Design of WebViewer -3.0
For
Visualization of Classic DEVS results using Scalable Vector Graphics

Systems Engineering Project Report

SYSC 5900

Submitted to:

Submitted Gabriel Wainer
Department of Systems and Computer Engineering
Carleton University
1125 Colonel By Drive
Ottawa, ON, Canada
gwainer@sce.carleton.ca

Submitted by
Ankit Kumar Vig
101120055
Department of System and Computer Engineering
1125 Colonel By Drive
Ottawa, ON, Canada
ankitvig@cmail.carleton.ca

Table of Contents

1.Introduction

1.1. Project Overview

DEVS is a popular formalism for modelling complex dynamic systems using a discrete-event abstraction. The project focuses mainly on designing a client-side application for simulation visualization and analysis platform for Classic DEVS models. The Classic DEVS models can be easily designed in using CD++ tool. However, visualizing the input, output events and changes in the model states can only done by evaluating the data. There is no way till now to visualize the instantaneous changes in the states. Considering the presence of large amount of data and different models there are chances that certain state changes or inputs outputs can be neglected. To give a better picture to the modeler and for the better analysis of the results.

The main aim of the WebViewer 3.0 (the latest version) is to visualize and analyze the simulation results of Classic DEVS. The aim is to make WebViewer (precious version) more generic not only representing just the Cell DEVS simulation results and to improve user interactivity.

The major part of WebViewer 3.0 focuses mainly on Classic DEVS results. The tool gives freedom to the user to use any of their desired means to design DEVS Diagram using SVG to visualize their DEVS models where they can be creative and design their models as per their requirement for the specific DEVS model. Then the user's SVG is mapped with the simulation result LOG from CD++ in order to visualize the results of Classic DEVS.

The tool is developed entirely in HTML5 and JavaScript, requires no separate installation, can be run locally on a user's computer and minimizes external dependencies. It allows users to visualize and animate their simulation results, to navigate to different time steps, record videos of their simulation, inspect the state of individual model, and export the raw data in JSON for further processing in other external programs. It leverages the data-driven document (D3) JavaScript API to provide statistical analysis capabilities in the form of animated charts that display data derived from the simulation as it is being executed.

This report does not contain any sample code from the WebViewer. The latest code is made available at GitHub repository. https://github.com/AnkitVig/CDWeb--3.0 and at

ARS Lab : https://github.com/SimulationEverywhere/CD-WebViewer-2.0/tree/DEV-Ankit

To run application use link: https://ankitvig.github.io/CDWeb--3.0/

Demo video to run Visualization: https://youtu.be/-XCs9xhjOSQ

1.2. Motivation

Visualizing something is always more impactful as compare to the numerical data. It helps to increase the userbase and makes it easier to explain the idea behind the experiments. Visualizing the state and outputs of the DEVS model was a complicated task.

There are many applications for of a DEVS model. People design models for simulating various real-time applications. There are various tools design just to model the applications. However, none of them focus on the ideology of visualizing the results. The CD Web viewer is one of the tools where users can view their simulated results. The major focus of the CD web viewer is viewing the cell DEVS results. By taking the same concept as a benchmark we aim towards designing the generic web viewer with minimum dependency on any external libraries and software. Which gives the advantage to the user to visualize the results as per his convenience. discrete-event abstraction. The Discreet event abstraction level, a timed sequence of events input to a system (or internal, in the case of timeouts) cause instantaneous changes to the state of the system. We focus mainly on visualizing the classic DEVS results by using SVG image for animating and highlighting the input, output and the changes in the state of different models. events as per the simulated logs for all the atomic and the coupled models.

1.3. Scope of Work

The WebViewer is developed entirely in HTML5 and JavaScript, requires no separate installation, can be run locally on a user's computer and minimizes external dependencies. Improving communication and collaboration with domain experts and stakeholders through a more interactive means of data visualization than traditional static media like images or videos. The major application of the WebViewer is to visualize Cell DEVS simulation results only. There are several improvements that are required to increase the user's interactivity. The updated version of the WebViewer is inclined towards the idea of making the WebViewer more generic and interactive. Aim is to make the WebViewer enable to process different simulation results without any dependency of external libraries. The new features added concentrate towards the Classic DEVS results. The major challenge in viewing classic DEVS results is to them on a web page which unlike cell DEVS cannot be represented using a grid. So, to resolve this issue the SVG is used to represent a classic DEVS model into the system.

2. Background

2.1. Classic DEVS

DEVS is a common denominator for multi-formalism hybrid systems modelling. The structure and the behaviour of different models is defined using DEVS formalism. The DEVS methodologies can be used to design different real-life models which are expensive or impossible to experiment. DEVS can be used to model different case scenarios for a model which helps in analysing all aspects of the experiment. DEVS model is one that can represent any system that produces a finite number of state transitions in a finite amount of time. DEVS allows for the description of system behaviour at two levels. At the lowest level, an atomic DEVS describes the autonomous behaviour of a discrete-event system as a sequence of deterministic transitions between sequential state as well as how it reacts to external input (events) and how it generates output (events). At the higher level, a coupled DEVS describes a system as a network of coupled components. An atomic model has two transition functions an external transition function and an internal transition function. Each model has a persistent state. As soon as the input is received the external transition function is invoked. The internal transition function is triggered with the output function. Between events, the state does not change, resulting in a constant state trajectory. The time advance is always monitored till the external transition function changes or time advance function expires. whichever occurs first. In the latter case, the output function λ, also dependent on the current state, will issue an output value y before triggering an internal state transition function δint. In turn, the internal transition function will update the state s and a new ta is calculated. The model will execute this loop until it passivates (i.e. ta is set to infinity). Without any further external inputs, the model will remain passive (St-Aubin, Hesham, Wainer 2018). The combination of different DEVS atomic models or the models which are part of same sub system, makes a coupled model. The connections denote how components influence each other. Output events of one component can become, via a network connection, input events of another component. [Zei84a].

2.2. DEVS Formalism

The DEVS formalism is a set of conventions introduced in 1976 for the specification of discrete event simulation models. The DEVS formalism fits the general structure of deterministic, causal systems in classical systems theory. Due to the modular and hierarchical modeling views, as well

as its simulation-based analysis capability, the DEVS formalism and its variations have been used in many applications of engineering. DEVS is primarily used for the simulation of queueing networks and performance models. It is most applicable for the modelling of discrete event systems with component-based modularity. It can, however, be used much more generally as a simulation assembly language, or as a theoretical foundation for these formalisms. Since the state in a DEVS model is updated in an event-driven fashion, processing cycles are not wasted on uneventful time periods of the simulation. This contrasts with discrete-time simulation where the simulation is advanced in discrete timesteps regardless of the underlying state changes. Thus, DEVS generates the minimum amount of output data required for visualization and analysis of all state updates. (St-Aubin, Hesham, Wainer 2018).

DEVS and Object-Oriented Programming

Support for modular and hierarchical model design is one of the DEVS formalism's most compelling attributes. It is often remarked, however, that widely adopted object-oriented programming practices provide the same benefits. Technically, object-orientation and DEVS are not alternatives to one another. Many simulations have been implemented using object-oriented programming features in conjunction with DEVS conventions. That said, the analogy between object-oriented classes and DEVS models deserves some discussion. In an objected-oriented language, classes include methods that take arguments, deliver return values, and reassign member variables. Similarly, DEVS models include transition functions that take inputs, deliver outputs, and reassign state variables. The difference is that these transition functions also depend on the simulated time elapsed since the previous event. It is possible to include this temporal information in object-oriented code, but it is not the norm. The other major difference between object-orientation and DEVS is that, with the former, it is common practice to design classes that reference one another explicitly.

There are many other types of DEVS formalism such as:

- DEVS with simultaneous EVENTS (Parallel DEVS)
- Dynamic Structure DEVS
- Quantized DEVS
- Generalized DEVS (GDEVS)

An Example: ABP DEVS model

ABP (Alternating Bit Protocol) is a communication protocol to ensure reliable transmission through an unreliable network. The sender sends a packet and waits for an acknowledgment. If the acknowledgment doesn't arrive within a predefined time, the sender re-sends this packet until it receives an expected acknowledgment and then sends the next packet. In order to distinguish two consecutive packets, the sender adds an additional bit on each packet (called *alternating bit* because the sender uses 0 and 1 alternatively). A DEVS model called "ABP Simulator" is created to simulate the behavior of the Alternating Bit Protocol.

The ABP Simulator consists of 3 components: sender, network and receiver. The network is decomposed further to two subnets corresponding to the sending and receiving channel respectively.

The behavior of receiver is to receive the data and send back an acknowledgment extracted from the received data after a time period. The subnets just pass the received data after a time delay. However, in order to simulate the unreliability of the network, only 95% of the data will be passed in each of the subnet, i.e. 5% of the data will be lost through the subnet.

The receiver and subnets have two phases: *passive* and *active*. They are in *passive* phase initially. Whenever they receive input, they will be in *active* phase, and send out an output (with a probability of 95% in subnet) after a time duration. The state will then be changed back to *passive* phase. The *receiveing_time* of the receiver is a constant while the *delay* in subnets is non-deterministic value expressed by a normal distribution with a mean and deviation.

The behavior of the sender is much more complicated. Its state depends on the following user defined state variables: *Alt_bit*, *sending*, *ack*, and *packetNum* in addition to *phase*. The sender changes from initial phase *passive* to *active* when a *controlIn* signal is received. It is then in *sending* mode to send a packet plus an alternating bit. When a *sending_time* is elapsed, the packet is assumed to be sent out, and the sender is waiting for the acknowledgment. If the *timeout* expires, the sender will re-send the previous packet with the alternating bit. If the expected acknowledgment is received before the *timeout*, the sender will send the next packet. It will change back to *passive* phase when all packets have been sent out successfully. Output will be generated when a packet is sent out (*packeSent, dataOut*) or an expected acknowledgment is received (*ackReceived*). For simplicity, the packet sent out by the sender is just the packet sequence number plus an alternating bit, (e.g. 11 for the first packet, 100 for the 10^{th} packet etc.). Thus the

packet sequence number (e.g. 1 for the first packet, 10 for the 10^{th} packet etc.) is sent to the *packetSent* port while the packet sequence number plus the alternating bit (e.g. 11 for the first packet, 100 for the 10^{th} packet, etc.) are sent to the *dataOut* port. The *controlIn* signal is a positive integer indicating how many packets should be sent in a session. Both the *sendint_time* and *timeout* are constants.

ABP Simulator used as an example here, to read logs of the ABP simulation results and display visualisation results by designing DEVS diagram using SVG.

2.3. Importance of Modeling and Simulation

Model is a representation of a real system into a virtual system which helps in predicting the effects of the changes to the system model is created in such a way so as all the properties of each part are included. Simulation of a system is the operation of a model in terms of time or space, which helps analyze the performance of an existing or a proposed system. The results so obtained by the simulation are then validated with the actual model, a model is valid only if the model is an accurate representation of the actual system, else it is invalid. Simulation models consist of the following components: system entities, input variables, performance measures, and functional relationships. Developing a model is beneficial in cases where it helps us to understand how the system really operates without working on the real time system. The user can configure his model for analyzing different case scenarios and allow the user to monitor their effects on the outputs without working on the real time system. The system requirements can be determined by reconfiguring the same model. Certain systems are so complex that it is not easy to understand their interaction with other models and the state of the model at a given time. However, Modelling & Simulation allows to understand all the interactions and analyze their effect. Additionally, new policies, operations, and procedures can be explored without affecting the real system. Modelling & Simulation can be applied to the following areas − Military applications, training & support, designing semiconductors, telecommunications, civil engineering designs & presentations, and E-business models.

Additionally, it is used to study the internal structure of a complex system such as the biological system. It is used while optimizing the system design such as routing algorithm, assembly line, etc. It is used to test new designs and policies. It is used to verify analytic solutions.

2.4. WebViewer Overview

The Cell-DEVS WebViewer is a minimalistic client-side web application built entirely in HTML5 and JavaScript. It requires only two external dependencies, Whammy.js to record videos of canvas animations in. webm format and the Data-Driven Documents (D3) library for dynamic charts. In this section, the user interface, the data structures and the general flow of the application will be discussed. 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). It is meant to offer a better loading process for the simulation results, more control over the visualization and more user-friendly visualization of results (St-Aubin, Hesham, and Wainer). The Cell-DEVS WebViewer is entirely built with HTML5 and JavaScript.

Client-side JavaScript is the most common form of the language. The script should be included in or referenced by an HTML document for the code to be interpreted by the browser. It means that a web page need not be a static HTML, but can include programs that interact with the user, control the browser, and dynamically create HTML content.

The JavaScript client-side mechanism provides many advantages over traditional CGI server-side scripts. For example, you might use JavaScript to check if the user has entered a valid e-mail address in a form field. The JavaScript code is executed when the user submits the form, and only if all the entries are valid, they would be submitted to the Web Server. JavaScript can be used to trap user-initiated events such as button clicks, link navigation, and other actions that the user initiates explicitly or implicitly.

HTML 5 is the fifth and current version of HTML. It has improved the markup available for documents and has introduced application programming interfaces (API) and Document Object Model (DOM). The latest versions of Apple Safari, Google Chrome, Mozilla Firefox, and Opera all support many HTML5 features and Internet Explorer 9.0 will also have support for some HTML5 functionality. The mobile web browsers that come pre-installed on iPhones, iPads, and Android phones all have excellent support for HTML5.

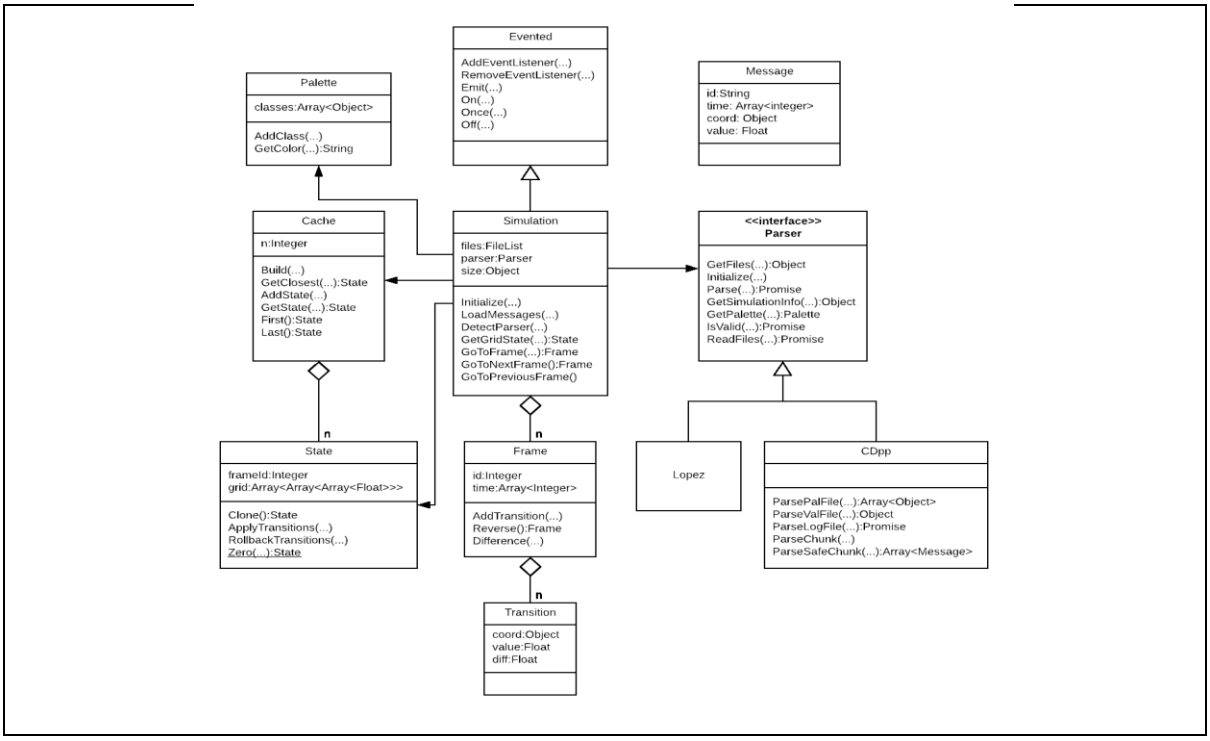
HTML5 introduces several new elements and attributes that can help you in building modern websites. One of these features include SVG which is very useful for useful for vector type diagrams like Pie charts, Two-dimensional graphs in an X, Y coordinate system etc. SVG will be very useful for advancement of WebViewer for visualization of Classic DEVS, which we will see

in further parts of the report. The HTML DOM is a standard object model and programming interface for HTML which is the main concept that is used by the WebViewer. It defines: The HTML elements as objects, the properties of all HTML elements, the methods to access all HTML elements and the events for all HTML elements.

2.5. CD WebViewer 2.0 Architecture

The Cell-DEVS formalism was originally presented by Wainer and Giambiasi, it is a combination of cellular automata and the discrete event system specification with explicit timing delays (Wainer and Giambiasi 2002). Each cell behaves as an atomic model and the entire cell space is a coupled model where each cell is coupled to their neighbors through input and output ports. Cells typically receive inputs from neighboring cells however, the formalism also provides a way to couple cells with classic DEVS models. Whenever an external event is received through an input port, the local computing function fires and determines the future state of the cell.

The Cell-DEVS Web Viewer is a lightweight, web-based software that allows users to visualize, interact with and analyze results of cellular automata-based simulators such as CD++.it is a standalone application programming interface (API) that would allow developers to build their own viewers for cellular automata simulations.



CDWebViewer 2.0 Architecture (Fig. i)

Steps to visualize the simulated results using CD Webviewer is by loading the files i.e. the user need to upload his initial values file (*.val*), the log file(*.log*), model file (*.ma*) and the pallet file (.pal) files for viewing the results. And then the parsing of the data is done by first evaluating all the files uploaded by the user so as invalid files can be detected only at the first step. Then after successfully receiving the required the data is divided into small chunks so as to reduce the time of processing the data for this, we use HTML5 filereader and the data is evaluated in the form of messages. This step posed some challenges since file reading with HTML5 is an event based asynchronous process and each parsing strategy must be tested against the files sequentially until a valid one is found. To solve this issue, the Promise class of the ES6 specification was key. 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 parsing strategy cannot be identified, the system will throw an error with proper feedback to the end user (St-Aubin, Yammine, Nayef, Wainer 2018). The 2MB chunk so extracted is then passed to filter the incomplete messages and then the parsed messages are stored into the chunks and this is done till the end of file. Then a parsing strategy is designed to identify the common pattern within the log files to extract the required states of cells at a time instance. After the parsing process data is defined into the array data structure of frames each frame consists of time value and array of transitions for a given time step. All the data is then passed using the event handlers to the simulation class after loading all the required data is loaded to the WebViewer, then the load button is activated. The User Interface (UI) of the web application is divided into the following parts: The Control widget, the DropZone widget, the Info widget, the Settings widget and the Palette widget and the Playback Widget. The Control widget, shown below, also allows the user to interact with the visual representation of the loaded simulation results.

The DropZone widget is where all the files used for visualization are uploaded by the user, the user can make changes to the number of grids or frames per second etc. under the settings widget. Before proceeding with simulation playback, the user can configure the visualization through the Settings and Palette widgets. Note that this can also be done after the simulation has been initially rendered and while it is being played back. These widgets are UI representations of the underlying Palette and Settings Evented object stored in the Simulation. Whenever any of the values of either class changes, a PaletteChanged or a SettingsChanged event is fired. The system reacts to these events and modifies the simulation playback accordingly. Through these widgets, there are many

options available to the user, they can modify the size of the rendered Canvas, speed up the playback, set the playback to be looped, etc. They can also modify the original palette included with the simulation files then save it as a new palette file. The system stores palette information in a Palette class which is essentially a collection of rendering classes that associate a color to a range of cell states. At this point, the user can animate the visualization by interacting with the Playback widget. This widget acts like a standard media control bar, it allows users to go to the first or last frame, to step one frame back or forward, to play backwards and forwards or to seek a specific Frame using the range slider. The Simulation object always keeps the current state of the simulation in memory through a State object.

The second version of the CDWebViewer also includes two major new features. The first one is a line chart, synchronized with the playback widget, where the state of selected cells is tracked across a simulation. This feature was implemented following feedback received from users of the first version of the platform. The second improvement is the integration of the viewer with RISE, a RESTful distributed simulation platform developed earlier at the Advanced Real-time Simulation Laboratory. RISE allows users to remotely manage the entire lifecycle of a simulation and to retrieve results of previously completed simulation. With the RISE platform, modelers only need to send their model files to the server through the CDWebViewer, they do not require any other simulation software. Other stakeholders can select a previously executed simulation to be visualized in the viewer. (St-Aubin, Yammine, Nayef and Wainer)

2.6. Importance of SVG

Scalable Vector Graphics (SVG) is an XML-based markup language for describing two-dimensional based vector graphics. SVG is essentially to graphics what HTML is to text. SVG is a text-based open Web standard. It is explicitly designed to work with other web standards such as CSS, DOM, and SMIL. SVG images and their related behaviours are defined in XML text files which means they can be searched, indexed, scripted and compressed. Additionally, this means they can be created and edited with any text editor and with drawing software. SVG is an open standard developed by the World Wide Web Consortium (W3C) since 1999.

SVG can be directly embedded into HTML using SVG or by using HTML in SVG by reference or including directly and JavaScript can be used to modify and animate SVG elements. Content of SVG can be styled by using CSS. SVG includes complete DOM (Document Object Model) and

has a high level of compatibility and is consistent with HTML DOM. There are several advantages of using SVG over other image formats, for example, SVG images can be easily created or edited with any text editor because SVG files are completely XML files. The SVG images are zoomable as per the size required and can be modified easily maintaining high quality at any resolution. There are two ways to add SVG to your Webpage. First is, Embedding SVG directly into your HTML or non-XHTML using <*svg*> tag. Second can be using the HTML tags like object, embed, iframe or image to embed SVG into the webpage. A *.svg* file is known as a standalone SVG file, because the SVG is an entire, independent document. When the browser receives a standalone SVG file, it reads it using the XML parser. SVG code in a .html or .xhtml file is known as inline SVG, because the SVG code is written directly "in the lines" of the HTML markup. The difference between .html and .xhtml is whether the browser will use the HTML parser or the XML parser.
SVG solves many of the problems that are faced in modern Web Development like,

- **Programmability and Interactivity**

  As SVG is completely editable and can also be scripted using JavaScript providing us all kind of animations and interactions added to the images using CSS or JavaScript.

- **Accessibility**

  SVG files are completely XML files, which are text based. SVG can be indexed or searched.

- **Performance**

  SVG files are smaller file sizes as compared other graphical representative files. Because of the smaller size, it improves the web performance.

- **Scalability and Responsiveness**

  To render graphics in the browser, SVG uses shapes like circle, rectangle etc., numbers and coordinates which makes it independent of resolution and infinitely scalable. This is helpful when dealing with higher resolution screens. Other image formats like GIF, JPEG have fixed dimensions, which causes them to pixelate when they are scaled. Although various responsive image techniques have proved valuable for pixel graphics, they will never be able to truly compete with SVG's ability to scale infinitely.

- **Editing Capabilities**

  SVG files are unique in that they can be edited in graphic editing programs like other images, but also in a text editor where the markup can be adjusted directly.

There are several reasons to use SVG in a webpage like its resolution independent and responsive behaviour, It has a navigable DOM, SVG can be animated or styled as per user's requirement and also it is interactive. All modern browsers support SVG. Practical uses of SVG's are logos, icons, graphs and images. In WebViewer, we are using SVG to design images that is, DEVS Diagram for the classic DEVS model.

2.7. Importance of Visualization in Simulation

As we know, Simulation produces a large amount of data and this data is used to replicate and reproduce real-world operations of each of the component. The data obtained through simulation helps us to establish correlations between the resources and the yield produced, both individually and collectively. Also, we test different combinations of inputs and different scenarios in simulation.

The goal of interactive visualization is to provide the ability to rapidly examine (and understand) data through visualization methods. Data visualization means presenting raw data through graphical representations that allow viewers—business analysts and executives—to explore the data and uncover deep insights. This visual format enables one to make quick and effective decisions since it is much easier for people to comprehend information through visuals rather than the raw reports.

Visualization technology allows us to represent visually the data that we obtain through simulation. Instead of racking your brains trying to process raw simulation data, with this technology you can simplify complex information and turn it into a friendly 2D drawing, graph, high-quality picture, or even a 3D animation.

By providing numerical data in an easy-to-understand format, visualization allows everyone – regardless if they have a technical background – to interpret and analyse data efficiently. This in turn helps to speed up workflows and facilitates communication. Data visualizations provide context, enabling engineering teams to find flaws or help explain complicated issues. Handling large amounts of data in a pictorial format to provide a summary of unseen patterns in the data, revealing insights and the story behind the data to establish a business goal.

There are several benefits of data visualization like, Better analysis of simulation results. Patterns can be identified easily in the result logs. Finding errors becomes an easy task.

In brief, visualization technology makes simulation data accessible.

3. Specification and Design

3.1. Classic DEVS Parser

The project is designed to read CD++ logs for Classic DEVS models. Classic DEVS results are saved as a *.log* file. The *.log* file contains all transitions that were made during the simulation. An example of or line of result from the log file is in the following format:

(Mensaje X / 00:00:10:000 / top(01) / controlin / 20.00000 para sender(02) "Mensaje X" indicates the input for model – 'sender' or (02) at a time step "00:00:00:000".

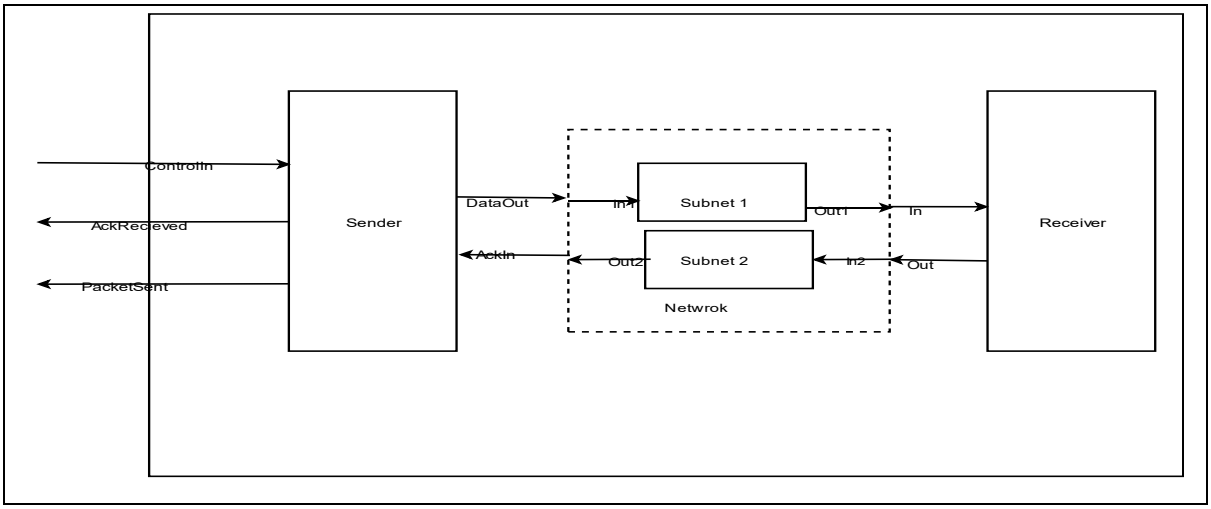
(Mensaje Y / 00:00:32:987 / receiver(06) / out / 1.00000 para top(01)), which indicates the output for model - 'receiver' or (06) at time step '00:00:32:987'. The log file is read and only the models for the output of the log is saved as a simulation model in the WebViewer. We determine the particular pattern for the CD++ logs where we focus mainly on output messages i.e. "Mensaje Y". We aim to detect the output message and extract the name of the atomic model which in the former mentioned is "receiver" the time "00:00:32:987" and "out" specifies the name of the output port. Which is stored in the transition array in the simulation class and is read as the time advances. To design SVG file, "id" attribute has to be defined for each coupled and atomic model. This "id" is read from the logs, in each log, the third parameter determines a model with it's ID in brackets, for instance, receiver (06). So, id = "06" has to be assigned to receiver model in the SVG file.

Reading Classic DEVS Logs

The first step in the WebViewer is to upload the files as soon as the file is read a small chunk is used in filereader function which classifies the logs into Classic DEVS and Cell DEVS models and then data is parsed accordingly. The file once identified as the Classic DEVS models is then read in chunks and the input and output message are read. From which each model name time and state of the model input and output model is extracted and passed into the simulation class and then the data so processed is used to visualize the result. To visualize the simulation, result the user need to design the model in SVG format. SVG technology is an open source copyrighted material of the W3C consortium and it is a language for 2D graphics within the XML (extensible Markup Language). The combination between SVG and JavaScript offers a powerful platform usable for interactive 2D graphics, comparable to the Flash and Java technologies. The SVG file uploaded is then mapped with the simulation results.

3.2. Representation of Classic DEVS model using SVG

After uploading the make file(.ma) and the (.log) file to the Drop box the WebViewer will categorize the files to Classic DEVS category and will open a widget where the user can upload the DEVS Diagram (SVG file) designed. The SVG file needs to be designed in a fashion so as the proper visualization of the model. During the design of the SVG the user can use any shape circle, rectangle or polygon to design its atomic and coupled models. We use an example of Alternate BIT protocol model for the reference.



Alternate Bit Protocol SVG (Fig. ii)

In alternate bit protocol the user must map the log files as per the designed model. Each polygon must be assigned with his respective model name for instance <rect model = "sender" x="187" y="80" width="120" height="270" fill="none" stroke="#000000" type ="model"/>. In the former mentioned we have two atomic model Sender and Receiver and one coupled model Network the coupled model contains two atomic models Subnet 1 and Subnet 2. An atomic model "network" is designed as a rectangle for each atomic model the user must give model name in the used rectangle as model = "sender" the tag model must have the value same as in (*.ma*) file. The user can define the type of the rectangle by using tag type so as to categorize his structure and models into atomic or coupled this is not mandatory however is advised for the good practice. The connections between the different models can be given using the path. <path type = "Output" d="M 307 189.89 L 378.63 190.91" fill="none" stroke="#000000" stroke-miterlimit="10" model = "*dataout*"/>. The modeler can also define the type of the connection in the SVG file as input or output by defining the type of the port. In the connection user can design any stroke size or any

type of line he just needs to mention the connection name as per his make file(.ma) as 'model = "*dataout'*. The user can group his couple model by giving group id and type <g id ="sender" type="*atomic*">. in a group of the atomic model user need to add all the connection(paths) to different atomic models. We use switch tag to give names to the model. So, user can use different languages and later it can be used if the site is used in other language. The whole SVG file is given in appendix.

3.3. Standardization of SVG

For proper working of the model we need to set some standards in designing of SVG file. User is independent to use any third-party software to design his model he just needs to follow certain guidelines for proper visualization.

Each model must be grouped whether its atomic or coupled and all the polygons and connections must be defined with in the same group. The user is independent to include text if he likes, He just needs to be careful to use switch tag and all the names using tag <Text> must be given in the next line of the model or the path motioned. We follow the conventions of make file(*.ma*) by first designing all the components and grouping them into type atomic and coupled. The atomic models of a coupled model must be defined in the same coupled group. In the end the user needs to implement all the paths and define them using <path> tag. The below is the example of standard SVG file for the former mention ABP model.

```
<g id ="sender" type="atomic">
    <rect id = "02" model = "sender" x="187" y="80" width="120" height="270" fill="none" stroke="#000000"  type ="model"/>
    <g transform="translate(227.5,208.5)">
    <text x="20" y="12" fill="#000000" text-anchor="middle" font-size="12px" font-family="Helvetica">Sender</text>
    </g>
</g>
```

Atomic model SVG implementation (Fig. iii)

Figure (iii) represents the piece of code from the SVG file used to design the DEVS Diagram. The code here shows how to draw an atomic model. The atomic model lies within a group tag with id = "sender" and we define the type of the model as "atomic". Here we draw a rectangle to define an atomic model (or any other atomic/coupled model). We need to provide attributes to rectangle while designing SVG like width, height, x-axis, y-axis etc. The type of the rectangle is given as

“model”, because it represents a model or component of the DEVS model. Text is to be added to each model so as to name the model. The user needs to define the location of the text by specifying the x-axis and y-axis coordinates with respect to the rectangle and path followed.

```
<g id ="Network" type="coupled">
<rect id = "03" model = "network" x="387" y="120" width="230" height="210" fill="none" stroke="#000000" stroke-dasharray="3 3" type ="model"/>
<g transform="translate(437.5,208.5)">
<text x="40" y="100" fill="#000000" text-anchor="middle" font-size="12px" font-family="Helvetica">Netwrok</text> </g>
    <g id ="subnet1" type="atomic">
        <rect id = "04" model = "subnet1" x="437" y="155" width="120" height="60" fill="none" stroke="#000000"  type ="model"/>
        <g transform="translate(470.5,198.5)">
        <text x="20" y="1" fill="#000000" text-anchor="middle" font-size="12px" font-family="Helvetica">Subnet 1</text> </g>
    </g>
    <g id ="subnet2" type="atomic">
        <rect id = "05" model = "subnet2" x="442" y="225" width="120" height="60" fill="none" stroke="#000000"  type ="model"/>
        <g transform="translate(470.5,258.5)">
        <text x="20" y="1" fill="#000000" text-anchor="middle" font-size="12px" font-family="Helvetica">Subnet 2</text> </g>
    </g>
</g>
```

Couple Model SVG implementation (Fig. iv)

Figure (V) represents how we define the coupled models to design a DEVS diagram. First you need to define the outer model lies with in a group tag with id = “network” and type =” coupled”. All the atomic models within the coupled model has to be in the same group tag. Making sure we follow the former motioned coding conventions for defining atomic models.

```
<g type = "links">
 <path d="M 307 189.89 L 378.63 190.91" fill="none" stroke="#000000" stroke-miterlimit="10" model = "dataout"/>
 <path d="M 383.88 190.98 L 376.83 194.38 L 378.63 190.91 L 376.93 187.38 Z" fill="#000000" stroke="#000000" stroke-miterlimit="10" />
<g transform="translate(314.5,187.5)">
<text x="21" y="12" fill="#000000" text-anchor="middle" font-size="12px" font-family="Helvetica">DataOut</text> </g>


 <path d="M 388.15 250.62 L 315.41 249.89" fill="none" stroke="#000000" stroke-miterlimit="10" model = "ackin"/>
 <path d="M 310.16 249.84 L 317.19 246.41 L 315.41 249.89 L 317.12 253.41 Z" fill="#000000" stroke="#000000" stroke-miterlimit="10" />
<g transform="translate(319.5,241.5)">
<text x="15" y="12" fill="#000000" text-anchor="middle" font-size="12px" font-family="Helvetica">AckIn</text> </g>

 <path model = "in1" d="M 387 194 L 431.63 194" fill="none" stroke="#000000" stroke-miterlimit="10" />
 <path d="M 436.88 194 L 429.88 197.5 L 431.63 194 L 429.88 190.5 Z" fill="#000000" stroke="#000000" stroke-miterlimit="10" />
<g transform="translate(418.5,188.5)">
<text x="8" y="12" fill="#000000" text-anchor="middle" font-size="12px" font-family="Helvetica">In1</text> </g>

 <path d="M 557 201 L 612.63 201" fill="none" stroke="#000000" stroke-miterlimit="10" model = "out1"/>
 <path d="M 617.88 201 L 610.88 204.5 L 612.63 201 L 610.88 197.5 Z" fill="#000000" stroke="#000000" stroke-miterlimit="10" />
<g transform="translate(561.5,195.5)">
<text x="13" y="12" fill="#000000" text-anchor="middle" font-size="12px" font-family="Helvetica">Out1</text> </g>
</g>
```

Defining links in SVG (Fig. v)

At last we define all the input and output and the connections using the path rag setting the stroke width. The user has to be careful in stating the connection names as the links in the (.ma) file. The user must use the model attribute he needs to give the same name of the input
and output links as in the(.ma) file.

Key SVG Standards

- Some of the key standards to be followed while designing a DEVS Diagram using SVG. Each model should be enclosed within a <g></g> group tag, to maintain the consistency throughout the SVG.
- Each group should have an attribute "type" so as to categorize the different categories of groups in DEVS like atomic models, coupled models, links.

```
<g id ="receiver" type="atomic">
```

Type SVG (Fig. vi)

- Defining model attribute for each of the SVG element with the name of the model or link used while defining the DEVS components and links. Set "id" attribute of SVG. "id" attribute has to be defined for each coupled and atomic model. This "id" is read from the logs, in each log, the third parameter determines a model with it's ID in brackets, for instance, receiver (06). So, id = "06" has to be assigned to receiver model in the SVG file.

```
<rect id = "02" model = "sender" x="187" y="80" width="120" height="270" fill="none" stroke="#000000"  type ="model"/>
```

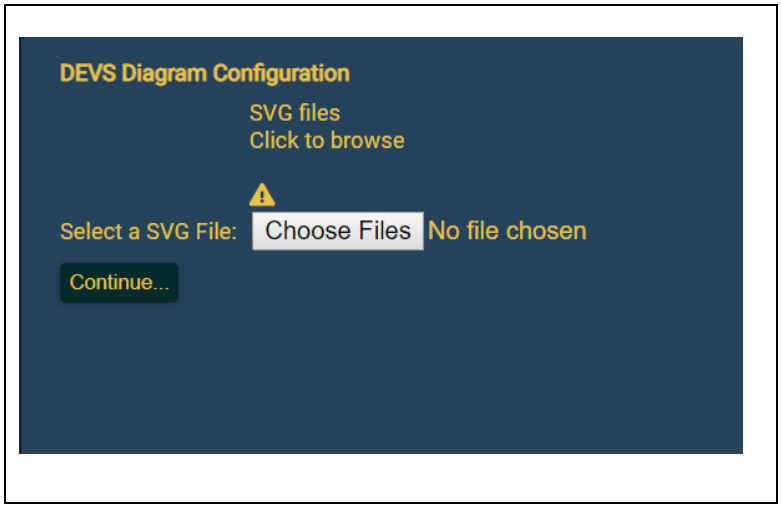
Attribute model SVG (Fig. vii)

- Maintaining the consistency by giving an attribute "Type" to each element i.e. defining either component or link.
- Making sure that there is no attribute "**pointer-events**" in any of the element of SVG. If used, then the value must not be set equal to "None".
- Keeping the SVG as simple as possible. Here in the example we used to draw SVG, it is made sure to use only below given SVG elements to avoid complexity and improve understanding. <SVG>, <rect>, <g>, <path> and <text> are used to draw SVG of a simple DEVS Diagram.

4. Implementation

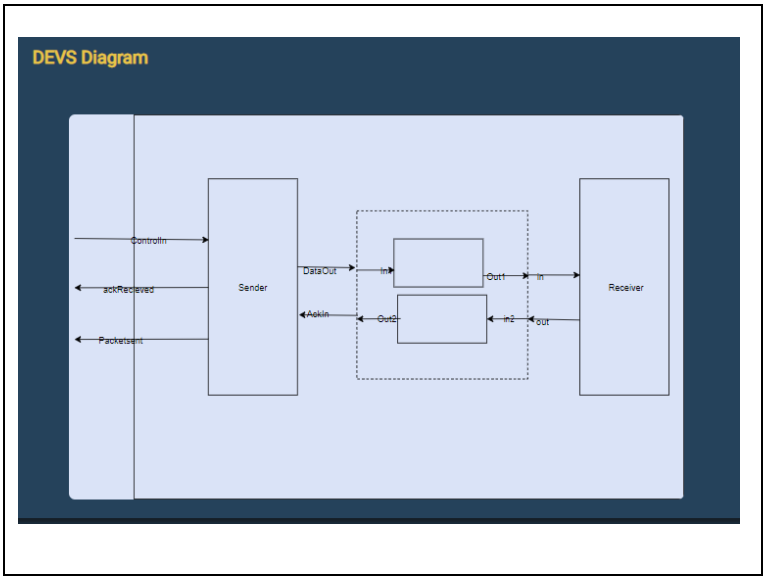
4.1. Dropbox for SVG file and Loading SVG in Web viewer

Once the simulation has been loaded with log and the .ma file, it's time to visualise the DEVS Diagram with help of SVG in the WebViewer simulator. The DEVS diagram is being created by the user using SVG. So, the user is asked to input the SVG file as shown in the figure 4.1.1:



DEVS Diagram Dropbox (Fig. viii)

Once the SVG file is entered by the user, User must click continue to load SVG in Webviewer. On pressing Continue, User can see the following Screen, where we have SVG loaded.



Loaded DEVS Diagram (Fig. ix)

The first challenge is to read the file uploaded by the user. The SVG uploaded by the user is not such a large file and thus, it can be directly read into a string. The string is by the configurator and parsed to the viewer in next step, where the SVG is directly embedded in a html tag, that defines a division inside the widget. There are several ways to load SVG within your webpage, for example, <object>, <embed> tags in HTML.

An SVG image can be added as a code directly within your HTML5 page using outer `<svg>` tags, and the external SVG file user creates, already has outer <svg> tags. The method works in all HTML5 browsers and permits animation, scripting and CSS and this is the reason we load SVG directly to our webpage, using outer <svg> tags. This makes it easier to manipulate the SVG image, change attributes like fill colour, width, height as per the requirements when resizing the widget or running the simulation tests.

After the SVG has been loaded to the WebViewer, next comes the list of functions that have been used to manipulate and animate the DEVS diagram (SVG) to visualize the results of simulation. The results of the simulation are given by the log file, that has been already loaded before loading SVG.

4.2. List of Functions on DEVS Diagram

Code flow for DEVS Diagram:

After the Simulation is load by entering the .log and .ma file, the user gets a predefined list of all widgets available for the visualization of the simulator, Classic DEVS in this case. Here, the explanation of the flow of code for the DEVS Diagram is explained in brief. When the user chooses an option to load the DEVS Diagram Widget, the definition and configurator for DEVS Diagram are called. The control reaches to the configurator first, where a user is asked to upload an SVG file in UI. When a user uploads the file, the "change" event is handled for the input tag that is accepting the file. The file content is then read by the FileReader object of javascript defined in ParseFile function in class Sim, which returns the promise to the file content and it is saved in this.svg variable. Configurator class then emits the content of the SVG file in a variable to the AutoDevsDiagram Class. After inputting the file, the user is asked to continue. On continue, control goes to the definition of DEVS Diagram widget, that is, AutoDevsDiagram class, which is responsible for loading SVG on the webpage and other events handling on DEVS Diagram. First of all, SVG is loaded on the webpage by calling a function this.Widget.SetSVG(config.svg); with parameters accepted from configurator class. SetSVG() is implemented in class DevsDiagram, where the SVG is loaded on the webpage and setting up initial attributes of the SVG like height, width, viewbox, etc. All other events on DEVS diagram like click, mouse move, simulation move, etc are defined in the AutoDevsDiagram class. Mouse events of DEVS diagram are handled in class DevsDiagram and simulation events are handled by Simulation class. Data() function in AutoDevsDiagram returns all the transitions values from all simulation frames. This transition array helps us to find the models at a specific simulation state during the simulation is being rendered and changes are made to those models of SVG which are present at that transition frame. A brief description of all functions in class DevsDiagram is defined below with their result samples.

The following is the list of function applied on SVG to visualize the results of simulation in DEVS Diagram.

a. Reading SVG using *FileReader*

   Once the file is uploaded, by HTML input tag, next step is to read the file. The JavaScript's *FileReader* type can read the file and store it in a JavaScript variable. There are several formats that a *FileReader* creates to represent file data and the format. Reading file can be done through calling one of *FileReader's* method, that is, *readAsText (),*

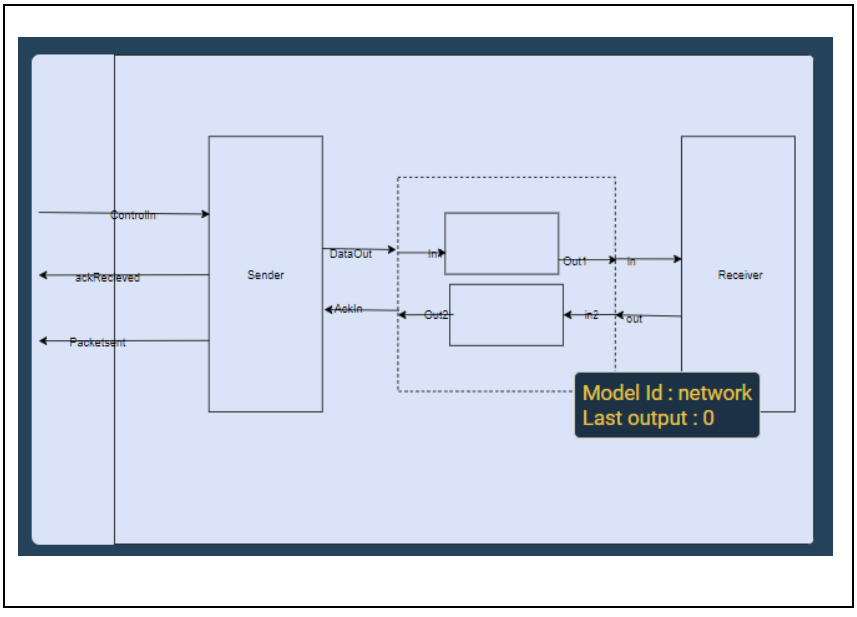
readAsBinaryString (), readAsArrayBuffer () and readAsDataURL(). The result of the read is always represented by event.target.result.

b. Setting the SVG

Now, that the SVG has been loaded, we need to make SVG image to fit to the widget and we manipulate the VIEWBOX attribute of the SVG according to the size of the widget, and along with this we set the height and width of "svg" attribute to 100% , which means the whole SVG image with its each attribute will now automatically adjust to the outer attribute size provided to it, every time when loaded or resized. The initial state of the diagram is saved here, initial state means the saving the colours and other attributes of the model.

c. Hover on DEVS diagram models(components)

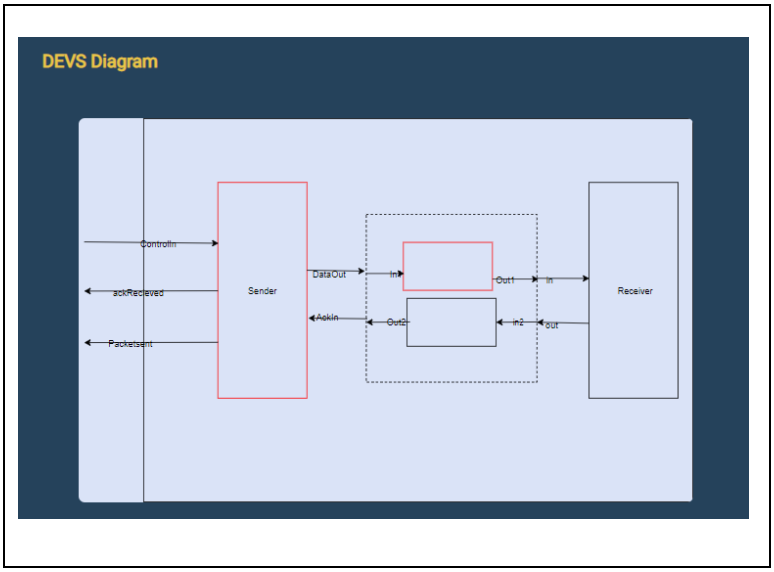
When the user hovers over the SVG image, user needs to know what the name of the specific model is. We are here displaying the name of the model and the last state output of that model on mouser over the model element using tooltip and hide the tooltip when the mouse is out of the model element. So, when the results of simulation are being displayed, user may be able to know the last state output of the model at that time frame. The results of Hover over the diagram are shown in the figure (x):



Hover function (Fig. x)

d. On-Click on DEVS diagram models(components)

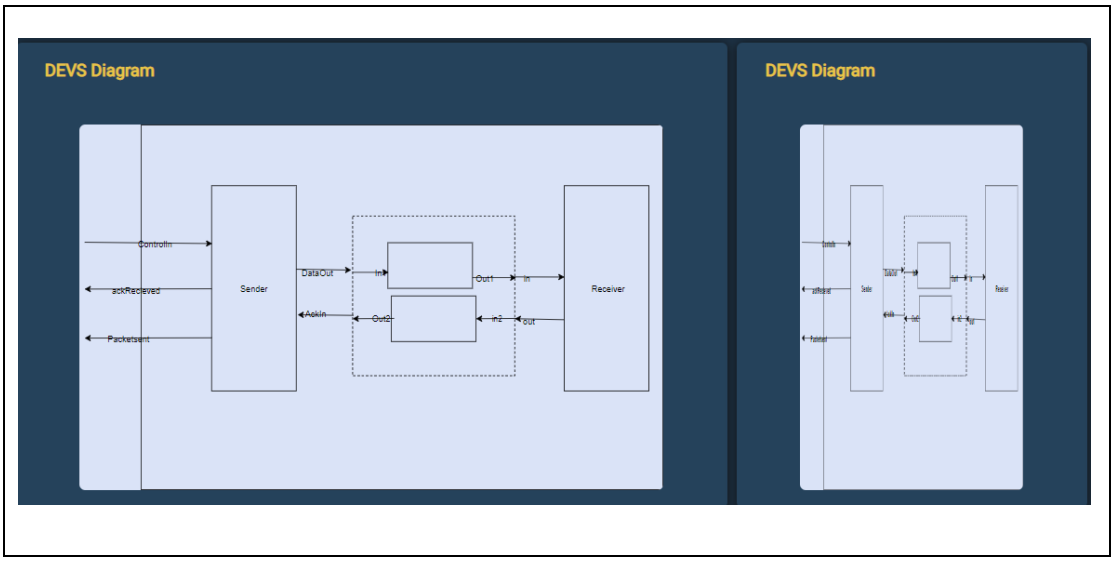
When the user clicks on any of the models, the model has to be selected. The model selection is being shown by adding a border colour to the model, so that it is differentiated from all other models. Here we are marking the selected model border Red in colour as shown in the figure (xi):



Model Selection (Fig. xi)

e. Resize SVG image

Resizing the SVG image when the widget is expanded or contracted. This is done by assigning a size to the division containing SVG image, a height and width that will change as per the changes in the widget size. The figure below shows the changing size of the SVG image and the widget.

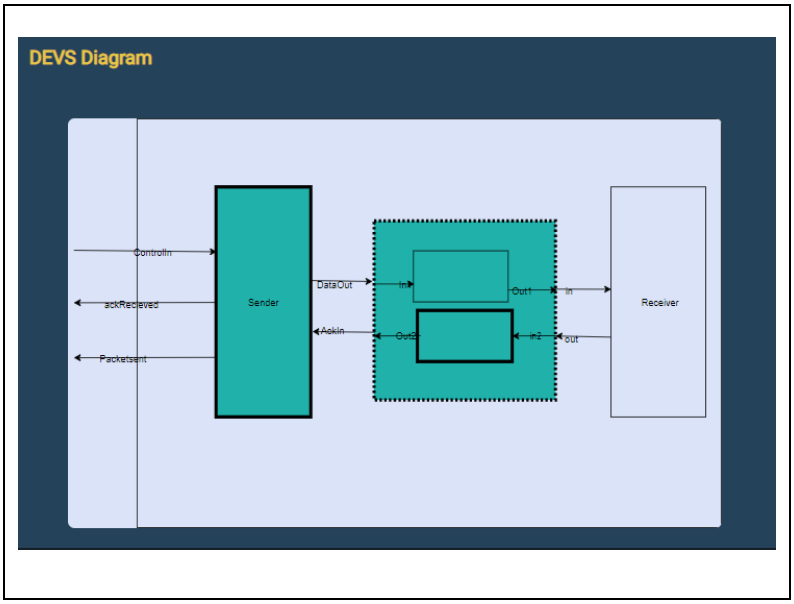


Resize function (Fig. xii)

f. Simulation Rendering

Once the transitions have been read from the .log file and the cache built, the initial states of the model in DEVS Diagram is drawn using the initial values. Starting from the states of the models at time 0, the transitions for the next time step in the simulation log file are applied simultaneously that must be rendered to the DEVS diagram. This process is repeated until the end of the log file is reached. Once the transitions have been read from the .log file, the initial models state is drawn using the initial states.

Once the states of the model in DEVS Diagram is rendered, the WebViewer is ready to playback the reconstructed simulation using the playback widget. The user can control the speed of playback, navigate to specific time-steps. When the Simulation moves ahead, the changes of the states are being highlighted for the models (components) of the DEVS Diagram by filling the colour in each model that is active at the simulation frame. The below figures represent the DEVS Diagram models states at different time-steps.



Simulation Rendering (Fig. xiii)

g. On Simulation Jump

Since the visualization is reconstructed from the transitions, navigating freely through the simulation requires special consideration. When the user skips ahead or navigates back using the slider, many time-steps may need to be applied to the current state to obtain the time-step requested by the user. Depending on the duration of simulation, the state at that time step must be visualised in the DEVS Diagram.

h. Draw Changes to DEVS Diagram

Draw changes function deals with making any changes to the SVG Image, when the time step changes during the visualisation of simulation, when user skips or navigates back or ahead using the slider, or when the selection is made. This function takes the model and the style changes to be made to the model is accepted as the parameter and the style changes are made to the provided model.

i. Resetting the DEVS Diagram to its initial state.

Every time the SVG image changes with the simultaneously time steps, the SVG image style changes at each time step, but that needs to be reset before the next transition because at the next time step, the state of the models changes , which continues till the end of the

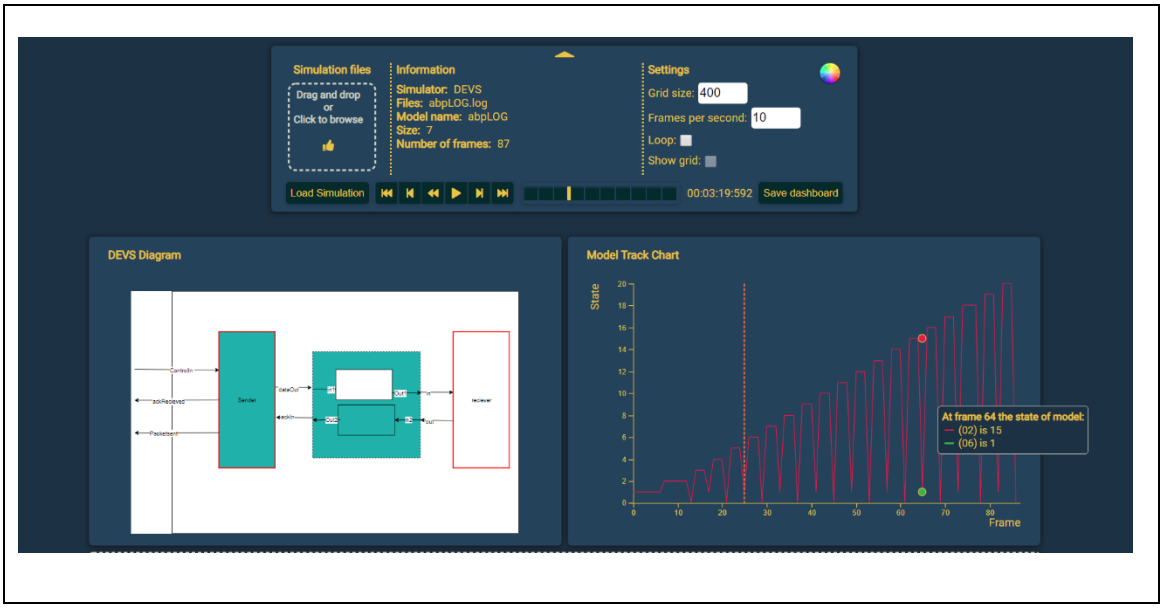
logs of simulation results. Reset DEVS diagram changes the SVG image back to its initial state, which is saved. SVG image is reset every time before any changes can be made to it.

4.3 Results and Evaluation

The project is set out to visualise the simulation results of Classic DEVS models using DEVS diagram. This section of the report shows the simulation results represented by DEVS diagram for ABP DEVS model. Some evaluation has already been given with the implementation of each part of the project and is not repeated in this section.

Simulation Rendering (Fig. xiv)

Figure (xiv) shows the results for the state frequency chart and DEVS Diagram. When the Simulation moves ahead, the changes of the states are being highlighted for the models (components) of the DEVS Diagram user can track the output states for the models that user has to use to configure the state frequency chart and pass the model state values. The user can keep track of a model state and can also compare the state values of different time stamps or same time stamps.



Model Track Chart (Fig. xv)

In Figure (xv) two atomic model's sender and receiver are selected and are being monitored with the Model Track chart. The user can monitor state of both the models at a given time frame by hovering over the chart it shows the state values of both the models.

Steps to run the simulation

1. Try it at https://ankitvig.github.io/CDWeb--3.0/
2. The simulation logs are available in the logs folder.
3. As in WebViewer- 3.0, we mainly focus on the visualization of Classic DEVS Models. So, we here consider an example of ABP DEVS model. The simulation logs is available in the folder : https://github.com/AnkitVig/CDWeb--3.0/tree/master/log/ABP
4. The SVG file required to visualize the DEVS Diagram is also available in folder https://github.com/AnkitVig/CDWeb--3.0/tree/master/log/ABP
5. The .log and .ma files are dropped as input to load the simulation into the WebViewer Dashboard.
6. Successful Uploading of these files will enable the "Load Simulation" button. Click the button to load the simulation.
7. To Visualise DEVS Diagram, select DEVS Diagram Widget and click Load.

8. On loading DEVS Diagram Widget, user will be asked to upload the SVG file. Upload the SVG file and click "Continue".
9. The DEVS Diagram will be loaded in the WebViewer, user can now visualise the simulation results through Playback.
10. Setting can be configured as per preference or requirement through control widget.
11. To view State Frequency Chart, enter the states to be monitored, click "Continue". The states will be represented in form of a bar graph.
12. To view Model Tracking Chart, select the models from DEVS Diagram to be monitored.
13. *Demo Video : https://youtu.be/-XCs9xhjOSQhttps://youtu.be/-XCs9xhjOSQ*

5. Conclusion

In this project we present an improved and reengineered version of WebViewer which is a web-based visualization and analysis platform for results of cellular automata-based simulations. The new version of the software proposes an object-oriented architecture to visualize and analyse the simulation result of classic DEVS models.

The third version of the WebViewer is a generic visualization tool that deals with both the Cell DEVS and Classic DEVS simulation. The new features that are included in this version is visualization of simulation results of Classic DEVS using a DEVS Diagram, state frequency chart and model track chart. The DEVS diagram is a visual representation of DEVS model designed using CD++ tool kit which includes all the atomic and coupled models. The DEVS diagram is designed using Scalable Vector Graphics (SVG) by following certain standards mentioned in the report. This version of the WebViewer focuses on reading the simulation logs of classic DEVS models using FileReader. The FileReader object allows the user to load a very large .log file quickly, pass it into chunks and load it in memory of WebViewer to render animate and analyze. The SVG file is also read using FileReader object and the SVG is directly embedded into the webpage representing the DEVS Diagram. Visualization of DEVS model allows us to represent visually the data that we obtain through simulation. Instead of racking your brains trying to process raw simulation data, with this technology you can simplify complex information.

This increases the productivity of the user by reducing the need for custom file parsing. It also reduces the user dependency on installing some third-party software's to visualize his result. It is a step forward in supporting the cloud architecture user can remotely load his files and can visualize the results. The use of rise platform helps user load the simulation result from the sever and visualize them remotely without help of any eternal software through his web browser. To increase the user base of modelling and simulation tools a user-friendly tool is designed for better analysis of different models. This helps in increasing the public for DEVS formalism and simulation on academic and industrial level.

Future Work

Future work includes Developing more charts for better comparison and data analysis of the Classic DEVS models results. Developing more parsers for different logs such as CADMIUM to make the WebViewer more generic. The user still has to upload the file in the new widget an update is needed to help user for uploading all files in single dropbox. To make it more interactive some user guidelines need to be added. Work can be done on designing the widgets to help the user rearrange the widgets as per the required results. The dependency on external SVG can be eliminated a system can be designed to auto generate the DEVS diagram based on the ma or log files. Till now tool is designed just to view the output results there is a need to include the input parameters for better understanding and visualization. A pallet file can be generated by the user to give different colours at different state values to differentiate or monitor the crucial states. The software dependency can be removed by integrating rise and cadmium to generate classic DEVS as well as cell DEVS results. Other simulation tool kits and external libraries can be embedded into the tool.

6. Appendix

```
<svg xmlns="http://www.w3.org/2000/svg" width="500" height="300" viewBox="-0.5 -0.5 828 481" style=
"background-color:rgb(255, 255, 255); border-radius:5px;">
<g>
<rect id="FirstCouple"  model = "FirstCouple" x="87" y="0" width="740" height="480" fill="none" stroke="#000000"  type ="coupled"/>

<g id ="sender" type="atomic">
    <rect id = "02" model = "sender" x="187" y="80" width="120" height="270" fill="none" stroke="#000000"  type ="model"/>
    <g transform="translate(227.5,208.5)">
    <text x="20" y="12" fill="#000000" text-anchor="middle" font-size="12px" font-family="Helvetica">Sender</text>
    </g>
</g>

<g id ="receiver" type="atomic">
    <rect id = "06" model = "receiver" x="687" y="80" width="120" height="270" fill="none" stroke="#000000"  type ="model"/>
    <g transform="translate(725.5,208.5)">
    <text x="21" y="12" fill="#000000" text-anchor="middle" font-size="12px" font-family="Helvetica">Receiver</text> </g>
</g>

<g id ="Network" type="coupled">
<rect id = "03" model = "network" x="387" y="120" width="230" height="210" fill="none" stroke="#000000" stroke-dasharray="3 3" type ="model"/>
<g transform="translate(437.5,208.5)">
<text x="40" y="100" fill="#000000" text-anchor="middle" font-size="12px" font-family="Helvetica">Netwrok</text> </g>
    <g id ="subnet1" type="atomic">
        <rect id = "04" model = "subnet1" x="437" y="155" width="120" height="60" fill="none" stroke="#000000"  type ="model"/>
        <g transform="translate(470.5,198.5)">
        <text x="20" y="1" fill="#000000" text-anchor="middle" font-size="12px" font-family="Helvetica">Subnet 1</text> </g>
    </g>
    <g id ="subnet2" type="atomic">
        <rect id = "05" model = "subnet2" x="442" y="225" width="120" height="60" fill="none" stroke="#000000"  type ="model"/>
        <g transform="translate(470.5,258.5)">
        <text x="20" y="1" fill="#000000" text-anchor="middle" font-size="12px" font-family="Helvetica">Subnet 2</text> </g>
    </g>
</g>
<g type = "links">
<path d="M 307 189.89 L 378.63 190.91" fill="none" stroke="#000000" stroke-miterlimit="10" model = "dataout"/>
<path d="M 383.88 190.98 L 376.83 194.38 L 378.63 190.91 L 376.93 187.38 Z" fill="#000000" stroke="#000000" stroke-miterlimit="10" />
<g transform="translate(314.5,187.5)">
<text x="21" y="12" fill="#000000" text-anchor="middle" font-size="12px" font-family="Helvetica">DataOut</text> </g>


<path d="M 388.15 250.62 L 315.41 249.89" fill="none" stroke="#000000" stroke-miterlimit="10" model = "ackin"/>
<path d="M 310.16 249.84 L 317.19 246.41 L 315.41 249.89 L 317.12 253.41 Z" fill="#000000" stroke="#000000" stroke-miterlimit="10" />
```

```
<g transform="translate(418.5,188.5)">
<text x="8" y="12" fill="#000000" text-anchor="middle" font-size="12px" font-family="Helvetica">In1</text> </g>

<path d="M 557 201 L 612.63 201" fill="none" stroke="#000000" stroke-miterlimit="10" model = "out1"/>
<path d="M 617.88 201 L 610.88 204.5 L 612.63 201 L 610.88 197.5 Z" fill="#000000" stroke="#000000" stroke-miterlimit="10" />
<g transform="translate(561.5,195.5)">
<text x="13" y="12" fill="#000000" text-anchor="middle" font-size="12px" font-family="Helvetica">Out1</text> </g>

<path d="M 620.91 254.61 L 568.37 254.96" fill="none" stroke="#000000" stroke-miterlimit="10" model = "in2"/>
<path d="M 563.12 254.99 L 570.09 251.45 L 568.37 254.96 L 570.14 258.45 Z" fill="#000000" stroke="#000000" stroke-miterlimit="10" />
<g transform="translate(584.5,248.5)">
<text x="7" y="11" fill="#000000" text-anchor="middle" font-size="11px" font-family="Helvetica">In2</text> </g>

<path d="M 617 200 L 680.99 200.14" fill="none" stroke="#000000" stroke-miterlimit="10" model="in"/>
<path d="M 686.24 200.15 L 679.23 203.63 L 680.99 200.14 L 679.25 196.63 Z" fill="#000000" stroke="#000000" stroke-miterlimit="10" />
<g transform="translate(629.5,195.5)">
<text x="5" y="12" fill="#000000" text-anchor="middle" font-size="12px" font-family="Helvetica">In</text> </g>

<path d="M 686.64 256.31 L 623.37 255.12" fill="none" stroke="#000000" stroke-miterlimit="10" model = "out"/>
<path d="M 618.12 255.02 L 625.18 251.65 L 623.37 255.12 L 625.05 258.65 Z" fill="#000000" stroke="#000000" stroke-miterlimit="10" />
<g transform="translate(628.5,251.5)">
<text x="9" y="12" fill="#000000" text-anchor="middle" font-size="12px" font-family="Helvetica">Out</text> </g>

<path d="M 7 154.39 L 181.47 156.08" fill="none" stroke="#000000" stroke-miterlimit="10"  model = "controlin"/>
<path d="M 186.72 156.13 L 179.69 159.56 L 181.47 156.08 L 179.76 152.56 Z" fill="#000000" stroke="#000000" stroke-miterlimit="10" />
<g transform="translate(82.5,149.5)">
<text x="26" y="12" fill="#000000" text-anchor="middle" font-size="12px" font-family="Helvetica">ControlInÂ </text> </g>

<path d="M 187.15 215.62 L 13.37 215.99"  fill="none" stroke="#000000" stroke-miterlimit="10" model ="ackrecieved"/>
<path d="M 8.12 216 L 15.11 212.48 L 13.37 215.99 L 15.13 219.48 Z" fill="#000000" stroke="#000000" stroke-miterlimit="10" />
<g transform="translate(45.5,211.5)">
<text x="35" y="12" fill="#000000" text-anchor="middle" font-size="12px" font-family="Helvetica">AckRecieved</text> </g>

<path d="M 187.15 280.12 L 13.37 280.49" fill="none" stroke="#000000" stroke-miterlimit="10" model ="packetsent" />
<path d="M 8.12 280.5 L 15.11 276.98 L 13.37 280.49 L 15.13 283.98 Z" fill="#000000" stroke="#000000" stroke-miterlimit="10" />
<g transform="translate(39.5,274.5)">
<text x="30" y="12" fill="#000000" text-anchor="middle" font-size="12px" font-family="Helvetica">PacketSent</text> </g>

<path d="M 445.91 254.61 L 393.37 254.96" fill="none" stroke="#000000" stroke-miterlimit="10" model = "out2"/>
<path d="M 388.12 254.99 L 395.09 251.45 L 393.37 254.96 L 395.14 258.45 Z" fill="#000000" stroke="#000000" stroke-miterlimit="10" />
<g transform="translate(414.5,248.5)">
<text x="13" y="12" fill="#000000" text-anchor="middle" font-size="12px" font-family="Helvetica">Out2</text></g>
</g>
```

7. References

- Discrete-Event Modeling and Simulations: A Practitioner's Approach, Gabriel A. Wainer
- Analytics and Visualization Of Spatial Models As A Service Bruno St-Aubin, Eli Yammine, Majed Nayef, Gabriel A. Wainer
- SVG Language (Scalable Vector Graphics) For 2D Graphics in XML and Applications Marin Vlada
- N-dimensional Cell-DEVS. G. A. Wainer, N. Giambiasi In Discrete event dyamic systems. April 2002
- A CELL-DEVS VISUALIZATION AND ANALYSIS PLATFORM Bruno St-Aubin, Omar Hesham and Gabriel Wainer Department of Systems and Computer Engineering Carleton University
- The DEVS Formalism Rhys Goldstein1, Gabriel A. Wainer, Azam Khan
- "Distributed simulation of DEVS and Cell-DEVS models in CD++ using Web-Services". G. Wainer, R. Madhoun, K. Al-Zoubi. Simulation Modelling Practice and Theory, Elsevier, Maryland Heights, MO, USA. Volume 16October 2008
- ANALYTICS AND VISUALIZATION OF SPATIAL MODELS AS A SERVICE Bruno St-Aubin Eli Yammine Majed Nayef Gabriel Wainer
- Multifacetted modelling and discrete event simulation - Zeigler – 1984
- Discrete Event System Specification (DEVS) Modelling and Simulation Yentl Van Tendeloo, Hans Vangheluwe
- JavaScript Frameworks for Modern Web Development Sufyan bin Uzayr Nicholas Cloud Tim Ambler
- ABP Simulator model (http://www.sce.carleton.ca/faculty/wainer/wbgraf/samples/alternatebitprot.zip)