Decoupling visualisation for better DEVS-based simulation applications

Bruno St-Aubin & G. A. Wainer

JOURNAL OF SIMULATION
https://doi.org/10.1080/17477778.2023.2265872

RESEARCH ARTICLE

Decoupling visualisation for better DEVS-based simulation applications

Bruno St-Aubin and G. A. Wainer

Department of Systems and Computer Engineering, Carleton University, Ottawa, Canada

ABSTRACT
Simulation visualisation is an effective way of understanding and communicating complex systems and processes. Among other advantages, it increases model transparency and intelligibility for all categories of users including non-experts, and it can be used by modellers as a tool to debug models in development. However, simulation visualisation is often tightly coupled to specific simulators, and, therefore, there is no way to reuse visualisation tools efficiently. Here, we present a specification that can be used to decouple visualisation engines from simulators. The specification also considers storage optimisation to support web-based simulation applications. We also present an implementation that supports the web-based representation and animation of outputs issued from simulators based on the discrete event system specification (DEVS) and Petri Nets.

ARTICLE HISTORY
Received 23 May 2022
Accepted 27 September 2023

KEYWORDS
Decoupled visualisation; visualisation interoperability; simulation environment

Introduction

Visualisation is an effective way of understanding and communicating complex systems and processes. It increases intelligibility and enhances the tractability of data, processes, or theories for all categories of users, from the layperson to the scientist, expert in the field. The usefulness of visualisation for simulation has been recognised in the field since its early years (Hurrion, 1978), it still generally takes second stage to the advancement of theory, research on performance and application domain-specific model development. In Collins et al. (2015), the authors argue that research mostly focuses on the mechanics of simulation and that, since visualisation does not directly affect the simulation, it is seen as a secondary consideration. Simulation experts typically rely on ad hoc visualisation mechanisms that are specifically built for and tailored to their application domain. For example, modellers will write data processing and visualisation scripts to transform their simulation results into chart-based analytics.

The situation is similar for simulators based on formal methodologies that can be employed in diverse fields of applications. In fact, their visualisation capabilities, when available, tend to be even more limited than those of domain-specific simulators. This could be because these simulators are mostly issued from academic contexts where resources are more limited. Indeed, significant resources and efforts are required to develop comprehensive visualisation, and there is a certain lack of interest from the research community regarding this topic (Collins et al., 2015). For example, CD++ (Belloli et al., 2019; Wainer, 2002), DEVSJAVA (Sarjoughian & Zeigler, 1998), and ADEVS (Nutaro, 2023) are simulators that can be used to simulate a range of systems using the Discrete-Event Systems Specification formalism (DEVS) (Zeigler et al., 2000) but they only offer basic visualisation and analysis capabilities. Simulation software based on non-DEVS formalisms fare similarly. CPNTools, for example, is a Petri Nets simulation software that offers visualisation capability solely based on diagram representations of models (Westergaard & Verbeek, 2018). However, it should be noted that some long-standing simulators based on formal methods and with larger user communities such as OpenModelica (Modelica Association, 2019) offer more extensive capabilities (2D charts, 3D visualisation, and additional options to display analytical charts) (Eriksson et al., 2008; Höger et al., 2012). Commercially available simulation software also tends to fare much better regarding visualisation capabilities. However, their visualisation capabilities generally rely on log files that are either hidden to the user or use proprietary formats. Results cannot be easily imported into other visualisation tools (St-Aubin et al., 2023).

Reusable visualisation tools could fill that gap. A once built visualisation application, compatible with many simulators, and adaptable to different scenarios, would benefit simulation practitioners without requiring them to invest the resources required in building their own dedicated tool. There are additional advantages on top of improving development costs. A well-made, verified, and

CONTACT Bruno St-Aubin gwainer@sce.carleton.ca Department of Systems and Computer Engineering, Carleton University, 1125 Colonel By Drive, Ottawa, Canada

reusable visualisation tool increases transparency in the dissemination of simulation results. The shortcomings of a model cannot be hidden through visualisation since the reusable tool has been used in other contexts and its reconstruction of the simulation trace has been verified to be accurate. Easily accessible visualisation can also serve as a debugging tool for modellers and therefore supports the modelling process. Before reusability for visualisation tools can be achieved, there are several obstacles that must be addressed:

(1) **Issues caused by ad-hoc or user-defined formats for simulation results**. In some cases, the logging format is established by the modellers themselves, leaving even less potential for reusability or interoperability. Similarly, the use of ad-hoc solutions (for instance, JavaScript Object Notation – JSON – combined with Jupyter notebooks or Python scripts) helps with interoperability at the syntactic level, but the semantics of the models which cannot be easily dealt with, reducing the chances for interoperability and reuse.
(2) **Inexistent or inaccessible simulation logs**. In certain cases, the simulation software does not expose simulation results or does so in a generic format that does not follow a clear specification. This can be the case in well-established commercial tools that support the complete simulation lifecycle. An effect of this is decreased transparency because results cannot be easily loaded in other tools, and the consequence is the difficulty for developers to use the best tools available, as well as making end users rely exclusively on a single software to study their models through visualisation or analysis.
(3) **The large volume of simulation logs**. Log formats can be verbose and contain redundant information, leading to needlessly voluminous files. For web-based applications, this can quickly become detrimental to user experience since a browser has limited resources and because these artefacts often need to transit through a network.
(4) **Lack of interoperability for simulation logs**. Simulators rely on specific log formats or user-defined formats to store results. A common format would allow seamless comparison between results of multiple simulators, regardless of the formalism they use. This can be useful for a user that must translate models from one formalism to another. Users should be able to compare results of the original and translated simulations quickly, statistically, and exactly.

The goal of this research is to define a specification that can be used to store simulation results, structure them, and decouple them from the simulator. The specification improves the organisation of the simulation results and the model structure information that is required to interpret such results using systematic processing mechanisms for visualisation. Normally, visualisations are specifically tailored for use cases and need simulation results to be formatted. The contribution of the specification is to improve reuse across multiple domains of applications and simulation scenarios. The main idea is to construct a metadata specification to save all the information needed to reconstruct a simulation trace and support the user in interpreting the model. The specification preserves the components of a simulation model, which involves translating the structural parts of a simulation model into an easy-to-use data structure that can be conveniently manipulated by an application. The specification is focused on the work of the simulation and developer experts and can improve the end user experience. The specification retains a certain level of readability so that it remains approachable by non-simulation experts, which facilitates the development of applications that are decoupled from simulators. The goal is to achieve a balance between completeness and convenience for developers.

Metadata specifications for simulation results are uncommon and, to the best of our knowledge, always proposed in the context of specific application domains (for example, Grunzke et al., 2014). The specification we propose in this research is the first attempt at a generic metadata specification for simulation results in the field. The proposed metadata specification is based on the formal aspects of the Discrete Event System Specification (DEVS) but can support other variants of DEVS as well as other non-DEVS formalisms. We have successfully used it to decouple visualisation for many DEVS simulators as well as a Petri Net simulator as a proof of concept. Another major contribution is the introduction of a method to leverage the specification to visually reconstruct simulation traces for web-based simulation applications. To show the usefulness of the proposed specification and metadata, we developed three different web-based visualisation engine prototypes compatible with various simulators that are presented as case studies.

This paper is organised as follows. Section 2 presents an overview of the current state of visualisation applications for simulation software. Section 3 details the specification we designed to store simulation results and considerations for its implementation in a web-based environment. Section 4 presents an implementation of a prototype DEVS WebViewer, a web application to visualise simulation results. We conclude by reviewing the work and the next steps for this research.

Background

The bulk of research in Modeling and Simulation (M&S) focuses on improving the performance of simulators, developing new simulation formalisms, improving modelling techniques or developing models for specific scenarios (Collins et al., 2015). A disproportionately small amount of research is conducted on the steps executed after a successful simulation run and the role of visualisation in simulation, although most people, except for simulation developers, will only see a simulation through its visualisation (Knowles Ball & Collins, 2012).

Over the past decade, panels of prominent researchers have recurrently identified challenges in modelling and simulation. In Taylor et al. (2013), Loper identifies the role of M&S in the systems engineering lifecycle as a challenge. She notes that intuitive, multi-usage, visual support would provide a common collaboration and decision-making platform for the many actors that take part in an engineering process. In Taylor et al. (2015) and Taylor et al. (2013), Yilmaz noted that the reproducibility of M&S research is a challenge: most models are never independently replicated by anyone but the original developer. Reusable visualisation and analysis tools can help modellers confirm that reproduced models behave as intended. Zander and Mosterman note that the M&S field should capitalise on the fact that citizen developers are increasingly proficient with technology to provide a wider offering of online platforms, mobile applications, etc. A specification for simulation results is a first step towards reusable APIs that citizen developers can use to build web simulation applications.

Visualization and analysis in simulation

Most simulation visualisation research is domain-specific and corollary to the development of simulation models, analytical methods, case studies, etc. It is rarely the main topic in domain-specific research projects. Visualisations and analyses are prepared to extract and convey meaning from simulation traces. Healthcare research is one field where simulation visualisation is commonplace. For example, the CLINSim discrete event simulator and its associated visualisation platform are used to study queues in hospitals (Kuljis et al., 2001). This tool was used to reduce wait times in clinics by allowing non-simulation experts, doctors, and nurses, to understand the impact of their decisions on wait times. The authors note that implementation of visualisation was costly. In Ben-Tovim et al. (2016), the authors introduce a discrete event simulation-based tool for hospital patient flow management with an emphasis on its visualisation capabilities. It relies on a conceptual representation, adapted to real-world hospitals, to visualise patient pathways through different facilities (surgical, emergency, medicine, etc.)

Biology, ecology, and forestry are other fields where rich domain-specific simulation visualisations have been developed. CAPSIS, for example, is an open-source software designed to model and simulate forest growth modelling and yield (Dufour-Kowalski et al., 2012). A graphic user interface allows forest managers to build models considering parameters such as forest area, climate zones, tree species, growth rates, mortality, etc. Simulation traces can be rendered in many ways: analytical charts, 2D or 3D spatial representations. The visualisation elements in CAPSIS are tailored to the forestry domain; they mostly consist of conceptual representations of trees. In Zoellner et al. (2018), researchers present a simulation and visualisation framework to follow microbial contamination of produce across supply chains. The framework relies on differential equations specific to the domain of microbial contamination. The tool does not visually represent supply chains rather, it provides a series of analytical features to users such as line charts, bar charts, heatmaps, etc. Domain-specific visualisation methods are used in many other fields: physics (Stukowski, 2010; Sand et al., 2011), crowd modelling (Al-Habashna & Wainer, 2016; Van Schyndel et al., 2016), construction (Han et al., 2012), marine logistics (Blindheim & Johansen, 2022; Zhao et al., 2019), building engineering (Chen et al., 2017; Hamza & DeWilde, 2014), etc.

In all cases mentioned above, visualisations are specifically tailored for the use case or rely on third party, domain-specific software. This requires that simulation results be formatted according to the format expected by the visualisation or that the visualisation tool be coded in such a way that it can handle the simulation results as they are output. This leads to visualisation engines that are tightly coupled to simulators and difficult to reuse across multiple domains of applications or in different simulation scenarios. To decouple the simulator from the visualisation engine, it is important to define a specification for log files that will contain all the elements required to visualise the model and provide sufficient information for users to understand it.

Research on the structure and organisation of simulation log files is sparse. In Hao et al. (2016), researchers propose an extensible markup language (XML) specification to store simulation and game results as well as a Python application programming interface (API) to read it. They also identify typical issues caused by improvised simulation log files. Parsing unstructured log files is tedious, difficult, and error prone; the number of exceptions that must be handled when processing the log file is generally unpredictable. This specification is intended to support and facilitate the analysis of log files in specific gaming and simulation scenarios. For example, it considers n-gram event sequence matching in both the specification and analysis API. It is not meant to support generic simulation; it lacks flexibility and details required to represent complex simulation models: it is tailored to games than simulations.

Visualisation capabilities for simulations can be categorised in different ways. In Vernon-Bido et al. (2015), four main types of visualisations are identified: concept and diagram visualisation which relies on conceptual models and flowcharts, quantitative visualisation which uses analytical charts and graphs, pattern and flow visualisation that focuses on interaction between elements, and seek and find visualisation which allows users to manipulate data as the simulation occurs. The specification supports all but the last category of visualisation since it is primarily meant for post-simulation visualisation. The last category is more dependent on the simulator being interactive; it must be able to let the user modify the simulation experiment as it is being executed. The specification would likely be compatible with this type of visualisation, but we did not test it on an interactive simulator.

The specification we propose in this paper is for simulations based on DEVS, PDEVS, and other variants of DEVS. With regard to that formalism, visualisation capabilities can be further categorised. In Van Tendeloo & Vangheluwe (2017), the authors focus on evaluating existing DEVS simulation tools and review seven academic tools and a single proprietary, commercial tool. Although visualisation is only one of the seven aspects they evaluate, they still identify five clear stages of visualisation for discrete event-based simulators:

(1) Identification of models and when they are triggered.
(2) Visualisation of a model's state at any given time.
(3) Visualisation of messages exchanged between models.
(4) Identification of a model's internal and external transitions.
(5) Visualisation of the sequence of exchanged messages.

Pitfalls of visualization and analysis for simulation

In Collins et al. (2015), the authors discuss the seeming opposition between the fact that visualisation is a secondary concern for modellers while at the same time often being the only part of a simulation that decision-makers use. Therefore, visualisation can disproportionately influence the understanding of the system and any decision that results from it. They identify four ways in which visualisation can potentially mislead the interpretation of simulation results: the inclusion of extraneous elements, of baseless endogenous elements (visual fluff), inaccurate interpretation (due to a disconnect between representation and model), and accessibility issues (colour blindness or other impairments). In Banks & Chwif (2011), the authors review numerous aspects of modelling and simulation and discuss data collection, model building, verification and validation, analysis, etc. They also suggest that a visualisation should provide a general view of the model and be organised according to several well-defined criteria. Like Roman (2005), they also warn readers to not get confused by fancy graphics that may be misleading. They note that visualisation can support the validation of a model, increase the acceptance of a model by decision-makers and increase sales.

The lack of commonly adopted standards for simulation outputs contributes to the difficulty in achieving interoperable visualisation of simulation results. This has been identified in Shao et al. (2015) where authors cite the size of output data, their heterogeneity, and the fact that they are tightly coupled to the simulation tool as factors that limit their interoperability. The authors of Li et al. (2018) also note that data reduction is increasingly important for simulation visualisation since "the ability to generate and observe data is going up faster than the ability to store data". Since there is no commonly adopted standard for logging messages that are output over the course of a simulation, there is also no standard way of consuming them in a visualisation. Analysts must rely on custom scripts to read log files into their preferred tool and programming language. Typical data science tools such as Jupyter or Tableau allow users to write scripts that convert complex, verbose, sometimes redundant log files into a format that can be plotted more easily as analytical charts or visual reconstructions of their simulation. In a context where data science is omnipresent, where citizen developers dispose of increasingly diverse tools to consume data, it makes sense that a clear specification to archive simulation results would facilitate post-simulation visualisation and analysis. In addition, increasingly complex simulation models lead to simulation outputs that grow in volume considerably, making it important to consider the optimisation of the size of the artefacts to consider limits of web-based software and for storage in general.

When authors discuss pitfalls of simulation visualisation, they rarely offer tools to alleviate or avoid them, the implication being that the burden is on the software developer. A common specification for simulation outputs is an opportunity to address some of these pitfalls. A well-known and well-documented specification provides a way to decouple visualisation from the simulator. This makes it possible for different developers to build dedicated visualisation platforms that are independent from a simulator. Independent visualisation platforms would be agnostic to the limits of a model or a simulator and, therefore, would lessen the potential for a modeller to compensate their shortcomings either intentionally or unintentionally. This is a way to increase transparency in the presentation of simulation results and, therefore, avoid some pitfalls identified previously. For accessibility issues, which can be expensive to address in software development, reusing a single visualisation platform previously evaluated for accessibility could reduce costs associated with the development of an ad hoc, accessible visualisation tool. These aspects are discussed in the following sections, where we introduce a specification to support web-based visualisation of simulation results with a focus on platforms that use DEVS, PDEVS, and other variants of the DEVS formalism.

The DEVS formalism

DEVS (Discrete EVent Systems Specification) (Zeigler et al., 2000) is a formalism to describe systems whose states change either upon the reception of an input event or due to the expiration of a time delay. Unlike the discrete-time simulation approach, DEVS uses a continuous time base and allows for asynchronous model execution, improving the efficiency of the simulation without losing accuracy. Based on general dynamic systems theory, DEVS provides a sound M&S framework to define hierarchical discrete-event models in a modular way, where a system is described as a composition of behavioural (atomic) and structural (coupled) components. An atomic model is defined as follows:

$$M = <X, S, Y, \delta_{int}, \delta_{ext}, \lambda, ta>$$

where X is the input events set, S is the state set, Y is the output events set, δ_{int} is the internal transition function, δ_{ext} is the external transition function, λ is the output function, and ta is the time advance function. Figure 1 shows states and variables in DEVS models.

As shown in Figure 1, the semantics of DEVS models are as follows. Each atomic model has input

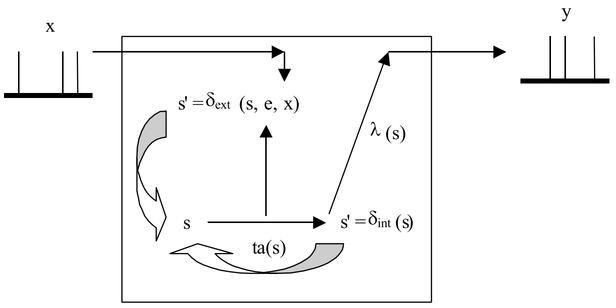


Figure 1. DEVS semantics.

(X) and output (Y) ports to communicate with other models. Every state (S) in the model is associated with a time advance (ta) function, which determines the duration of the state. Once the time assigned to the state is consumed, an internal transition is triggered. At that moment, the model execution results are spread through the model's output ports by activating an output function (λ). Then, an internal transition function (δ_{int}) is triggered, producing a local state change. Input external events are collected in the input ports. An external transition function (δ_{ext}) specifies how to react to those inputs. A coupled model groups several DEVS components into a compound model that can be regarded, due to the closure property, as a new DEVS model. This allows hierarchical model construction. When external events are received, the coupled model must redirect the inputs to one or more components. Similarly, when a component produces an output, it may have to map it to another component, or as an output of the coupled model itself.

The simulation is executed in a message-driven fashion. CD++ (López & Wainer, 2004; Wainer, 2002) and Cadmium (Belloli et al., 2019) are two open-source environments that support both standalone and parallel/distributed simulation of DEVS models. The environments provide two major frameworks: a modelling framework that allows users to define the behaviour of atomic and coupled models using a built-in graph-based specification language or C++; and a simulation framework that creates an executive entity for each component in the model hierarchy to carry out the simulation in line with the formalisms. CD++ messages fall into two categories: content messages include the external message (X) and output message (Y) that encode the actual data transmitted between the models, while control messages include the initialisation message (I), collect message (@), internal message (*), and done message (D) that are used to synchronise the simulation. During the simulation, all messages exchanged between the models are recorded in log files. Log files for DEVS simulators exhibit non-negligible differences and a common specification would simplify and clarify the process of loading the results in memory for the visualisation software.

The two examples in the figures below illustrate this issue through log file excerpts for DEVS models in CD ++ (Figure 2) and Cadmium (Figure 3). Figure 2 shows that, for CD++, the log files consist of the different messages discussed above stored in a text file. Each line starts with "Message" and its type followed by the current simulation time. The last part of the message includes the simulation source and destination (from – to), and their corresponding internal code, specified by a unique integer number. In the case of output messages (Message Y), we also include the

```
Message I / 00:00:00:000 / Root(00) to top(01)
...
Message * / 00:00:00:000 / cpu(05) to npc(10)
Message Y / 00:00:00:000 / npc(10) / out2 / 1.000 to cpu(05)
Message D / 00:00:00:000 / npc(10) / 00:00:10:000 to cpu(05)
Message X / 00:00:00:000 / cpu(05) / in2 / 1.000 to pc_latch(11)
...
```

Figure 2. A fragment of a CD++ log file for a DEVS model.

```
00:02:40:000
[iestream_input_defs<Message_t>::out: {16 0}] generated by model
input_reader
00:02:43:000
[Subnet_defs::out: {16 0}] generated by model subnet1
...
```

Figure 3. A fragment of a Cadmium log file for a DEVS model.

port name through which the output is transmitted (in the figure, out2) and a value (in this case, 1.000). Similarly, the input messages record the port used, as well as the value of the input received. Finally, the Done (D) messages include the time to the next scheduled internal transition for the model (in this case, 10:000 s).

For Cadmium, the log is a text file that shows all the message bags generated by the atomic models every time the simulator collects the outputs. At simulated time 2 min 40 s, input_reader generates an output message {16 0} that is transmitted through the output port out (iestream_input_defs<Message_t>::out:), which is retransmitted by the coupled subnet1 at 2 min 43 s. This coupled model uses the output port out of the coupled model subnet1.

Supporting web-based visualisation of DEVS simulation

As noted previously, current DEVS simulators present various issues with regard to the results they output. One important problem is that most of them do not output the model structure. Consequently, it is very complex to write a reusable visualisation engine since there is no data structure to associate visual elements (for example, SVG nodes) to simulation model components. In addition, there is no contextual information provided about the values output by models: numbers alone convey little meaning to end users. Another important issue is that simulation results are output in a verbose format with needless redundancy.

In the CD++ excerpt presented in Figure 2 previously, each line is a single message, and we can see that there is a considerable amount of unneeded data. First, there are a series of messages that are not useful for visualisation. In fact, only the messages beginning with *Message Y* contain relevant data about the messages output by models. We can also see that there are additional words such as *Message* and extra whitespace that were added for human readability of the log files. Each output message contains the destination of the message after *to* which is not required since we can deduce the destination from the model structure. There are other elements that are redundant, such as model names, port names, and time advance values. These can be eliminated in different ways using a proper data structure as shown in the next section. Cadmium messages can be similarly reduced to their minimal expression required for visualisation. In both cases, significant gains in file size can be achieved, but the magnitude of the gains depends on the models. For example, a model that outputs many messages per period will show a significant gain since time values will not be stored for each message.

Description of the specification

The specification stores all the information required to reconstruct a simulation trace and support the user in interpreting the model. To achieve this, it is important for it to preserve all the components of a simulation model. This involves translating the structural parts of a simulation model into an easy-to-use data structure that can be conveniently manipulated by an application. As discussed earlier, typical simulator output formats focus on the messages exclusively and provide no information on the structure of the model. There is no information available to relate model components to one another. With simulation outputs alone, it is nearly impossible to illustrate the path that a message travels within the model, to reconstruct a message as intended by the modeller or to provide contextual information about a message emitter. We also attempt to retain a certain level of simplicity and readability so that it remains approachable by non-simulation expert software developers. This is meant to facilitate the development of simulation-based applications that are

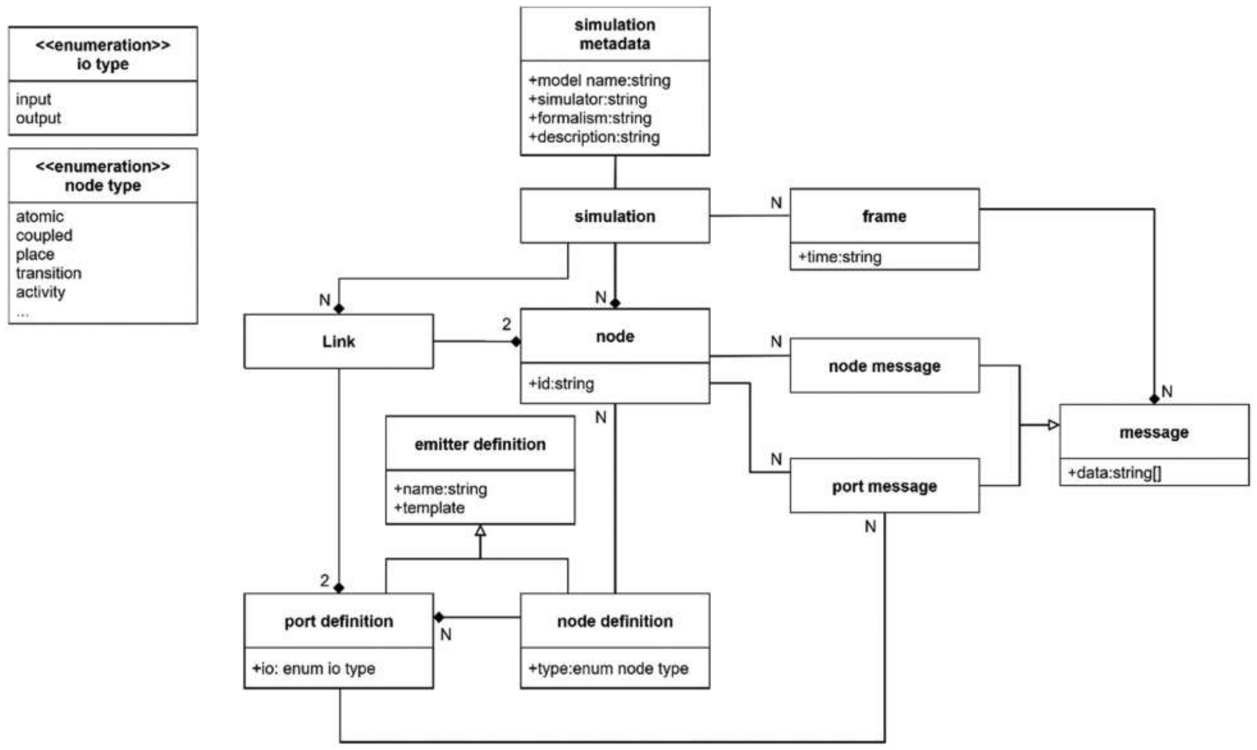


Figure 4. UML representation of the specification.

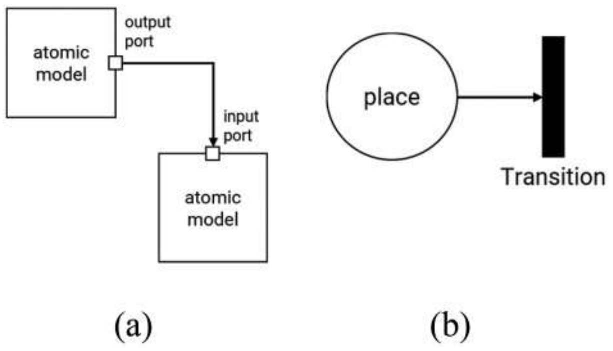


Figure 5. (a) In DEVS, a link connects two model ports and (b) in Petri Nets, a link relates places and transition nodes.

decoupled from simulators. The goal is to achieve a balance between completeness and convenience so that developers can appropriate the structure easily.

A conceptual data model of the specification is shown in Figure 4. The data structure captures a *simulation* which is composed of the structural elements of a simulation model and the messages that it outputs over its execution. The *simulation* itself should be associated with *metadata*. Although it is not used to reconstruct the simulation, it serves to provide context to the user. Metadata should contain information that provides context to developers and end users. At the time of writing, the *metadata* class is not well defined, it should be the subject of additional research. In other fields, GIS for example, metadata is usually the subject of its own specification. As a placeholder while a more comprehensive *metadata* definition is defined, we consider a simple *metadata* object containing the name of the simulation model, the name of the simulator employed, the name of the formalism, and a summary description of the model.

Structural elements of a model are stored in two lists that compose the model structure: *nodes* and *links*. Although the specification was initially designed for DEVS, PDEVS, and other variants of DEVS, it can also represent other graph-based formalisms. That is, *nodes* and *links* can represent different concepts depending on the formalism. In DEVS for example, a *node* is used to represent a model, atomic or coupled, while a *link* is a coupling between two *ports* (Figure 5, left). In the case of a Petri Net, *nodes* would be places or transitions while a *link* would be equivalent to an arc between them (Figure 5, right). In typical Petri Net models, *ports* are not necessarily named, but it is still important to distinguish them since tokens travel by specific arcs. *Ports* allow us to determine which arc should be activated at a given time in the visualisation.

Regardless of the formalism, each *node* in the *nodes* list must provide a unique *id* for identification purposes and must be associated with a *node definition* which indicates the type of node to use for that *Node*. In a DEVS model for example, multiple models may share the same model definition. These models will share the same characteristics, they will have the same ports, output the same message types, etc. In large simulation scenarios, there can be many thousands of instances using the same *node definition*. To avoid repeating these common characteristics, *node definitions* are captured once and associated to multiple *node* instances. A *node definition* also stores a type which varies according to the formalism employed. In DEVS, a *node definition* type would be either *atomic* or *coupled*. In Petri Nets, it would be either a *place* or *transition*. *Node definitions* are also composed of a list of *port definitions*, each with a type that can be *input* or *output*. Both *port definitions* and *node definitions* are instances of *emitter definitions*. *Emitter definitions* are used to represent the structural elements through which messages can be output. An *emitter definition* contains a name used for labelling purposes and a template used to reconstruct the messages that are emitted by a *node* port or a *node* as will be explained further in this section. In DEVS, for example, a simulator can log messages that contain the state of a model at a given time: these types of messages are associated with *nodes*. It can also log messages that are output by a model: these are associated to a model port which is represented by the combination of a *node* and a *port definition* from the *node*'s list of *port definitions*.

The second list contains *link* objects that store the origin and destination *ports*. Since *ports* are associated with *nodes*, storing a *link* requires storing a reference to the *node* and a reference to a *port definition* that exists in the *port definitions* associated to the *node*.

The specification also contains a list of *frames* that represent each time step of a simulation trace. Each frame contains a time step value which is expressed as a string since time representation can vary greatly according to formalisms, simulators, and even simulation models. A *frame* also contains one or more *messages*. An individual *message* can be a *node message* in which case, the *message* will be associated to a *node* element through the *node* attribute. It can also be a *port message* in which case, it will be associated to a port element through the *node* and *port* attributes. This provides the flexibility required to visualise a static message related to a *node* (for example, a model that emits its state in DEVS) or a message that travels through a link from a *port* (for example, a model that outputs data towards another model in DEVS). This also allows us to store both *node* and *port* messages in the same list. Some simulators, such as Cadmium, store these messages in different files which requires the timeline of messages to be reconstructed when they are read. Storing both message types in the same list reduces the processing required by the application when reconstructing the simulation. Regardless of its type, each message also has a data attribute which stores a list of strings that must be templated to represent the value of the message.

Many simulators use complex formats to log their messages, regardless of whether they originate from a node or a port. Adding to that complexity is the fact that the structure of a message can vary by model and by port of a model. The message templating process in the specification provides a way to reduce the repetitive and often verbose messages output by simulators to the essential data values they contain. This is another measure meant to minimise the size of output files. At runtime, the software reconstructs the full messages so that the context required to interpret the message is not lost for the end user. To achieve this, each *message emitter*, *node definition*, or *port definition*, must provide a distinct template to conserve and rebuild the messages at runtime. Templates are strings that contain template substitution sequences. The sequences provide a simple mechanism to inject the list of data values contained in a message into the template. The templates and substitution patterns can take any shape. They could be, for example, fully formed sentences or serialised objects using a syntax such as JSON or XML. An example of message reconstruction is shown in the next section where we present a potential implementation of the specification.

The specification presented contains the required data to clearly implement three of the four types of visualisation defined in Vernon-Bido et al. (2015). A graph structure, such as the simulation structure data model presented before, can easily be displayed as a flowchart or other types of conceptual model. As seen before, when processing messages for visualisation, if it is associated to a port, then it is possible to identify the corresponding link through which it travels. It is therefore possible to visualise patterns and flows in the execution trace. Quantitative visualisation is relatively simple, it is a matter of extracting the relevant data points from the messages to build an analytical chart or graph. The specification does not offer explicit support for the seek and find capability but, as argued before, this is more a matter of simulator implementation than specification of simulator output artefacts.

In the context of DEVS simulations, if a simulator output provides the required data, then this specification unambiguously supports all five levels of visualisation defined by Vangheluwe and Van Tendeloo (Van Tendeloo & Vangheluwe, 2017). Messages are associated with a model or a port therefore, it is possible to show a triggered model and messages travelling between models by identifying links associated to ports. Since it is also possible to store the state of a model in a message associated to a model node, it is possible to display that state. Similarly, if the simulator logs a message when a model triggers an internal or external transition, it will be possible to display it. Finally, by sequentially displaying all port messages stored in the data structure, an animated sequence of inter-model communication can be visualised.

Considerations for the implementation of the specification

The data structure presented in the previous section provides the minimal information required to reconstruct a visualisation of a simulation trace that fulfils the different categories and levels of visualisation discussed before. However, when implemented in a software application, we need to consider various additional issues, particularly when used in web-based applications. Web-based applications are convenient for users: they are lightweight, do not require installation, can run on any machine that has an internet connection and access to a browser, etc. However, they also suffer from many drawbacks: computing resources are limited within a browser, memory is also limited, and transferring files over a network is generally a bottleneck for such an application. These issues were considered when the specification was defined.

The data model in Figure 4 contains a sub-structure to hold structural elements, message emitters and links, and another sub-structure to hold messages organised by frames. We developed a prototype implementation using JavaScript Object Notation (JSON) to represent the former and a CSV-like format for the latter. JSON provides us with flexibility and readability which are useful to represent the structure of a model at the cost of more a verbose result. This is acceptable to represent the structure of the model since its size is generally insignificant compared to the messages, particularly when precaution is taken to avoid repetition. To store messages, we considered other well-known formats such as eXtensible Markup Language (XML) or again, JSON. However, they are verbose and, therefore, lead to large file sizes that can be prohibitive for web consumption. JSON, for example, requires field names to be repeated for each entry. XML requires that opening and closing tags be repeated for each entry. Furthermore, both XML and JSON formats must be entirely read before being processed. Reading in "chunks" will lead to improperly formatted fragments. Due to browser limitations and the volume that messages can reach, a format that can be read line by line is convenient, if not mandatory. Here is an example of the specification implementation we designed for the visualisation platform that will be discussed in the following section (Figure 6). For clarity's sake, the example is abridged:

On the left-hand side, we find the metadata object which contains the name of the model, the simulator and formalism used, and a short description. We then find a list of node_definitions for which the correspondence with the specification is straightforward: each element has the attributes of an emitter definition: name, type, and template as well as a nested list of port definitions. Port definitions are also emitter definitions and, therefore, share the same attributes. The next component of the implementation is a list of nodes where each element has an id to identify the node as well as

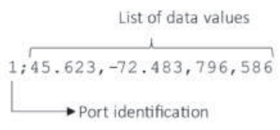
```
{
   "metadata": {
      "model": "ABP",
       "simulator": "CDpp",
       "formalism": "DEVS",
       "description": "A simple network protocol ..."
   },
   "node_definitions": [{
      "name": "network",
      "type": "coupled",
      "template": null,
      "port_definitions": [{
         "name": "out1",
         "type": "output",
         "template": null
      }, ...
      ]
   }, ...
   ],
   "nodes": [{
      "id": "subnet1",
      "node_definition": 0
   }, {
      "id": "subnet2",
      "node_definition": 0
   }, ...
   ],
   "links": [[0, 2, 1, 0], [1, 1, 2, 0], ...]
}
```

```
00:00:20:000
1,1;11.00000
1,3;1.00000
00:00:22:987
4,1;11.00000
00:00:32:987
3,1;1.00000
00:00:50:000
1,1;11.00000
1,3;1.00000
00:00:51:957
4,1;11.00000
00:01:01:957
3,1;1.00000
00:01:04:992
5,1;1.00000
1,4;1.00000
00:01:14:992
1,1;20.00000
1,3;2.00000
00:01:17:174
4,1;20.00000
00:01:27:174
3,1;0.00000
00:01:44:992
1,1;20.00000
1,3;2.00000
00:01:48:841
4,1;20.00000
```

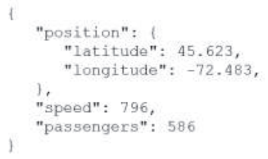
Figure 6. An implementation of the specification (abridged). On the left side, the structural elements, including message emitters and links (JSON). On the right, a list of messages output by the models in the simulation (CSV derived).

```
{
   'name': 'airplane_1',
   'type': 'atomic',
   'template': '{"position": {"lat":0, "lon":1},"speed":2, "passengers":3}'
   'port_definitions': [...]
}
```

(a)



(b)



```
{
  "position": {
    "latitude": 45.623,
    "longitude": -72.483,
  },
  "speed": 796,
  "passengers": 586
}
```

(c)

Figure 7. (a) A sample node definition, with template, (b) a single state message, and (c) message reconstructed from a template.

a node_definition attribute which contains an index that indicates the position of the corresponding node definition the node_definitions list. By storing node definitions separately instead of nesting them within each node, we avoid repetition. Finally, it contains a list of links where each element is an array of four indices. The first and second indices indicate the origin port and the third and fourth indicate the destination port. For each couple, the first index indicates the position of the node in the nodes list, while the second indicates the position of the port definition in the port definitions list associated to the node definition of the node. Links are easy to reconstruct programmatically, and the format requires minimal storage space.

On the right-hand side, we show an example of messages. Here, lines with a single string contain a new time step and consequently, identify the beginning of a new frame. Each following message is assigned to the same frame until another time step is found. Figure 7 below explain how each message can be associated to a node or a port and how the emitter definition template can be used to reconstruct the complete message:

Each individual message in a frame can be associated to either a port or a node. The numbers on the left-hand side of the semi-colon identify the emitter by index. For a node message (state message in DEVS), a single number will indicate the position of the node in the nodes list. For a port message (output message in DEVS) two numbers are necessary, one to represent the position of the node, same as state messages, and another to indicate the position of the port definition in the port definitions list associated to the node type of the node. This added complexity is necessary to uniquely identify ports since port names are not unique within a model: many node types can have ports with the same name. This allows us to reference ports and models without using string identifiers. This significantly reduces the size of the resulting file since these values are repeated many times across a simulation output. Although this makes the log files less human-readable, they remain easy to reconstruct programmatically. In cases where the consuming software requires a properly nested structure (e.g., port instances nested within models), it is trivial to reassociate them.

(b) above represents a state message for the airplane_ 1 model (a). Knowing its state message template, the list of data values contained in the message can be injected into the corresponding template. The implementation we propose relies on a serialised JSON structure where each substitution sequence is an integer that references the index of the data point in the list of data values associated to each message. The first step to reconstruct the message is to deserialize the template into a JSON object. The resulting tree-like object is then traversed, and each leaf is replaced by the corresponding data value from the list of data values a message contains as in (c). This is a simple operation since each number indicates the position index of the value in the list of data values. Once this procedure is complete, the data message is restored. In this case, we can see, for example, the latitude and longitude positions of the airplane, its speed, and the number of passengers it carries. The data, now replaced in context, offer more comprehensive options for visualisation and interaction. It can now be explained to the end users rather than only presented without context.

Case study: The specification in the DEVS WebViewer

Using the specification, it is possible to build automated and generic visualisation tools. This section introduces different case studies as examples of what can be built using the specification and it shows how to build visualisation solutions for simulation applications. First, the DEVS WebViewer visualisation platform, presented in St-Aubin & Wainer (2019) and St-Aubin et al. (2018), was used as a testbed for the

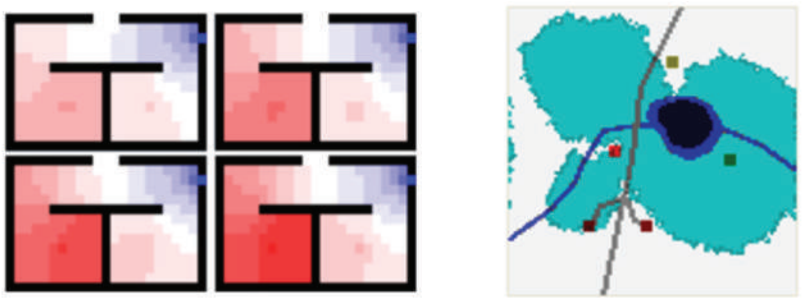
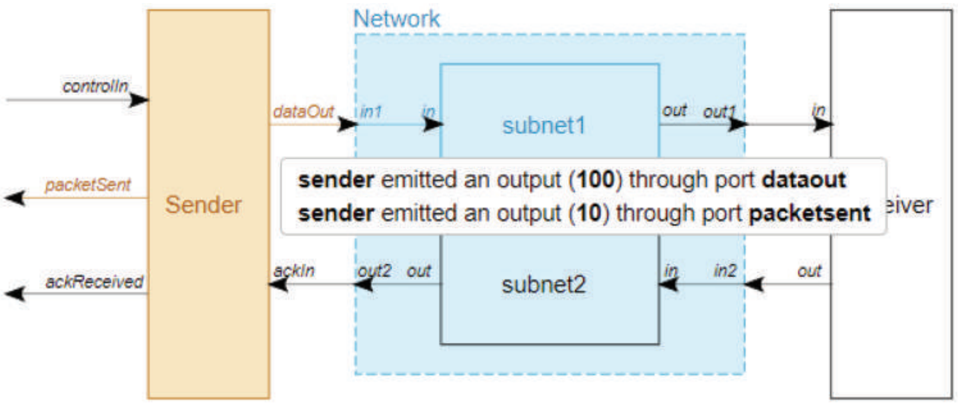


Figure 8. Visualizing simulations with the DEVS WebViewer. Left, a classroom CO2 simulation (Khalil et al., 2020) and right, a logistic urban growth simulation (St-Aubin & Wainer, 2019).



sender emitted an output (**100**) through port **dataout**
sender emitted an output (**10**) through port **packetsent**

Figure 9. Alternate bit protocol: DEVS model simulated using CD++ and visualized using the DEVS WebViewer.

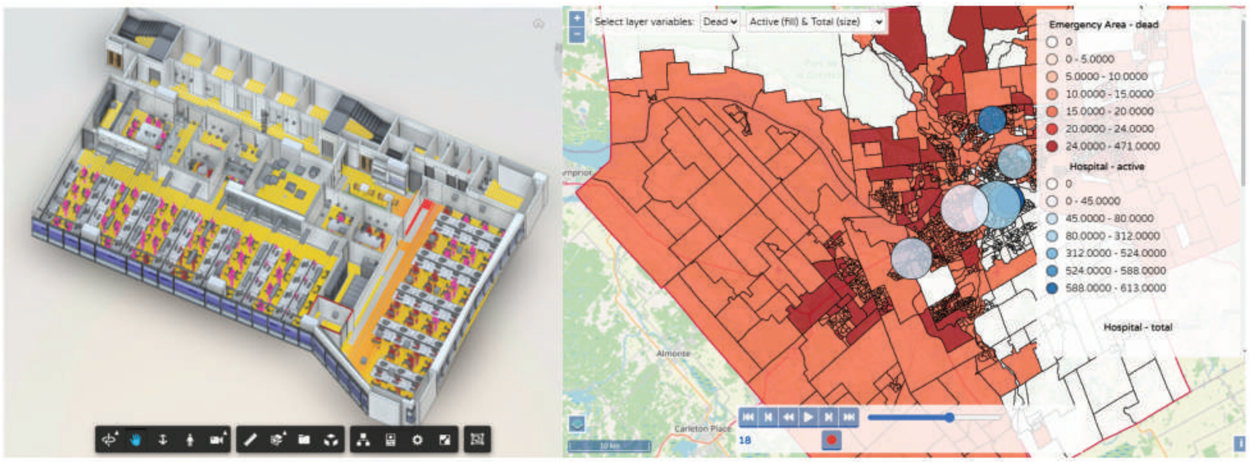


Figure 10. Two visualizations of disease spread models. The indoors model on the left is an integration of our API and the Autodesk Forge Viewer. The geospatial model on the right, an integration with the OpenLayers API.

specification and its implementation. It is a lightweight, web-based software that allows users to visualise, interact with, and analyse their simulation results. Some of the work supported by the specification and the visualisation platform is shown in Figure 8. In the image on the left, we can see a model to study the placement of CO2 sensors indoors. In this visualisation, red cells indicate higher concentrations of CO2, while blue cells indicate lower concentrations of the gas. This model was built using a simulator derived from CD++ (López & Wainer, 2004). In the image on the right, we show a simulation of urbanisation based on the logistic equation. Here, teal cells represent urbanised areas, darker teal cells represent areas undergoing urbanisation and other coloured cells represent different spatial features such as highways, rivers, or other points of interests. This model was implemented using the CD++ simulator. In both cases, the simulation outputs were converted to the specification discussed before, then visualised using the DEVS WebViewer. Generating the results in the specification can be achieved in different ways. The simulator can output in the specification directly or results can be converted from the simulator format to the specification format. In our case study, node types must also indicate the size of grid space since these are cellular model simulations. These details were omitted from the previous discussion for clarity's sake.

A modular API underlies the WebViewer application. One of its modules is an object-oriented data structure analogous to the specification that is used to contain and manipulate the execution trace of the simulation provided in the format specified before. Other modules offer various components and user interface elements that developers can use to build their own visualisation platform. Reconstructing the nested data structure and associating the messages to structural elements are straightforward. The first step is to parse the structure file, an operation natively supported by modern browsers. Once read, reconstruction of the object-oriented hierarchy can be achieved as follows:

Figure 9 above is an example of a CD++ DEVS model that represents a network where a sender sends a packet that transits through a network, is received by a receiver and an acknowledgement message is sent back to the sender. At runtime, the viewer uses the components of the specification to reconstruct the simulation trace. With the emitter id associated to each message of a given simulation frame, the viewer can relate the message to the element that emitted it, model ports in this case, and highlight the element. Here, we can see that the sender model is outputting a packet through its dataOut port going towards the network model that receives it through its in1 port then transfers it to its inner subnet1 model through it in port. At the same time, the sender model is outputting a package it received at a previous time step through its packetSent port. The diagram component that highlights the simulation trace and tooltip popup bubble that allows users to explore output messages in detail.

```
FOR each node in the structure
  instantiate a node object
  find node type by index in the structure
  assign the node type to the node
END FOR

FOR each link in the structure
  instantiate a link object
  find start and end nodes by index in the structure
  find start and end ports by index in the port definitions of each node
  assign nodes and port definitions to link
  add the link to the start node's list of links
END FOR
```

Messages contained in the messages file must be read line by line where each line represents a single frame in the simulation. Individual messages can be associated with their corresponding structural element following the procedure explained before.

With the data structure, reconstructed from the simulation results provided following the specification, it is possible to achieve the five levels of DEVS visualisation identified by Van Tendeloo in Van Tendeloo & Vangheluwe (2017) (triggered models, model's state, messages exchanged, internal and external transitions, sequence of messages). The DEVS WebViewer, however, focuses on animating the messages exchanged between models. To achieve this, it relies on a vector-based representation (SVG) of the simulation model where each graphic element is associated to a structural element in the data structure. The application then determines the path travelled by output messages using the data structure and highlights it on screen by changing the colour of corresponding the vector elements as shown below.

Other visualization platforms using the specification

As explained before, one of the major advantages of relying on a specification for simulation outputs is that a single visualisation platform can be reused for multiple simulators and formalisms. A corollary advantage of this is that case-specific visualisation platforms can be designed and developed quickly by reusing components of the base visualisation platform. This requires that the base platform from which components will be reused be built following best practices of software engineering. Components must be modular and as loosely coupled as possible so that they can be taken out of their context and brought into another one. The API underlying the DEVS WebViewer is based on elements of the modern web: it is modular and follows a rigorous object-oriented approach. Therefore, it is possible to reuse only parts of the API to build use case-specific applications based on the specification. Figure 10 shows an example of this:

The visualisation presented on the left is an indoor disease spread model built using the Cadmium Cell-DEVS simulator (ARSLab, 2023). It uses a module of the API to load the simulation data into the application and then uses the Autodesk Forge Viewer to render a 3D scene of the simulation trace. It also reuses a playback component to animate the visualisation and allow the user to step through all frames. Since the data structure holding the simulation results is event-enabled and the playback component handles these events, including this functionality in a new visualisation platform is trivial. Similarly, the example on the right is an integration of these modules with the OpenLayers API, a library to build web-based geographic information systems. The resulting visualisation shows disease spread at the city scale. Each polygon is a subdivision of space for which we used census of population statistics to build a simulation model. In this case, the simulation is run with another version of Cadmium Cell-DEVS which supports Irregular neighbourhoods and cell shapes.

Conclusion

In this paper, we discussed issues related to domain-specific visualisation for simulation with a specific emphasis on DEVS-based simulation. Users generally cannot reuse visualisation platforms across simulators and in different application scenarios. In most cases, they will build *ad hoc* visualisations to suit their needs. This is an obstacle to transparency in simulation, one of the better-known pitfalls of simulation. To address this issue, we propose to decouple visualisation from simulation models, simulators, and even simulation formalisms when possible. This can be achieved by establishing a specification for simulation results. We presented one such specification and an implementation that was successfully integrated to the DEVS WebViewer to visualise results from three DEVS simulators (CD++, Cadmium, and ADEVS) and a Petri Net Tool (Colored Petri Net Tools).

Through this platform and the specification that supports it, the advantages of a common result specification were made clear. Users can quickly and without effort generate visualisations for their models, regardless of which simulator they were using. This allows users to immediately debug models, to share results among researchers, to communicate results externally, to introduce newcomers to the theory of DEVS, to compare models reimplemented from other simulators, etc. In accordance with advantages discussed in the previous section, this contributes to faster model development cycles, increased collaboration, and higher visibility for the work accomplished.

The work accomplished demonstrates that such a specification does not have to be overly complex or elaborate, as long as it properly fulfils visualisation needs. In fact, the clearer and simpler it is, the more likely it is to be adopted and used by the community. However, more important than the specification and the implementation we presented is the argument itself that visualisation and simulation should, in fact, be decoupled. Indeed, providing better tools for model developers, simulation software developers, and consumers of simulation products is a major step towards democratisation of the field. A once developed, always ready to use visualisation platform could have radical impact on the simulation lifecycle. Modellers would avoid spending effort and resources on developing their own simulation visualisations. They could use common platforms to analyse, debug, compare, demonstrate, and share them with others. This would speed up the simulation development cycle and allow modellers to focus on modelling rather than the development of tools. This is particularly relevant in an academic context where resources are often limited. Any tool that supports the specification could be easily reused by modeller. A rich ecosystem of robust visualisation tools would emerge gradually and be at the disposal of the community.

The research presented is currently being integrated into a wider environment meant to manage the complete simulation lifecycle. As such, it will play a major role in the automated integration of simulation results into a decision-making platform. With regard to visualisation decoupled from simulation, there are still various aspects left to explore. More efforts should be spent to verify how the specification can be made to support an even wider range of formalisms, for example, system dynamics, business process modelling, finite state machines, etc. We specifically plan on better integrating cellular automata-based models into the specification. Although we have made efforts to minimise the storage required for output files that follow the specification and anecdotal evidence shows that it reduces the size drastically, more experimentation would be required to quantitatively evaluate the gains more clearly. This could also be an occasion to measure the impact on the processing time required to reconstruct simulation traces at runtime.

Acknowledgments

The authors would like to acknowledge Compute Canada for providing access to their Arbutus cloud-based infrastructure that allowed us to develop and test an implementation of the work we presented. The research was partially funded by NSERC.

Disclosure statement

No potential conflict of interest was reported by the author(s).

Funding

This work was supported by the NSERC - Canada.

References

Al-Habashna, A., & Wainer, G. A. (2016). Modeling pedestrian behavior with cell-DEVS: Theory and applications. *Simulation*, *92*(2), 117–139. https://doi.org/10.1177/0037549715624146

Arslab (Advanced real-time simulation laboratory). (2023). SimulationEverywhere, BIM-to-DEVS GitHub repository. https://github.com/SimulationEverywhere/BIM-to-DEVS

Banks, J., & Chwif, L. (2011). Warnings about simulation. *Journal of Simulation*, *5*(4), 279–291. https://doi.org/10.1057/jos.2010.24

Belloli, L., Vicino, D., Ruiz-Martin, C., & Wainer, G. A. (2019). Building devs models with the Cadmium tool. In *Proceedings - Winter Simulation Conference* (pp.45–59). National Harbor, Maryland: IEEE.

Ben-Tovim, D., Filar, J., Hakendorf, P., Qin, S., Thompson, C., & Ward, D. (2016). Hospital event simulation model: Arrivals to discharge–design, development and application. *Simulation Modelling Practice and Theory*, *68*, 80–94. https://doi.org/10.1016/j.simpat.2016.07.004

Blindheim, S., & Johansen, T. A. (2022). Electronic navigational charts for visualization, simulation, and autonomous ship control. *IEEE Access*, *10*, 3716–3737. https://doi.org/10.1109/ACCESS.2021.3139767

Chen, Y., Liang, X., Hong, T., & Luo, X. (2017). Simulation and visualization of energy-related occupant behavior in office buildings. *Building Simulation*, *10*(6), 785–798. https://doi.org/10.1007/s12273-017-0355-2

Collins, A. J., Ball, D. K., & Romberger, J. (2015). A discussion on simulations' visualization usage. *2015 Winter Simulation Conference (WSC)*. (pp.2827–2835). Huntington Beach, California: IEEE.

Collins, A. J., Ball, D. K., & Romberger, J. (2015). Simulation visualization issues for users and customers. *Simulation Series*, *47*(2), 17–24.

Dufour-Kowalski, S., Courbaud, B., Dreyfus, P., Meredieu, C., & de Coligny, F. (2012). CAPSIS: An open software framework and community for forest growth modelling. *Annals of Forest Science*, *69*(2), 221–233. https://doi.org/10.1007/s13595-011-0140-9

Eriksson, H., Magnusson, H., Fritzson, P., & Pop, A. (2008). 3D animation and programmable 2D graphics for visualization of simulations in OpenModelica. 49th Scandinavian Conference on Simulation and Modeling (SIMS'2008) (pp. 184–195). Oslo, Norway.

Grunzke, R., Breuers, S., Gesing, S., Herres-Pawlis, S., Kruse, M., Blunk, D., Garza, L., Packschies, L., Schäfer, P., Schärfe, C., Schlemmer, T., Steinke, T., Schuller, B., Müller-Pfefferkorn, R., Jäkel, R., Nagel, W. E., Atkinson, M., & Krüger, J. (2014). Standards-based metadata management for molecular simulations. *Concurrency & Computation: Practice & Experience*, *26*(10), 1744–1759. https://doi.org/10.1002/cpe.3116

Hamza, N., & DeWilde, P. (2014). Building simulation visualization for the boardroom: An exploratory study. *Journal of Building Performance Simulation*, *7*(1), 52–67. https://doi.org/10.1080/19401493.2013.767377

Han, S. H., Al-Hussein, M., Al-Jibouri, S., & Yu, H. (2012). Automated post-simulation visualization of modular building production assembly line. *Automation in Construction*, *21*(1), 229–236. https://doi.org/10.1016/j.autcon.2011.06.007

Hao, J., Smith, L., Mislevy, R., von Davier, A., & Bauer, M. (2016). Taming log files from game/Simulation-based assessments: Data models and data analysis tools. *ETS Research Report Series*, *2016*(1), 1–17. https://doi.org/10.1002/ets2.12096

Höger, C., Mehlhase, A., Nytsch-Geusen, C., Isakovic, K., & Kubiak, R. (2012) Modelica3D - Platform independent simulation visualization. *Proceedings of the 9th International MODELICA Conference*, 3-5, *Munich, Germany*, 76 (August 2015): 485–494

Hurrion, R. D. (1978). An investigation of visual interactive simulation methods using the job-shop scheduling problem. *The Journal of the Operational Research Society*, *29*(11), 1085. https://doi.org/10.1057/jors.1978.240

Khalil, H., Wainer, G. A., & Dunnigan, Z. (2020) Cell-DEVS models for CO2 sensors locations in closed spaces. In *Proceedings of the 2020 Winter Simulation Conference* (pp. 12). Orlando Florida.

Knowles Ball, D., & Collins, A. J. (2012) Simulation visualization rhetoric and its practical implications. *48th Annual Meeting of Southeastern Chapter of INFORMS*. Myrtle Beach, SC, pp.584–591

Kuljis, J., Paul, R. J., & Chen, C. (2001). Visualization and simulation: Two sides of the same coin? *Simulation*, *77* (3–4), 141–152. https://doi.org/10.1177/003754970107700306

Li, S., Marsaglia, N., Garth, C., Woodring, J., Clyne, J., & Childs, H. (2018). Data reduction techniques for simulation, visualization and data analysis. *Computer Graphics Forum*, *37*(6), 422–447. https://doi.org/10.1111/cgf.13336

López, A., & Wainer, G. A. (2004). Improved cell-DEVS model definition in CD++. 6th International Conference on Cellular Automata for Research and Industry (ACRI 2004) (pp. 803–812). Amsterdam, The Netherlands. https://doi.org/10.1007/978-3-540-30479-1_83

Modelica Association. (2019). Modelica tools. https://www.modelica.org/tools/index_html#commercial-modelica-simulation.

Nutaro, J. J. (2023). Adevs: A discrete EVent System simulator. https://web.ornl.gov/~nutarojj/adevs/

Roman, P. A. (2005) Garbage in, Hollywood out. *Proceedings SimtecT* (pp. 2–6). Sydney, Australia.

Sand, A., Kniivilä, J., Toivakka, M., & Hjelt, T. (2011). Structure formation mechanisms in consolidating pigment coatings—simulation and visualisation. *Chemical Engineering and Processing: Process Intensification*, *50* (5–6), 574–582. https://doi.org/10.1016/j.cep.2010.09.006

Sarjoughian, H. S., & Zeigler, B. P. (1998) DEVSJAVA: Basis for a DEVS-based collaborative M&S environment. *SCS International Conference on Web-Based Modeling and Simulation* (pp. 7) San Diego, CA, USA.

Shao, Y., Liu, Y., & Li, C. (2015). Intermediate model based efficient and integrated multidisciplinary simulation data visualization for simulation information reuse. *Advances* *in Engineering Software*, *90*, 138–151. https://doi.org/10.1016/j.advengsoft.2015.08.002

St-Aubin, B., Hesham, O., & Wainer, G. A. (2018). A cell-DEVS visualization and analysis platform. SummerSim-SCSC 2018 (pp. 157–168). Bordeaux, France.

St-Aubin, B., Loor, F., & Wainer, G. A. (2023). A survey of visualization capability for simulation environments. 2023 Annual Modeling and Simulation Conference (ANNSIM) (pp. 13–24). Hamilton, ON, Canada.

St-Aubin, B. & Wainer, G. A. (2019). A cell-DEVS model for logistic urban growth. SpringSim-ANSS: Tucson, AZ, USA.

Stukowski, A. (2010). Visualization and analysis of atomistic simulation data with OVITO–the open visualization tool. *Modelling and Simulation in Materials Science and Engineering*, *18*(1), 015012. https://doi.org/10.1088/0965-0393/18/1/015012

Taylor, S. J. E., Balci, O., Cai, W., Loper, M. L., Nicol, D. M., & Riley, G. F. (2013) Grand challenges in modeling and simulation. In *Proceedings of the 2013 ACM SIGSIM conference on Principles of advanced discrete simulation - SIGSIM-PADS '13* (pp. 403). ACM Press.

Taylor, S. J. E., Khan, A., Morse, K. L., Tolk, A., Yilmaz, L., & Zander, J. (2013) Grand challenges on the theory of modeling and simulation. In *Proceedings of the Symposium on Theory of Modeling & Simulation*. (pp. 1–8). San Diego, California.

Taylor, S. J. E., Khan, A., Morse, K. L., Tolk, A., Yilmaz, L., Zander, J., & Mosterman, P. J. (2015). Grand challenges for modeling and simulation: Simulation everywhere—from cyberinfrastructure to clouds to citizens. *Simulation*, *91*(7), 648–665. https://doi.org/10.1177/0037549715590594

Van Schyndel, M., Hesham, O., Wainer, G. A., & Malleck, B. (2016) Crowd modeling in the sun life building. *Proceedings of SimAUD 2016*. London, UK.

Van Tendeloo, Y., & Vangheluwe, H. L. (2017). An evaluation of DEVS simulation tools. *Simulation*, *93*(2), 103–121. https://doi.org/10.1177/0037549716678330

Vernon-Bido, D., Collins, A. J., & Sokolowski, J. A. (2015). Effective visualization in modeling and simulation. 48th Annual Simulation Symposium (ANSS '15) (pp. 33–40). San Diego, CA, US.

Wainer, G. A. (2002). CD++: A toolkit to develop DEVS models. *Software: Practice & Experience*, *32*(13), 1261–1306. https://doi.org/10.1002/spe.482

Westergaard, M., & Verbeek, H. M. W. (2018) CPN tools – a tool for editing, simulating, and analyzing colored Petri nets. http://cpntools.org/

Zeigler, B. P., Praehofer, H., & Kim, T. G. (2000). *Theory of modeling and simulation* (2nd ed.). Elsevier.

Zhao, M., Yao, X., Sun, J., Zhang, S., & Bai, J. (2019). GIS-Based simulation methodology for evaluating ship encounters probability to improve maritime traffic safety. *IEEE Transactions on Intelligent Transportation Systems*, *20*(1), 323–337. https://doi.org/10.1109/TITS.2018.2812601

Zoellner, C., Al-Mamun, M. A., Grohn, Y., Jackson, P., Worobo, R., & Schaffner, D. W. (2018). Postharvest supply chain with microbial travelers: A farm-to-retail microbial simulation and visualization framework D. W. Schaffner ed. *Applied & Environmental Microbiology*, *84*(17), 1–13. https://doi.org/10.1128/AEM.00813-18