic DEVS models has been included.

Any *Parser* class to be detected must be added in a constant array named PARSERS in the *Sim* class.

```
const PARSERS = [DEVS, CDpp, RISE];
```

As soon as the classic DEVS files (.log and .ma files) are added, the *DetectParser()* method of *Sim* class is to be called. This method picks the parsers mentioned in the above array and tries to detect one by one via Promise call to handle asynchronous action. The *DetectMethod()* eventually calls *IsValid()* method of each parser class to detect the parser type. The .log file belongs to which parser is detected through the algorithm shown in Figure #.

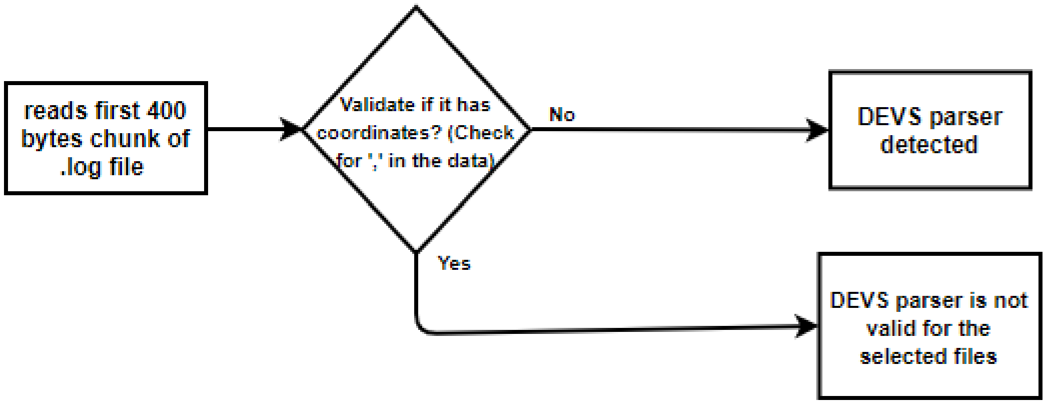
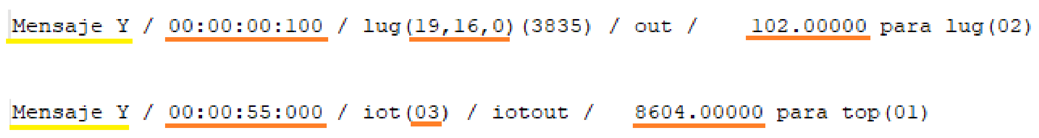

reads first 400 bytes chunk of .log file
Validate if it has coordinates? (Check for ',' in the data)
No
DEVS parser detected
Yes
DEVS parser is not valid for the selected files

Figure 18. Validation of DEVS .log files, detection of parser; *IsValid()*

If the log file contains the simulation result of Classic DEVS models, the id of each model coming from the simulator would be in a one-dimension format, not as a three-dimensional or two-dimensional format. The output data of .log files of Cell-DEVS and Classic-DEVS are compared in Figure 18. It can be clearly seen, both look almost similar, however, there is a difference in model id. The model id is (19,16,0) for the *lug* model (Cell-DEVS) whereas it is (03) for the *iot* model (Classic-DEVS). Thus, looking for a "," in the .log data is the key to detect the file from a simulator of the Classic-DEVS models as shown in Figure 19.



```
Mensaje Y / 00:00:00:100 / lug(19,16,0)(3835) / out /    102.00000 para lug(02)

Mensaje Y / 00:00:55:000 / iot(03) / iotout /   8604.00000 para top(01)
```

Figure 19. Comparison of .log file data of Cell-DEVS (line 1) with that of Classic-DEVS (line 2)

Once the parser is detected, the parsing of the simulation files starts when the user clicks the *Load Simulation* button. *DEVS* parser only focuses on parsing the .log file and the parsing strategy is pretty similar but much simpler than other parsers. The lines with ‘Mensaje Y’ are considered to be the output data as seen in line 1 of Figure 19.

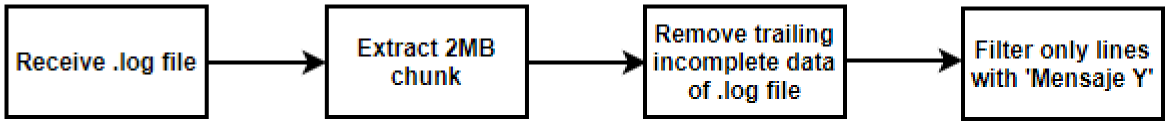

Receive .log file
Extract 2MB chunk
Remove trailing incomplete data of .log file
Filter only lines with 'Mensaje Y'

Figure 20. Parsing Strategy to filter Output lines from .log file – DEVS Parser

Once the output data is filtered from the .log file (See Figure 20), chunk by chunk, the required information (as highlighted in line number 2 of Figure 19) will be parsed and stored inside *Frame* and *Model* object. The algorithm (See in Figure 21) of the creation of *Frame* objects and *Model* objects are similar as used in any other parsers.

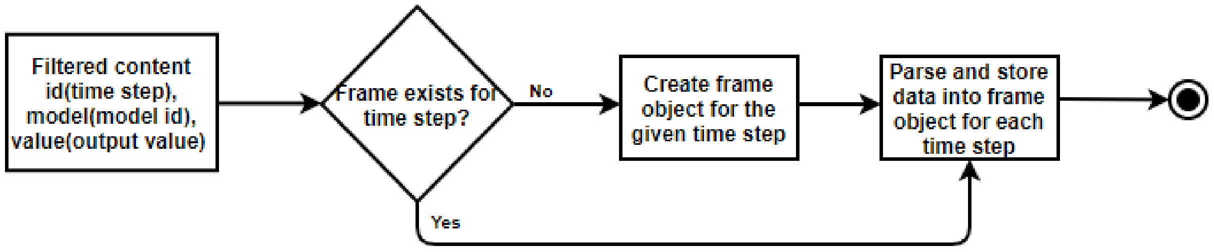
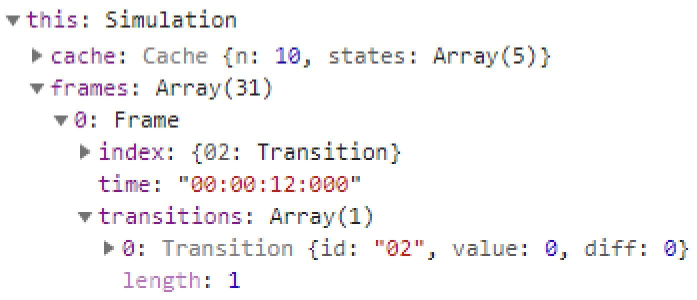

Filtered content id(time step), model(model id), value(output value)
Frame exists for time step?
No
Yes
Create frame object for the given time step
Parse and store data into frame object for each time step

Figure 21. Frame Creation in Classic-DEVS Parser

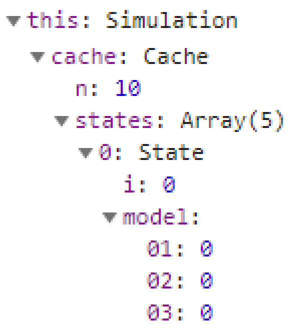
It was easier to parse Classic-DEVS as compared to Cell-DEVS because no coordinates involved here. For each array of output lines, the *Frame* objects are created at each time step, and transitions with model id and its value are added into the frames (See Figure 22). The data structure of *Frame* and *Transition* objects have been consistent as considered for the generic application.



```
▼this: Simulation
  ▶cache: Cache {n: 10, states: Array(5)}
  ▼frames: Array(31)
    ▼0: Frame
      ▶index: {02: Transition}
       time: "00:00:12:000"
      ▼transitions: Array(1)
        ▶0: Transition {id: "02", value: 0, diff: 0}
         length: 1
```

Figure 22. Data Structure of Frame & Transition Object [Classic-DEVS] in DEVS-WebViewer

The data structure of all the objects- components of *Simulation* object; be it *Frames, Transitions, States, Cache*, or any other object has been common throughout irrespective of parser type. The dimensions of models are hiding inside the *Model* object as a string value; the format of value inside the string is different; "x-y-z" for Cell-DEVS and "id" for Classic-DEVS (See Figure 23).



```
▼this: Simulation
  ▼cache: Cache
     n: 10
    ▼states: Array(5)
      ▼0: State
         i: 0
        ▼model:
           01: 0
           02: 0
           03: 0
```

Figure 23. Data Structure of Cache & State Object [Classic-DEVS] in DEVS-WebViewer

However, from the *Simulation* class and the application's perspective, both are just model ids. Hence, the objective of making a generic application for the DEVS simulation models is achieved.

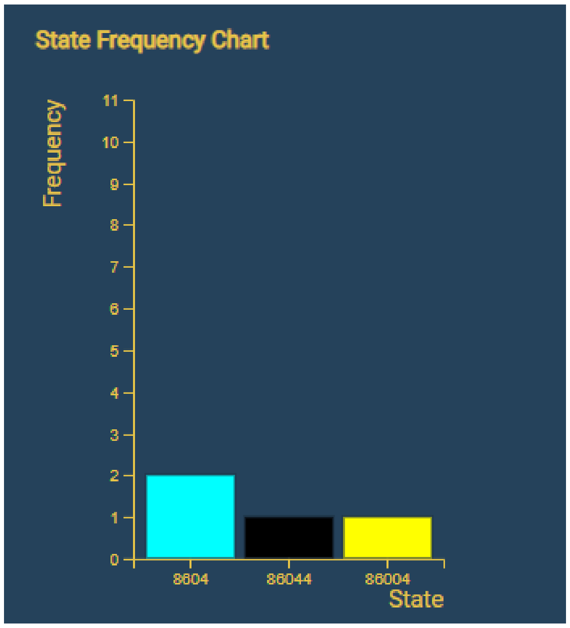
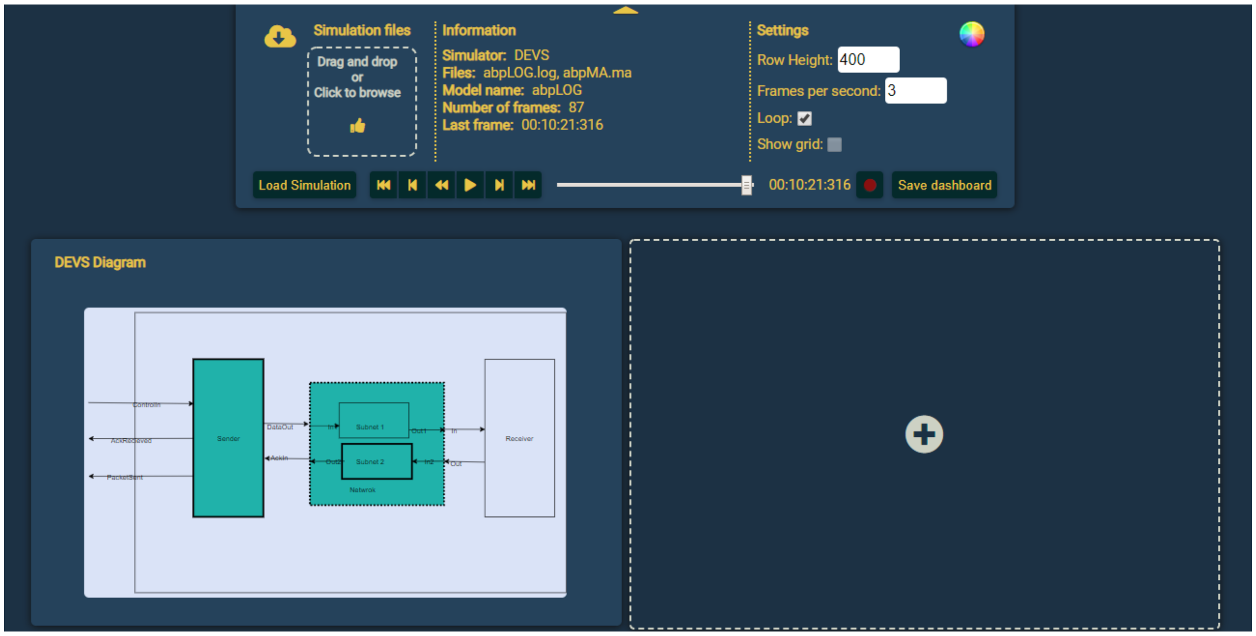


Figure 24. State Frequency Chart for Classic-DEVS (IoT model) in DEVS-WebViewer

The visualization tool for *StateFrequency Chart* is completely adapted with the new Classic-DEVS parser and showing the results as seen in Figure 24. Since there is no palette file uploaded as part of the simulation files for the Classic-DEVS model, the *Palette Editor* tool of the application is used from the *Dashboard* of the application.

Once the project is delivered, it is integrated with the other code of SVG animation to showcase the visual representation of Classic DEVS models as the simulation runs from the *Play Forward* button. This integrated application (DEVS-WebViewer) asks for an SVG image of the model to visualize the DEVS Diagram. It is observed that the SVG image is successfully rendering the required output data of the file (Seen in Figure 25) after the final code integration.



Simulator: DEVS
Files: abpLOG.log, abpMA.ma
Model name: abpLOG
Number of frames: 87
Last frame: 00:10:21:316

Row Height: 400
Frames per second: 3
Loop: ☑
Show grid: ☐

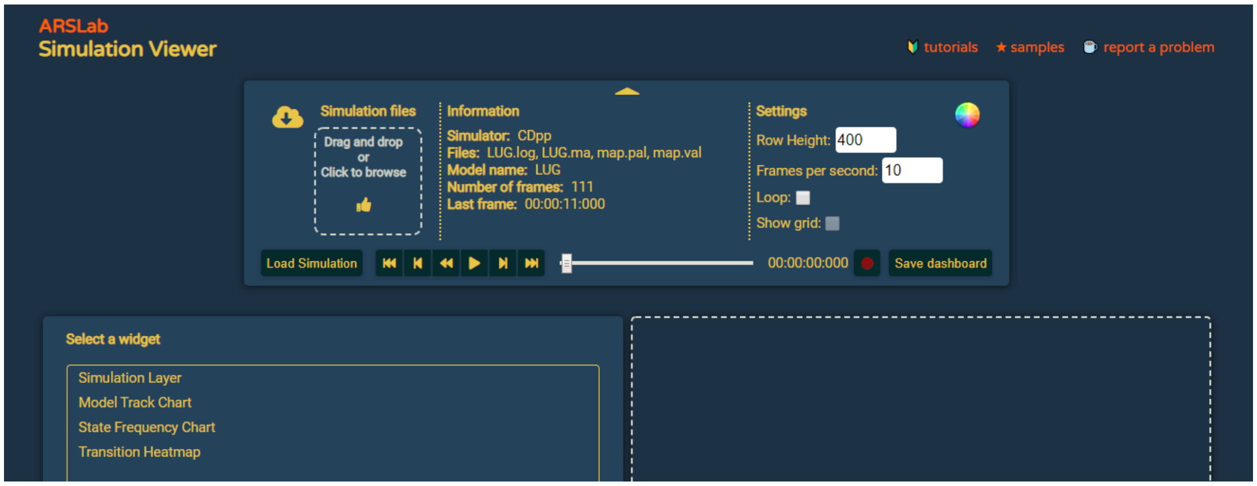
00:10:21:316

Figure 25. Visual Representation of ABP model; DEVS Diagram of a Classic-DEVS model [*Image Courtesy: DEVS Simulation Viewer tool after merging the codes*]

3.3 Additional Features

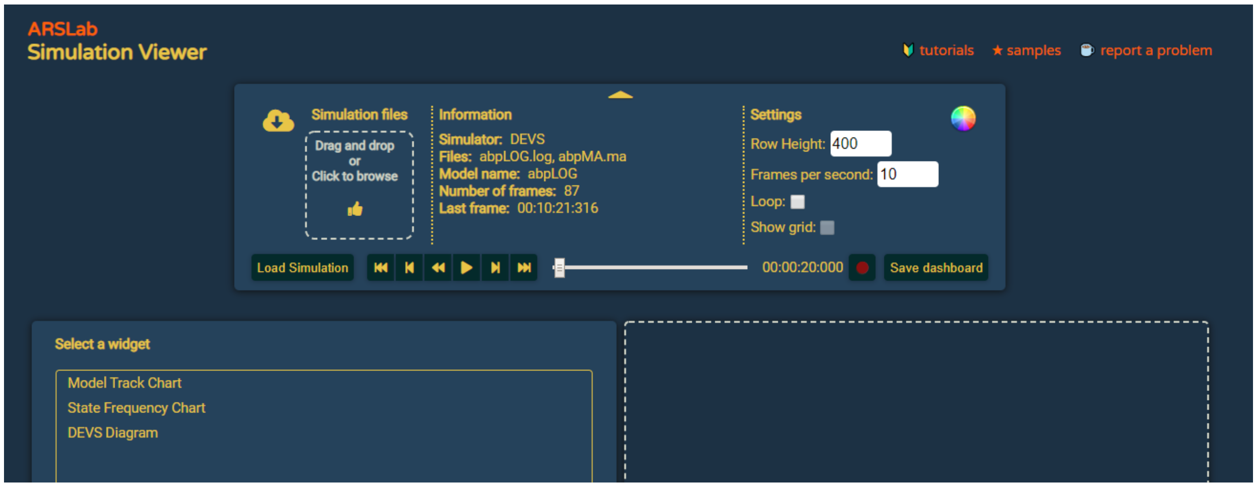
After accomplishing the objectives of the project, a couple of elements have also been added as part of additional features. These features are small enhancements and not mentioned in the project proposal. These are mainly user interface level changes and not much associated with the core development of the tool. Since the goal of building this web-application is also to enrich the user experience, therefore these additional elements can be considered as crucial as the functional aspect of the application.

One of the essential components to feature out is to restrict the visualization options depending on the type of model to be simulated. Cell-DEVS models should have the widget options to be selected from *Simulation Layer*, *Model Track Chart*, *State Frequency Chart*, *Transition Heatmap*. On the other hand, the Classic DEVS models should be analyzed through *Model Track Chart*, and *State Frequency Chart*. After the integration with the SVG code to animate the Classic DEVS simulation result, one new widget *DEVS Diagram* is also included. Figure 26 shows the different options in Select a Widget section when simulation files of different model types are loaded.



Simulator: CDpp
Files: LUG.log, LUG.ma, map.pal, map.val
Model name: LUG
Number of frames: 111
Last frame: 00:00:11:000

Row Height: 400
Frames per second: 10
Loop:
Show grid:



Simulator: DEVS
Files: abpLOG.log, abpMA.ma
Model name: abpLOG
Number of frames: 87
Last frame: 00:10:21:316

Row Height: 400
Frames per second: 10
Loop:
Show grid:

Figure 26. User Interface of Widget Selector for Cell-DEVS model (top) and Classic-DEVS model (bottom) [*Image Courtesy: DEVS Simulation Viewer tool after merging the codes*]

These components are separate out inside the *Auto* class of the application. These widget options are listed in the array variables of objects. One array was already there for Cell-DEVS models, another array of objects with required widgets for Classic DEVS has now been added. Depending upon the *Simulator* type from the *Information* section of the *Dashboard* UI, it returns the array of objects and that's how the list of widgets is shown accordingly to the user.

Another feature is also included to enable the user to choose a theme out of the available list of themes for the application. Nowadays, most of the applications and also few mobile operating system platforms provide this cool feature of selecting the mode (dark/light) or theme to enhance the comfortability in the direction of user experience. Most of these themes offer a good number of layout options. It is a wonderful option of adjustment for the users with colorblindness, they can pick the theme in accordance with their comfort.

These themes are entirely developed through CSS coding. There are four themes (theme1, theme2, theme3, theme4) already available as part of the CSS code of the application. To switch the theme, it was required to change the theme name in the index.html file. In this version, it has been brought at the front in the form of a drop-down from where user can easily access to change the theme as per the choice. The HTML code for the theme selector is included in the *Template* of the *Widget.Header* class. It has been a straightforward approach to make a small change in the code by using JavaScript's feature of updating the DOM (Document Object Model) content and changing the behavior. As the user selects a theme option from the dropdown, an event listener is triggered and the path of the theme's css file is modified as per the selected theme. This DOM content modification is done by changing the path under the link tag dedicated to the theme stylesheet of index.html. The different UIs of the WebViewer for the four themes are presented in the APPENDIX section of this paper (See Figure 27-31).

The *nls* object (of nls.js file) is also modified to make sure to have the UI component's names in the application as per the generic DEVS WebViewer. For instance, the name of the application is changed from *CD-WebViewer* to *Simulation Viewer*. *Cell Track Chart* is modified to *Model Track Chart*, etc. These name changes are handled by updating the *nls* object. A couple of modifications and few errors fixing was also been done when the application gets integrated with the code of the DEVS Diagram feature.

4. CONCLUSION

In this paper, an improved and a generic version of DEVS WebViewer is presented. This new version of the web-application follows the same methodology of object-oriented development as it was done in the previous version. The main objective of changing the grid-based application to a model-based application has been achieved by making it a generic DEVS WebViewer application from a Cell-DEVS version.

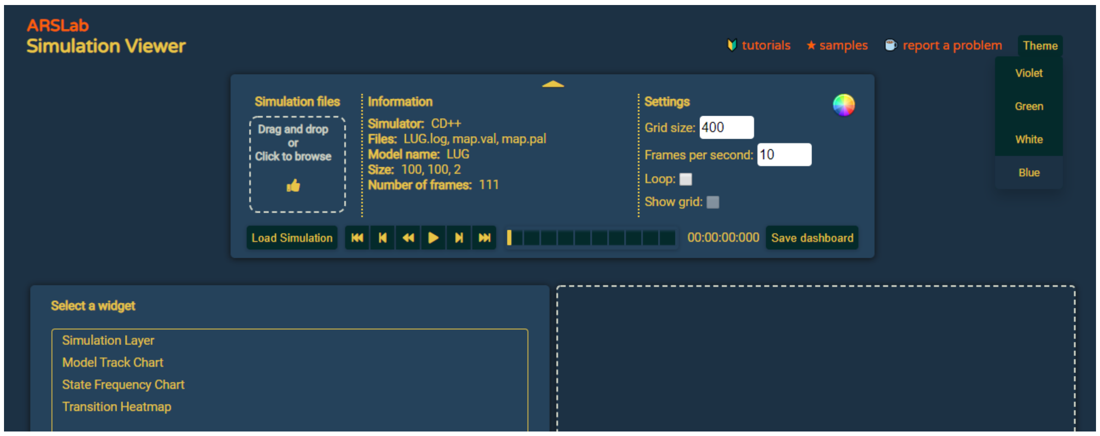
Elimination of the unnecessary classes and modifying the structure to make it flexible towards other DEVS models have been done while transitioning the architecture. The process of creating the frames has been simplified, as there is no more dependency on any message object, they are created as soon as parsing the result file of simulation. Major changes include the modification of the data structure of the simulation class and its all associates classes. The new data structures of different objects from various classes contain model objects instead of three-dimensional grid arrays that make it simpler to use. The unused attributes are also removed from these data structures. One of the significant improvement is achieved in terms of time complexities for a couple of algorithms, especially the ones handling the state objects. The substantial decline from $O[n^3]$ to $O[n]$ is attained due to the data structure modification. The changes in the visualization components of the application have been more of string manipulations to decompose model ids into coordinates and vice-versa, for cell-DEVS visualizations.

The addition of a separate DEVS parser class, to parse the simulation results of Classic DEVS models, has been one of the key contributions of the project. A new parser is necessary to have because of the different formats of data in the result. The visualization is seemed to work flawlessly for DEVS models when analyzed using State Frequency Chart. However, the visual representation of simulation is handled by another teammate in an SVG diagram of the Classic DEVS model.

A couple of UI level enhancements are also included as part of the additional features of the application. Depending on the type of simulator, the application allows the user to see only the relevant widgets for visualization and analysis. Furthermore, thinking about the improvement of user experience a feature to select a theme mode out of four modes has been added as a dropdown in the header section of the web-application UI.

The WebViewer is under process as a couple of students are working for further advancements. In the near future, a parser for Cadmium simulations will also be included in the application being the platform for a generic DEVS Viewer. A generic WebViewer is the foundation for making a giant DEVS Environment by the integration with the improved version of the RISE platform. The cloud-based application will provide an architecture with modeling and simulation as a service. There are future possibilities to develop standalone reusable DEVS APIs that would allow developers to build domain-specific applications.

APPENDIX

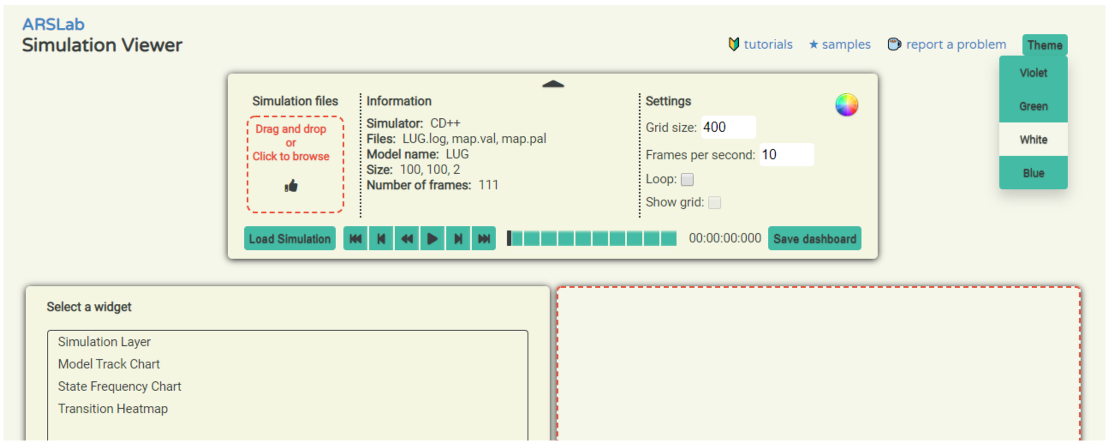


Simulator: CD++
Files: LUG.log, map.val, map.pal
Model name: LUG
Size: 100, 100, 2
Number of frames: 111

Grid size: 400
Frames per second: 10
Loop:
Show grid:

00:00:00:000

Figure 27. WebViewer UI for theme Blue mode

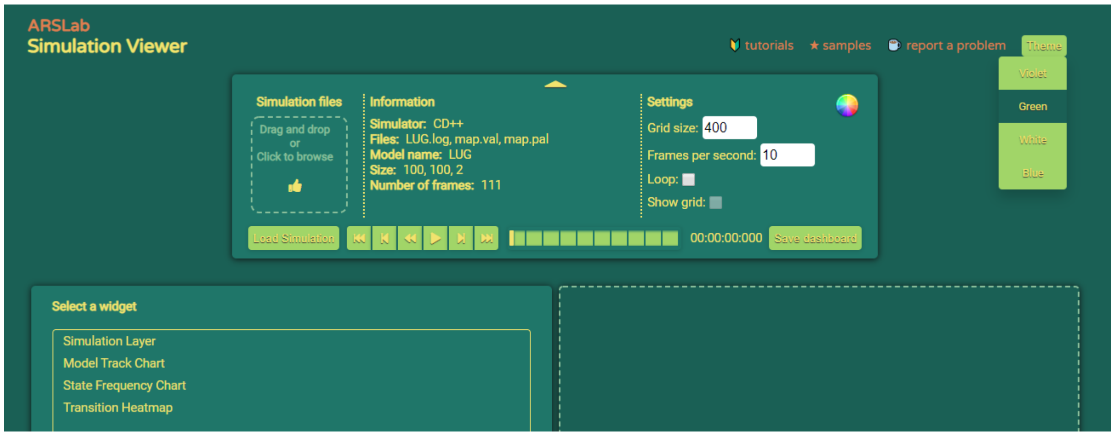


Simulator: CD++
Files: LUG.log, map.val, map.pal
Model name: LUG
Size: 100, 100, 2
Number of frames: 111

Grid size: 400
Frames per second: 10
Loop:
Show grid:

00:00:00:000

Figure 28. WebViewer UI for theme White mode

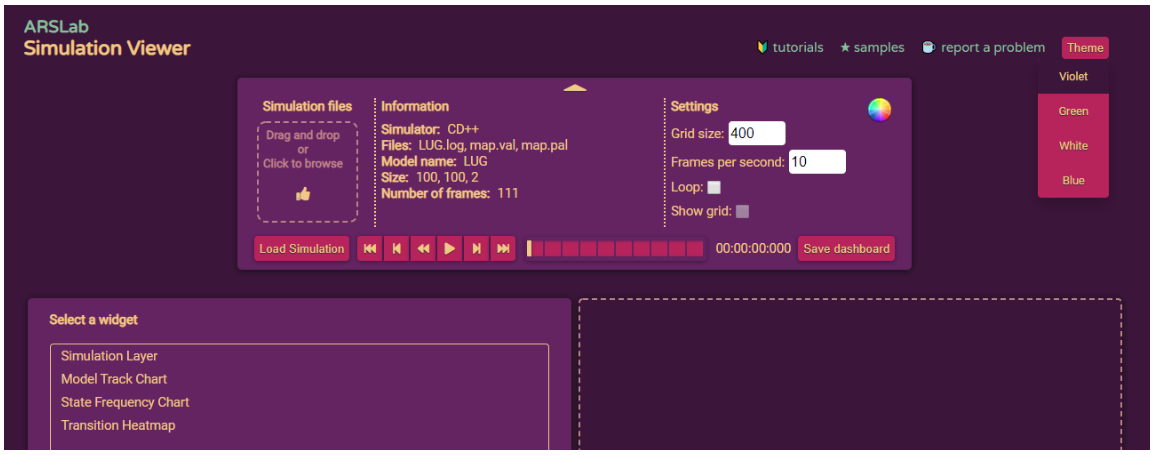


Information
Simulator: CD++
Files: LUG.log, map.val, map.pal
Model name: LUG
Size: 100, 100, 2
Number of frames: 111

Settings
Grid size: 400
Frames per second: 10
Loop:
Show grid:

00:00:00:000

Figure #. WebViewer UI for theme Green mode



Information
Simulator: CD++
Files: LUG.log, map.val, map.pal
Model name: LUG
Size: 100, 100, 2
Number of frames: 111

Settings
Grid size: 400
Frames per second: 10
Loop:
Show grid:

00:00:00:000

Figure 29. WebViewer UI for theme Violet mode

DELIVERABLE AND TOOL USAGE

The source code of the viewer is available on my GitHub page. The same source code is available under DEV-Navneet branch of the Advanced Real-time Simulation Laboratory's GitHub page. The final integrated version is available at ARS Lab's GitHub.

How to use the tool?

- Visit the Generic DEVS WebViewer tool
- Get ready with the simulation files. (Download folder of any model from ./log/ folder)
- Upload the simulation files from the downloaded log folder
- Click on "Load Simulation" button

- Based on the DEVS model type, it will show the visualization options to select from Widget
- Visualize and Analyze the simulation results

Note: *To visualize the Simulation of Classic-DEVS Model, please use the integrated version. Upload the simulation files of ABP model from the log folder. And follow the same steps as above. The log files can be downloaded from any of the Github repositories*.

REFERENCES

St-Aubin, B., O. Hesham, and G. Wainer, 2018, "A Cell-DEVS Visualization and Analysis Platform," in *Proceedings of the 50th Computer Simulation Conference*, Bordeaux, France.

Al-Zoubi, K., and G. Wainer, 2015, "Distributed Simulation of DEVS and Cell-DEVS models using the RISE middleware," in *Simulation Modelling Practice and Theory*, vol. 55, pp. 27–45.

Bergero, F., and E. Kofman, 2011, "PowerDEVS: A tool for hybrid system modeling and real-time simulation," *Simulation*, vol. 87, no. 1–2, pp. 113–132.

Bostock, M., Ogievetsky, V., Heer, J., 2011, D3: "Data-Driven Documents", *IEEE Trans. Visualization & Comp. Graphics* (Proc. InfoVis).

Chidisiuc, C., Wainer, G., 2007, "CD++ Builder: An Eclipse-based IDE for DEVS modeling", *Proceedings of the SpringSim'07 Simulation Multiconference*, Vol 2., 235-240.

Collins, A., Knowles, B., 2013, "Philosophical and Theoretic Underpinnings of Simulation Visualization Rhetoric and Their Practical Implications", *Ontology, Epistemology, and Teleology for Modeling and Simulation: Philosophical Foundations for Intelligent M&S Applications*, vol. 44, pp. 173-191.

Dale, K., 2016, "Data Visualization with Python and JavaScript"

Goldstein, R., S. Breslav, and A. Khan, 2016, "DesignDEVS: Reinforcing theoretical principles in a practical and lightweight simulation environment," in *Proceedings of the Symposium, on Theory of Modeling & Simulation*, TMS/DEVS 2016, Pasadena, CA, USA.

Goldstein, R., S. Breslav, and A. Khan, 2018, "Practical aspects of the DesignDEVS simulation environment," *Simulation*, vol. 94, no. 4, pp. 301–326.

Microsoft Developer Network (MSDN), 2017, “SVG vs Canvas: How to Choose, https://msdn.microsoft.com/en-us/library/gg193983(v=vs.85).aspx, consulted on Dec 16, 2017.

Sarjoughian, H. S., and V. Elamvazhuthi, 2009, “CoSMoS: a visual environment for component-based modeling, experimental design, and simulation,” in *Proceedings of the 2nd International Conference on Simulation Tools and Techniques*, Rome, Italy.

St-Aubin, B., Hesham, O., and Wainer, G., 2018, “A Cell-DEVS Visualization and Analysis Platform,” in *Proceedings of the 50th Computer Simulation Conference*, Bordeaux, France.

St-Aubin B., Yammine E., Nayef M., Wainer G., 2019, July 22-24, *SummerSim-SCSC*, Berlin, Germany; ©2019 Society for Modeling and Simulation (SCS) International.

Van Schyndel, M., Hesham, O., Wainer, G., 2016. “Crowd Modeling in the Sun Life Building”, *Proceedings of the Symposium on Simulation for Architecture & Urban Design* (SimAUD '16). SCS.

Vangheluwe, H.L.M., 2000, “DEVS as a Common Denominator for Multi-formalism Hybrid Systems Modelling”, *IEEE International Symposium on Computer-Aided Control System Design*.

Vangheluwe, Hans, 2005, ” The Discrete EVent System specication (DEVS) formalism”

Wainer, G., 2009. "Discrete-event modeling and simulation: a practitioner's approach." CRC press.

Wainer, G., 2014, "Cellular modeling with Cell-DEVS: A discrete-event cellular automata formalism." *International Conference on Cellular Automata*. Springer, Cham.

Wainer, G., and Giambiasi, N., 2002, “N-dimensional Cell-DEVS Models,” in *Journal of Discrete Event Dynamic Systems*, vol. 12, no. 2, pp. 135–157

Wainer, G., and S. Wang, 2017, “MAMS: Mashup architecture with modeling and simulation as a service,” in *Journal of Computational Science*, vol. 21, pp. 113–131.

Wikipedia, Data-Driven Document (D3.js), https://en.wikipedia.org/wiki/D3.js

Zeigler, B.P., Praehofer, H., Kim T.G., 1976, “Theory of Modeling and Simulation”, New-York: Wiley-Interscience.