ANALYTICS AND VISUALIZATION OF SPATIAL MODELS AS A SERVICE

Bruno St-Aubin
Eli Yammine
Majed Nayef
Gabriel Wainer

Department of Systems and Computer Engineering
Carleton University
1125 Colonel By Drive
Ottawa, ON, Canada
{bruno.staubin, eli.yammine, majed.nayef}@carleton.ca
gwainer@sce.carleton.ca

ABSTRACT

The Cell-DEVS WebViewer is a lightweight, web-based software that allows users to visualize, interact with and analyze results of cellular automata based simulators such as CD++. A first version of this platform was presented at SpringSim 2018, since then, many improvements have been added. Notably, it has been refactored to follow a more object-oriented architecture which is presented in detail in this paper. Other improvements include additional analytical features as well as an integration with RISE, a server-side distributed simulator that allows users to remotely manage the complete lifecycle of a simulation. The aim of this platform is to support stakeholders and decision makers by providing them with a simple, user-friendly graphic user interface that offers a deeper insight into the system under study.

Keywords: DEVS, Cell-DEVS, Visualization and Analysis, Remote simulation.

1 INTRODUCTION

One of the obstacles that DEVS (Discrete Event System Specification) faces is its limited approachability with regards to non-expert users. Absent proper tools, modeling and simulation with DEVS requires an in-depth knowledge of the theory underlying the specification. Researchers have tackled this issue in many different ways that mostly gravitate around user-friendly interfaces that abstract some of the DEVS theory (Goldstein, Breslav and Khan 2016; Goldstein, Breslav and Khan 2018; Bergero and Kofman 2011; Sarjoughian and Elamvazhuthi 2010). What is apparent is that most of these are focused on the modeling and simulation aspect of DEVS and few of them propose visualization and analysis tools to support end users and decision makers.

DEVS software development kits typically focus on performance, model debugging, verification and validation or the modeling process itself rather than analysis of the results. As can be observed in recent surveys of the field, DEVS frameworks tend to provide only summary visualization and analysis capabilities (Tendeloo and Vangheluwe 2017; Franceschini et al. 2014). More comprehensive exploration of results is usually relegated to third party software such as MatLab, R or Tableau. Narrowing this gap would provide greater usability and possibly contribute to a higher adoption rate of DEVS among the community of simulation and modeling users.

In this paper, we present a in-depth look into the Cell-DEVS WebViewer (CDWebViewer), a web-based platform for visualization and analysis of Cell-DEVS simulation results and other cellular automata based simulations (St-Aubin, Hesham and Wainer 2018). The viewer is coded entirely in HTML5 and JavaScript,

has minimal technological overhead, can be run locally on a user's computer and minimizes external dependencies.

This project contributes towards some important aspects of modeling and simulation. It seeks to improve modeler feedback by providing an intuitive user-friendly interface that offers good performance through the use of quickly maturing web technologies (HTML5, Canvas, WebGL). This allows users to easily load and explore their simulation results and to perhaps detect unforeseen issues with their models. It also enhances communication and collaboration between domain experts and stakeholders by providing a more interactive platform to discuss simulation results.

The source code of the viewer is currently available on the Advanced Real-time Simulation Laboratory's GiHub page (https://github.com/SimulationEverywhere/CD-WebViewer-2.0). A useable version of the viewer is also available on the laboratory's GitHub pages (https://simulationeverywhere.github.io/CD-WebViewer-2.0/index.html). Sample log files can also be downloaded from the same repository.

2 BACKGROUND

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.

As in classic DEVS, each cell also has functions for internal transition, an external transition, output and time advance. A cell is active as long as it receives external events or it has an internal event scheduled. When there are no more events, the cell passivates. When a cell changes state, its new state will be transmitted to neighboring cells only after a delay. This can be either a transport or an inertial delay. When using transport delays, the new state is guaranteed to be communicated to neighboring cells once the delay expires. Inertial delays provide a mechanism to pre-empt the state change and prevent the cell from transmitting to neighboring cells. This happens when a cell that was scheduled to change state receives an other event that overrides the cell state before the previous transition completes.

The main advantage of merging DEVS and cellular automata is increased performance. Since an influenced cell is only activated when an neighboring influencing cell changes state, unnecessary computations can be avoided. For example, in discrete time based simulation methods, the state of each cell in a cell-space is computed at each regular time step, even if the cell remains stable across the simulation. This potentially results in cycles wasted on computing an outcome that is already known. This optimization arguably comes at a cost of increased complexity for the modeler.

2.1 Visualization and Analysis for DEVS

The main concerns of the modeling and simulation research community are usually improving the modeling process itself, putting forth new models or developing better methods of verification and validation. Visualization and analysis generates limited amounts of research. However, efficient visualization and analysis of the simulation and its results are key to decision-making for stakeholders other than the modeler such as city officials, architects or medical practitioners. Data visualization, outside of modeling and simulation, has begun to generate a more significant amount of research in recent years. Typically, data visualization is focused on providing an abstract representation of large volumes of data in a way that facilitates its interpretation by users and provides deeper insights into the data. Considering the high volume of data that can be generated by simulations, it follows that data visualizations concepts can be applied to the analysis of simulation outputs.

Collins and Knowles Ball (2013) even argue that what matters most is what those outside of the modelling and simulation community can see. Presentation of the simulation's results is more important than the simulation itself. Their reasoning is that, for outside stakeholders, visualization and analysis is the first step in the verification and validation process. It is a way of increasing transparency in modeling and simulation. Although research on the subject is limited, it is starting to generate some interest. For example, Vernon-Bido, Collins and Sokolowski, attempt to define what can be considered an effective visualization for modeling and simulation (Vernon-Bido, Collins and Sokolowski 2015). They notably offer a classification of 4 types of visualizations for modeling and simulation, identify potential pitfalls such as extra details in visual representation or the gap between a perceived relationship in models and the relationship's usefulness. Their conclusion however, is that further research into the topic must be conducted before a definitive way of evaluating the effectiveness of a visualization is established.

Philosophical considerations aside, research on visualization and analysis of simulation results is typically rooted and constrained in specific application domains such as medicine, architecture or energy usage. For example, a research group recently developed a simulation and visualization framework to follow microbial contamination of produce across the supply chain (Zoellner et al. 2019). In this case, the framework relies on differential equations specific to the domain of microbial contamination. It proposes a variety of analytical features to users such as line charts, bar charts, heatmaps, etc. but they are all focused on microbial growth. Another example is CAPSIS, an open-source software designed for forest growth modelling (Dufour-Kowalski et al. 2011). Again, the platform offers a variety of analytical charts, 2D and 3D spatial representations and other visualizations specifically designed for the forestry domain. Examples of this are abundant and a more exhaustive review of them is outside the scope of this paper.

Research dedicated specifically to the visualization and analysis of DEVS simulation results is even more sparse. Most researchers in the DEVS community focus on the development of tools that increase the approachability of the DEVS modeling process itself. One of the goals of the Cell-DEVS WebViewer is to contribute to filling that gap for the DEVS community.

JavaScript is a dynamically typed, interpreted language that follows the ECMA-262 scripting language specification. Because of its omnipresence on the web, it is currently one of the most widely used general purpose programming language (ECMA International 2019). However, because it is an interpreted language, each web browser has to provide its own interpreter for the language which means browsers can be in a different stage of support for the specification. For example, Edge has supported the EcmaScript 6 (ES6) specification since August 2016 while Chrome has supported it from April 2017 (W3Schools 2019). Internet Explorer on the other hand, does not support ES6 and likely never will.

Until the advent of the ES6 specification in 2015, JavaScript classes were an unclear concept in the language. For historical technical reasons, JavaScript is prototype based, an uncommon form of object-oriented programming. Because of this, and the very flexible nature of the language, object-oriented design in JavaScript is not as widespread as in other languages. Developers typically favor a scripting approach. However, the ES6 specification included new keywords that aligned the language much better with other common object-oriented languages such as C# or Java. For example, it now provides a keyword to define a class and to extend them for inheritance purposes. These modifications can be qualified as "syntactic sugar" meaning that they don't change the internal workings of the language but they render it more concise, clearer and simpler.

Besides a class definition syntax, ES6 added many other useful features such as promises for asynchronous actions, get and set accessors for classes, anonymous functions, default values for parameters passed to functions, and many more. Another notable feature that was key to refactoring the software is the possibility to import and export modules from within other modules. Before this feature was available, any dependencies had to be included in the HTML file of the web application or page in the correct loading order. This inconvenience led developers to typically limit the number of files used which in turn, lead to very large, unorganized code files that were difficult to maintain. Using all these features from the new

specification, we were able to refactor the web viewer and follow a more appropriate software architecture which will be presented in details in this section.

3 WEBVIEWER ARCHITECTURE AND USER INTERFACE

The second version of CDWebViewer improves upon the shortcomings of the first one. It provides new analytical and interaction features, a more flexible interface for the user, additional options including the possibility of saving the current user session, etc. Most importantly though, it has been refactored from the ground up into a more modular code base. Although this is invisible for the end user, it was a necessary measure considering the first version lacked a cohesive architecture among its different parts. Indeed the original version used a very loose object-oriented approach, its components were strongly coupled and separation of concerns was lacking. This was likely due to several factors. First, the application was developed very quickly by researchers that did not means to carry the software into the future. Second, features were added haphazardly in a hit-and-run manner by contributors that stepped in and out of the project, without a long-term vision.

The impact of the lack of organization was particularly noticeable when new researchers and developers joined the project; their time-to-productivity was high. Simple improvements required unreasonable time to implement. It also resulted in a code base that was fragile, more difficult to maintain and evolve. Therefore, we opted to completely rebuild the viewer following a more modular, object oriented approach. The end result of this process is software that could possibly be published as a standalone application programming interface (API) that would allow developers to build their own viewers for cellular automata simulations.

3.1 CDWebViewer Architecture

One of the main goals of the second version of the CDWebViewer is to limit the development complexity involved in reconstructing the simulation so that it remains easy to maintain and extend in the future. Our hope is that this will reduce the time-to-productivity for new researchers when they start contributing. As such, we strived to build concise and well defined classes without overlap or code duplication. We also paid particular attention to the naming conventions for methods, functions and property names. The user interface has changed little since version one and was covered extensively in our previous paper about the viewer (St-Aubin, Hesham and Wainer 2018). Previously available features will be discussed summarily and new features will be expanded upon in greater details.

The first step in visualizing simulation results is to drag and drop the simulation files onto the "*dropzone*" widget. Depending on the simulator, there may be multiple files involved. For example, the CD++ simulator uses 4 different files, the initial values file (*.val*), the log file (*.log*), the palette file (*.pal*) and the model file (*.ma*). To lighten the burden on the user, the system automatically detects the type of simulation results being loaded and selects a parsing strategy to read and organize the data in memory. Parsing is done in two steps. The first one is to evaluate the files that were provided by the user. In some cases, the absence of files can be sufficient in determining that a parser is invalid for the files selected. The second step is to read a small portion of the log file using the HTML5 *FileReader* and evaluate the format of the messages.

Since there are no standards for simulation log files, each parser has to provide a way of identifying and validating log files prior to parsing. 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. Once the parsing strategy is identified, it is passed to the *Simulation* object. This is a simple implementation of the strategy pattern, a common pattern of development in software engineering.

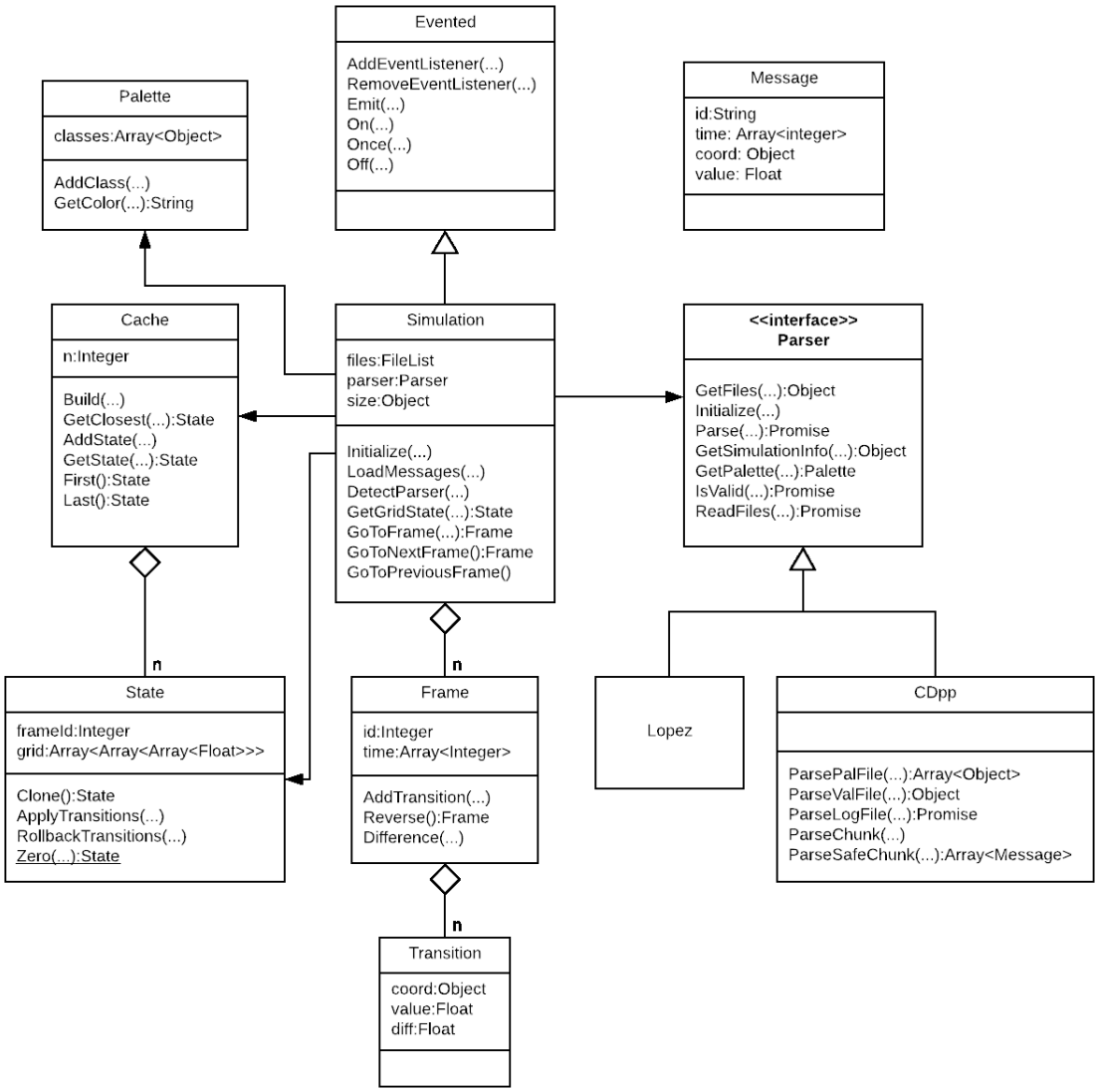

Figure 1 : Class diagram for the CDWebViewer

The *Simulation* class (figure 1) is at the heart of the CDWebViewer, it holds all the data required to visually reconstruct the simulation. It also provides methods and functions to load *.log* messages (*LoadMessages*), to advance (*GoToNextFrame*) or rollback (*GoToPreviousFrame*) time frames and to retrieve the cell-space state at a given time step (*GetGridState*). This class inherits from *Evented*, a class that gives its inheritors the capacity to dispatch events that can be listened to externally. This is useful to decouple the *Simulation* class from the different UI. Widgets in the UI react to events being dispatched by an instantiated *Simulation* object. For example, the *Simulation* object dispatches *Progress* events as the log files are being parsed, the UI reacts and shows a progress bar to the user. It also emits a *Loaded* event that signals the UI that the *Simulation* is ready to be rendered. This class will be referenced throughout this section.

The user then clicks the *Load* button which launches the parsing process. For CD++ files, parsing the *.val* and *.pal* files is straightforward. The *.pal* file is used to create a *Palette* object while the *.val* files provides the cell-space state for the first time step of the simulation. Parsing the *.log* file however, is more complex due to its size. Considering that simulation log files can reach sizes of dozens of gigabytes, reading it as a whole would exceed the memory limitations of the browser. To workaround this issue, we opted to read

files in chunks of approximately 2MB. As chunks of the *.log* file are read, they are parsed and stored into memory. For memory considerations, we do not store the entire cell-space state at each time step but rather store the event-driven state changes between time steps. Each parsing strategy must follow an interface that requires functions such as *IsValid()*, *GetFiles()* and *Parse()* to be implemented. At this point, the CDWebViewer provides parsing strategies for CD++ and the Lopez simulator, the process they follow is summarized in figure 2 below.

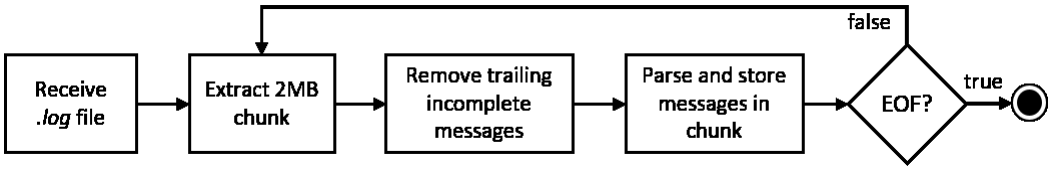

Figure 2 : Conceptual representation of the CD++ parsing strategy

The output of the parsing process is an array of *Message* objects that contains all the cell transitions for the simulation. This array is sorted into an ordered array of *Frames* 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. With all *Messages* parsed, the *Simulation* object builds a *Cache* object of cell-space states. This is done to speed up seeking *Frames* in simulations with a very large number of *Frames*. When the user jumps ahead many frames, the system can determine the closest cached frame and apply the transitions for each *Frame* between the cached *Frame* and the desired one. The caching process consists of storing the entire cell-space state every *n* frame including the first and the last cell-space states. Each cached cell-space state has to be a deep-copy of the current cell-space state otherwise, applying transitions from a cached state would permanently modify its content. The caching process is summarized in figure 3 below.

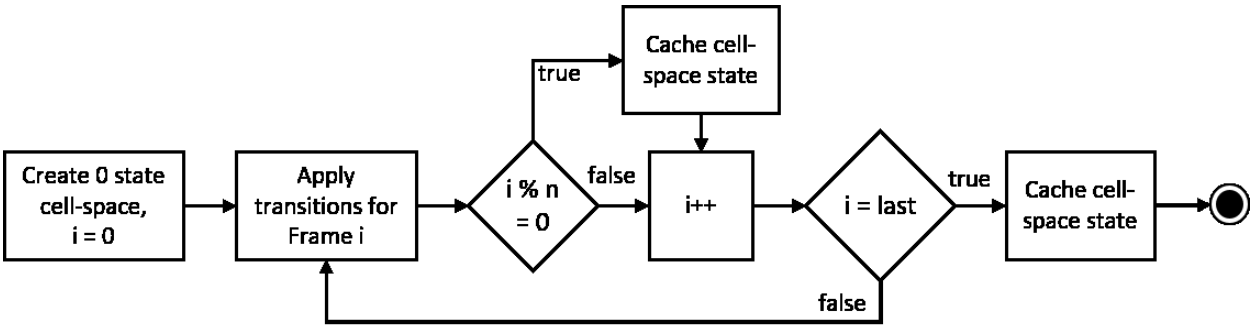

Figure 3 : Conceptual representation of the cache building process, given a cache interval of *n* time frames

The system also calculates the differences between each *Frame* and the previous cell-space state. Because each transition only contains the new state of a cell at a given time step, it is inefficient to reconstruct the simulation backwards. To do so, the system would have to apply all the *Transitions*, from time step 0 to the previous time step sequentially. To avoid this inefficiency, the system calculates and stores, for each *Frame*, the difference between the new cells states and the previous for each *Frame.* In the previous version of the CDWebViewer, the user was only able to go backwards to stored cache states. In the new version, going back to a previous time step is only a matter of adding the state deltas stored on the current *Frame* to the current state of the cell space. The limitation of the previous version has been overcome by this process. The sequence to build the differences is similar to cache building, for every *Frame,* the transitions are applied and the differences are calculated and stored before proceeding to the next *Frame.*

Once this preparation is complete the initial cell-space state is rendered and shown to the user through a *Canvas*, an HTML5 DOM element that behaves in a very similar manner as bitmap images. Summary tests were done to evaluate whether SVG elements were better than a *Canvas* element. Although SVG are

simpler to manipulate programmatically because they do not require any geometry calculations, we found that, when trying to draw very large amounts of polygons, performance decreases significantly. The system also derives a set of information from the loaded files and presents it to the user through the *Info* widget nested within the encompassing *Control* widget. In the previous version, information such as the size of the cell-space, the number of transitions, the number of frames, etc. were determined from the CD++ model (*.ma*) file. In this version it is determined from the *.log* file which renders the model (*.ma*) obsolete. The *Control widget*, shown in figure 4 below, also allows the user to interact with the visual representation of the loaded simulation results.

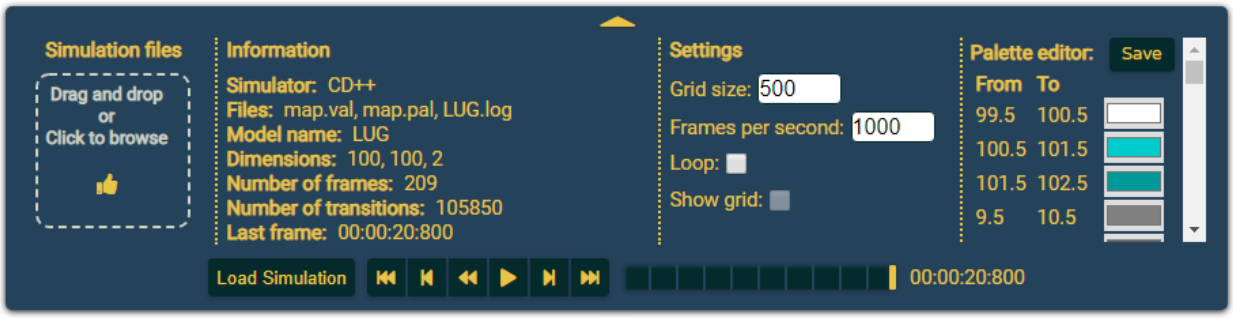

Figure 4 : The *Control* widget. From left to right, the *Dropzone* widget, the *Info* widget, the *Settings* widget and the *Palette* widget. The *Playback* widget is on the bottom row.

Before proceeding with simulation playback, the user is able to 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 *State* class contains the current *Frame* index as well as a 3 dimensional array that holds the current cell-space state. Each value in the 3D array corresponds to an individual cell's state. When the simulation moves forward, the transitions of the next *Frame* are applied to the current cell-space state. This is achieved by looping through each transition of the next *Frame*, finding the corresponding cell of the *State*'s 3D array and replacing its state value by the *Transition*'s new value. This is repeated until all frames have been applied. At this point, if the user has selected to loop the simulation continuously, the process restarts at frame 0. This playback loop is summarized in figure 5 below.

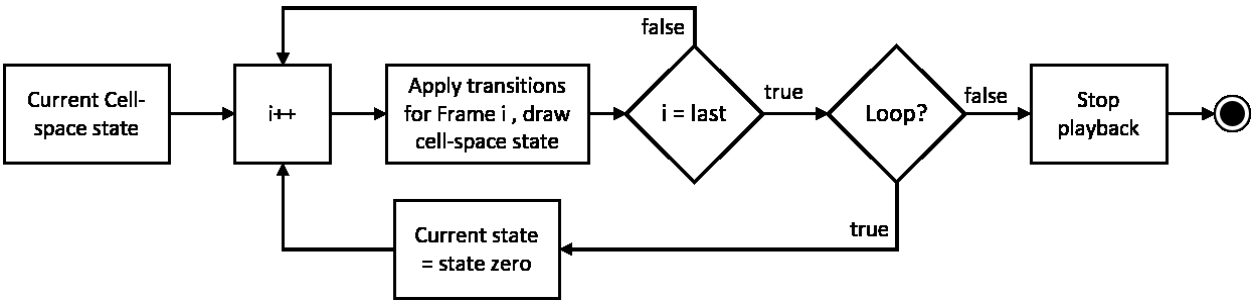

Figure 5 : Playback loop for moving the simulation forward automatically

Moving the simulation backwards is a similar process, the only difference being that instead of replacing the current state cells' values by the *Transition*'s new values, the differences calculated earlier are subtracted from the current value. To simplify the process and future maintenance of the software, it would be preferable to proceed in the same manner to move forward. Seeking a specific time step in the simulation involves different steps. First, the system finds the cached state closest to the one requested then it applies the transitions from the found state until the one requested. This process is summarized in figure 6 below.

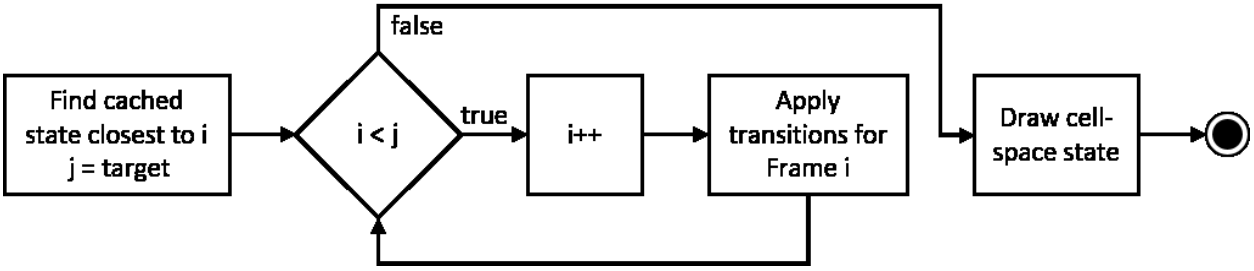

Figure 6 : Process to seek a specific *Frame* at timestep j

This section covered the architecture of the Cell-DEVS WebViewer, the loading and parsing process and its playback features. The application is in a constant state of evolution so it is likely that some of what was presented in this section will change in the near future.

3.2 Analytics

As the viewer plays back the simulation, statistics are compiled and the analytical user interface elements are updated to reflect them. In our previous paper, we presented two such tools, a bar chart showing the total number of cells of each state at the current time frame and a heatmap chart showing the total number of transitions accomplished by each cell at the current time step (See figure 7). These features will not be expanded upon in this paper. Instead, we will focus on the new features and improvements of the viewer's analytical capabilities.

Users of the viewer frequently requested the ability to track the state of user selected cells across the simulation. To achieve this, some groundwork was required since the application did not provide a way for users to select individual cells. Cell selection can be activated by a two state button located on the right-hand side of the canvas element. Once activated, the user can click on individual cells in the canvas to select them. A box will show around the cell and the system will construct the entire time series for the cell. As the simulation is playing back each frame, a dashed vertical line will move across the line chart to show the current frame. Another button next to the canvas allows users to clear their current selection. In a *canvas* element, contrary to an SVG element, individual cells are not individual DOM elements, they do not respond to UI events such as the user clicking on a cell or moving his cursor over it. Event handlers must be attached to the canvas itself and the cell's coordinates must be determined from the cursor's screen position and the grid's geometry.

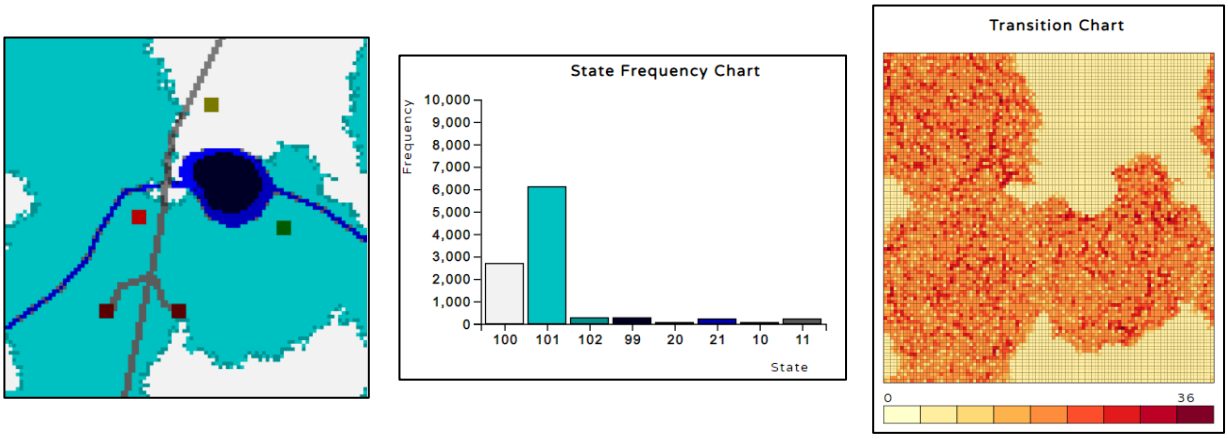

Figure 7 : Analytical features of the viewer. Left, a visual representation of an urban growth simulation. Center and right, the corresponding *State Frequency Chart* and *Total transition chart,* respectively

Constructing the time series associated to a cell's state across the simulation is a straightforward process. It is mostly a matter of looping through each selected cell, verifying that the frame contains a transition for that coordinate and then pushing the new state and the time step to an array associated to the cell. The data structure is then sent to the chart widget to be rendered (See figure 8). To draw the charts, we rely on the data-driven documents (D3) library developed by Michael Bostock. It is currently one of the most widespread JavaScript data visualization library on the web. It provides a series of tools which allow developers to easily inspect, manipulate and transform the document object model (DOM) of a web page (Bostock, Ogievetsky and Heer 2011) and contains a variety of features to draw a chart axes, labels and figures.

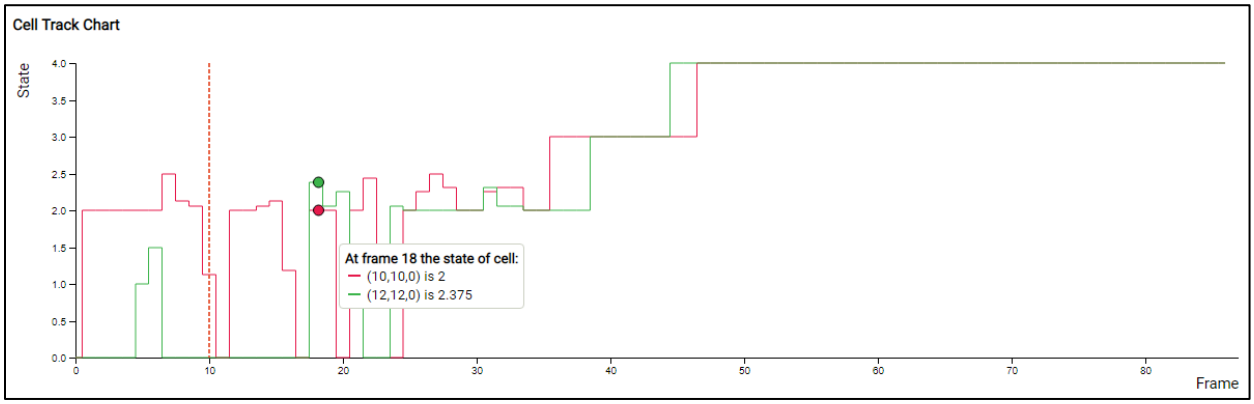

Figure 8 : The cell tracking line chart. A series is shown for each tracked cell. A vertical red dashed line indicates the current frame. As the user hovers over the chart, the state for the tracked cells are shown.

In the current version, the application stores 4 different data structures, one for the canvas, and another for each of the 3 chart features. Because updating these data structures and replaying the simulation in the canvas are computationally expensive operations, performance issues are starting to show. In a simulation with a large cell-space, the user interface is less responsive. Avenues of future development will involve delving into this issue and perhaps determining a way to combine data structures or compute the necessary data in a more efficient manner.

3.3 Integration with a Backend Simulator and Library (RISE)

One of our goals with the CDWebViewer is to support communication and decision-making for stakeholders that are likely unfamiliar and inexperienced with modeling and simulation. These users are generally removed from the modelling and simulation process and do not have access to the software and infrastructure required to design and run their own simulations. As a way to overcome this obstacle, we integrated the viewer with RISE, a RESTful distributed simulation platform that supports DEVS and other formalisms (Wainer and Wang 2017; Al-Zoubi and Wainer 2015).

RISE provides a suite of REST services that users can employ to remotely manage the complete lifecycle of a simulation. It uses the uniform HTTP methods (GET, POST, PUT and DELETE) to manipulate simulation resources on the server side. Through the services, users are able to create a workspace for their simulation, upload simulation files (in CD++, a model *.ma* file), execute the simulation and then download the results. Integration with a distributed simulation platform such as RISE solves two major issues for users. First, it offloads the computationally expensive simulation to a distributed infrastructure; users do not need an access to a high performance computing environment. Second, users do not require the software to be installed on their computers nor do they require the knowledge to operate the software. Their only responsibility is to design the models for the simulation and send them to the RISE platform through the CDWebViewer. This is also beneficial for other stakeholders such as decision-makers. Without being involved in the modeling and simulation process, without any knowledge of the theory, they are able to access results of simulations prepared beforehand by more expert users.

Within the CDWebViewer, there are currently two possible use cases. The first one allows users to run a complete simulation on the RISE platform. The first step is to create a workspace and send the model files to the server. This is achieved by dropping a workspace creation *XML* file and a *Zip* file containing the model files on the *Dropzone* widget then hitting a *Send* button. The system then sends a *PUT* request to send the *XML* payload and a *POST* request to upload the *ZIP* file. Once the request responds successfully, the system send a *POST* request to run the simulation. After the request returns successfully, the system sends a final *GET* request to retrieve the result files, at which point the viewer loads them into the viewer and the user can start the visualization and analysis process.

The second use case is simpler. From a dropdown menu, users select a simulation that has been previously run and hit a *Load* button. At this point, the system sends a single *GET* request to the RISE platform to retrieve the result files. The viewer then loads the results and is ready for visualization and analysis.

Although we successfully managed to integrate communication with the RISE platform in the viewer, there are some issues that limit it's usefulness as a way to democratize access to the M&S. As it currently stands, the RISE platform lacks discovery services, it is difficult to programmatically retrieve a full list of available workspaces. This means that the viewer is limited to a hard-coded list of available simulation results. A second issue is that the results files are missing some of the files required to reconstruct the simulation faithfully. Notably, the initial values file (*.val*) and the palette file (*.pal*) are missing. The consequence is that, when going through the second use case, the first iteration of the simulation is unavailable and the palette, as designed by the modeller, is also unavailable. Finally, sending an *XML* file to create a workspace should not be required. The user should be able to compose the *XML* payload through a more convenient user interface. These are limits that can be easily overcome by modifying the RISE platform but this was outside the scope of this project.

4 CONCLUSION

In this paper, we presented an improved and reengineered version of the Cell-DEVS WebViewer, a lightweight, web-based visualization and analysis platform for results of cellular automata based simulations. The new version of the software proposes a clearer, better organized architecture that follows better practices of object-oriented development. Our hope is that future developers and researchers working

on the project will show a better time-to-productivity and that the system will be easier to maintain and improve.

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 are able to select a previously executed simulation to be visualized in the viewer.

The viewer remains in a state of evolution, we envision various improvements in the near future. As noted in section 3.4, the RISE platform requires some work to provide a more seamless integration with client-side software such as the CDWebViewer. Adding discovery services to RISE would allow non-expert users to easily explore numerous sets of simulation results. Adding support for JSON based responses to the platform would also provide an easier way to integrate it with front-end applications such as the viewer. Another major task ahead is to redesign the user interface so that it is more intuitive to users. To do so, future researchers and developers will refer to the cognitive dimensions of notation put forth by Green (1989) in his seminal work on human computer interaction. Of course, other analytical features could be added to the application as well.

Interacting with simulation results remains an afterthought in most modeling and simulation software and research. The task is typically left to users who end up developing their own solutions. Offering dedicated, easy to access, visualization and analysis tools is one way to democratize access to the field. Integrating a server-side simulator such as RISE and providing an intuitive interface to communicate with it further reduces the barrier to entry.

REFERENCES

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., V. Ogievetsky, and J. Heer. 2011. “D^3 Data-Driven Documents”. in *Journal IEEE Transactions on Visualization and Computer Graphics*, vol. 17, no. 12, pp. 2301-2309.

Collins, A., and A. Knowles Ball. 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.

Dufour-Kowalski, S., B. Courbaud, P. Dreyfus, C. Meredieu, F. Coligny. 2011. “Capsis: an open software framework and community for forest growth modelling”. in *Annals of Forest Science*, vol. 69, no. 2, pp. 221–233.

ECMA International. 2019. “ECMA-262, 9th edition, June 2018”. https://www.ecma-international.org/ecma-262/9.0/index.html, Accessed March 06th 2019.

Franceschini, R., P.-A. Bisgambiglia, L. Touraille, P. Bisgambiglia, and D. Hill. 2014. “A survey of modelling and simulation software frameworks using Discrete Event System Specification”. in *Proceedings of the 2014 Imperial College Computing Student Workshop*, London, UK, pp. 40–49.

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.

Green, T. R. G. 1989. "Cognitive Dimensions of Notations". in *Proceedings of the Fifth Conference of the British Computer Society, Human-Computer Interaction Specialist Group on People and Computers V*, Nottingham, UK.

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

Tendeloo Y. V., and H. Vangheluwe. 2017. "An evaluation of DEVS simulation tools". *Simulation*, vol. 93, no. 2, pp. 103–121.

Vernon-Bido, D., A. Collins, and J. Sokolowski. 2015. "Effective visualization in modeling & simulation". in *Proceedings of the 48th Annual Simulation Symposium*, Alexandria, VA, USA.

W3Schools. 2019. "JavaScript versions". https://www.w3schools.com/js/js_versions.asp, Accessed on March 06th 2019.

Wainer, G., and N. Giambiasi. 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.

Zoellner, C., M. A. Al-Mamun, Y. Grohn, and P. Jackson. 2019. "Postharvest Supply Chain with Microbial Travelers: a Farm-to-Retail Microbial Simulation and Visualization Framework". in *Journal of Applied and Environmental Microbiology*, vol. 84, no. 17.

AUTHOR BIOGRAPHIES

BRUNO ST-AUBIN is pursuing a PhD in Electrical and Computer Engineering at Carleton University where he researches web based simulation and visualization. His email address is bruno.staubin@carleton.ca.

ELI YAMMINE is an undergraduate student in Biomedical Engineering who worked on the Cell-DEVS WebViewer as part of his 4th year engineering project.

MAJED NAYEF. is an undergraduate student in Biomedical Engineering who worked on the Cell-DEVS WebViewer as part of his 4th year engineering project.

GABRIEL WAINER is a Professor at the Department of Systems and Computer Engineering at Carleton University. He is a Fellow of the Society for Modeling and Simulation International (SCS). His email address is gwainer@sce.carleton.ca.