ENABLING COLLABORATIVE MODELING THROUGH A WEB LIBRARY OF DEVS MODELS

Hamza Qassoud
Bruno St. Aubin
Gabriel Wainer
Cristina Ruiz-Martin

Department of Systems and Computer Engineering
Carleton University
1125 Colonel By Drive
Ottawa, ON, CANADA
{HamzaAmhad, BrunoStAubin}@cmail.carleton.ca
{gwainer,cristinaruizmartin}@sce.carleton.ca

ABSTRACT

Conducting accurate simulation experiments is complex and time-consuming, and collaboration between domain-specific experts is crucial. Collaborative modeling can help to build and maintain a shared understanding of the problem space and the potential solution. Here we introduce a framework that enables collaborative modeling between technical and non-technical experts. This is achieved through a web application that provides a continuous integration system to modelers where they can share, collaborate, comment, organize, and revise their experiments. This is intended to simplify the simulation and modeling process due to the complex nature of the problem and the lack of compatibility between models and modeling tools. The web-based platform supports users in the development of domain-specific models, providing a straightforward way to simulate the models and analytical capabilities for comprehensive decision-making.

Keywords: Collaborative Modeling, Model Integration, Model Library, DEVS.

1 INTRODUCTION

Simulation modeling is a widely used methodology to study real-world systems and support managerial or technical decision-making through the use of models as a basis for simulations. Conducting accurate simulation experiments is a complex and time-consuming process that requires extensive domain-specific knowledge. For instance, simulating how a certain type of cancer propagates throughout the body requires both technical simulation and modeling knowledge as well as medical and biological expertise. Furthermore, an inefficient collaboration between domain-specific experts or between modelers is the main reason for delayed time-to-market, poor quality products and high cost of development. To succeed in today's market, modelers need a continuous integration system where they can share, collaborate, comment, organize, and revise their experiments.

Collaborative modeling refers to the use of well-known requirements analysis and modeling techniques in a collaborative manner to build and maintain a shared understanding of the problem space and the potential solution. The main premise is that requirements models, which have long been viewed as documentation

techniques, can also be put to great use as elicitation and analysis techniques in a collaborative setting with the delivery team and stakeholders jointly discussing the problem and solution.

Collaborative modeling starts by first defining the problem space; the goal here is for the team to have a mutual understanding of the context in which a problem occurs (i.e., the "problem space") and how potential solutions might fit into this problem space. Secondly, the team can use collaborative modeling to define a specific solution and provide a foundation for the team to identify implementation options. The models used in this context help identify features and user stories based on a common full understanding of the solution. Lastly, collaborative modeling can be used to define specific aspects of the solution, which is further describing the backlog items.

This work presents a framework that enables collaborative modeling between modelers, technical and non-technical experts. This is achieved through a web application that provides a continuous integration system to modelers where they can share, collaborate, comment, organize, and revise their experiments. This web application greatly speeds up the modeling process while reducing costs and improving results. This is done through a generic and convenient API that tracks documented experiments, modeling artifacts, the list of contributors, and results related to the experiments. This web application also enables a common repository where all modeling data and results can be stored reliably through data redundancy and versioned according to experiment needs and length.

The rest of the paper is organized as follows: in Section 2 we discuss related work in the area; Section 3 discusses the design and implementation of the distributed library of models; Section 4 presents a case study for managing hospital use under emergencies and the deployment of results in a Geographical Information System. Section 5 provides a conclusion.

2 BACKGROUND

The Discrete Event System Specifications (DEVS) is a simulation methodology derived from systems theory (Zeigler, Praehofer , and Kim, 2000). It is meant to model and simulate discrete-event dynamic systems and was first described by Bernard Zeigler in 1976. It provides a discrete event-based method to abstract systems into models that can be used for experimentation in cases where it is impractical or impossible to experiment on the real system. DEVS supports hierarchical and modular model development and provides a mechanism to couple models together. In doing so, it strongly favors the reusability of models and their composition into larger models of more complex systems.

DEVS follows what is known as component-based modeling, which involves grouping de-coupled, self-contained, and easily re-useable models to form more complex systems. It is a popular approach in engineering disciplines because it simplifies a complex problem into smaller subproblems, thus improving collaborative modeling by allowing engineers or modelers to work on independent subsystems rather than working on a shared model. Component based modeling has many other benefits such as a more robust code, less context-switch, makes onboarding other engineers or modelers much easier, and most importantly simplifies and enables the possibility of implementing a generic model that can be re-trained on a different subset of data and re-used across these sub models thus facilitating modeling automation.

In addition to the various issues that surround the modeling process itself such as the lack of re-usability of models and their lack of compatibility across different platforms and programming languages, there is also no standard for model outputs. Simulation results are formatted in an ad-hoc manner that depends on the simulator employed (St-Aubin, Loor, and Wainer, 2021). Hence it becomes much harder to extract insights from simulation results since there is not a common platform that can interpret them.

Due to the nature of the problem and the high cost of development, researchers try to lessen the burden on modelers. CosMoS for example is a popular tool that guides modelers through the process using a visual programming language (Sarjoughian and Elamvazhuthi, 2009). Modelers that use CosMoS only need to

code model behaviors in Java, then they can connect them through a frontend platform that then generates the structural elements of a more complex model (i.e. the top-model).

DesignDEVS is a more recent example of a visual graphic user interface meant to support the modeling process (Goldstein, Breslav and Khan, 2016). In a similar manner, this software allows modelers to pick pre-implemented models from a list, configure them through a UI and assemble them into a larger coupled model. It also provides debugging tools such as a timeline of output messages and a summary visualization capacity for cellular automata models. It provides a mechanism to alter the behavior of models using LUA, a simple scripting language. Although these modeling and simulation environments effectively support the simulation lifecycle, they are built as monolithic software: they combine modeling, simulation, and visualization into a single application. Users must remain within those applications to conduct the entire simulation experiment. This locks them into a specific simulator, and they have no possibility of customizing visualizations or analyses to cater the needs of a scenario specific decision-making process.

Representational state transfer (REST) is an application programming interface that conforms to the REST architectural constraints, like stateless communication. An API is a mediator between the users and the resources they want to get or persist. REST APIs are one of the most common kinds of web services available today. Twitter for example uses REST APIs to post and share tweets (Twitter, 2021). They allow various clients including browsers to communicate with a server via REST. Therefore, it is very important to design REST APIs properly and to follow the commonly accepted conventions so that the clients that use them can have an easier time understanding the services provided instead of them being different from what everyone expects.

Masse (2012) sets some guidelines for Common REST API Design. The guidelines include:

Accept and respond with JSON: JSON is the standard for transferring data, hence our APIs should accept JSON for request payload and also send responses in JSON. Server-side technologies or JavaScript have libraries and build-in methods that can decode and encode JSON without doing much work.

Use nouns instead of verbs in endpoint path: REST verbs specify an action to be performed on a specific resource or a collection of resources, example of these verbs are POST (create), PUT (update), GET (retrieve) and DELETE. Using verbs in endpoint paths does not add any new information. Hence, we should use the nouns which represent the entity that we are retrieving or manipulating in the pathname. For example, “POST /api/createUsers” can be shortened to “POST /api/users” since the meaning of “create” is embedded in POST.

Use logical nesting on endpoints: It is advisable to avoid mirroring the database model in the endpoints design to avoid giving attackers unnecessary information. For example, if we want an endpoint to get the comments for an experiment, we should append the /comments path to the end of the /experiments path.

Handle errors gracefully and return standard error codes**:** We should return appropriate HTTP response codes that indicate what kind of error occurred to help the user debug the failure. Common error HTTP status codes include:

- 400 Bad Request: The server could not understand the request due to invalid syntax.
- 401 Unauthorized: Although the HTTP standard specifies "unauthorized", semantically this response means "unauthenticated". That is, the client must authenticate itself to get the requested response.
- 403 Forbidden: The client does not have access rights to the content; that is, it is unauthorized, so the server is refusing to give the requested resource. Unlike 401, the client's identity is known to the server.
- 404 Not Found: The server cannot find the requested resource. In the browser, this means the URL is not recognized. In an API, this can also mean that the endpoint is valid but the resource itself does not exist. Servers may also send this response instead of 403 to hide the existence of a resource

from an unauthorized client. This response code is probably the most famous one due to its frequent occurrence on the web.
- 429 Too many requests: The user has sent too many requests in a given amount of time ("rate limiting").
- 500 Internal server error: The server has encountered a situation it does not know how to handle.
- 502 Bad Gateway: This error response means that the server, while working as a gateway to get a response needed to handle the request, got an invalid response.
- 503 Service Unavailable: The server is not ready to handle the request. Common causes are a server that is down for maintenance or that is overloaded. Note that together with this response, a user-friendly page explaining the problem should be sent.

Allow filtering, sorting, and pagination: The databases behind a REST API can be very large. Hence it is strongly advisable to return a subset of data to avoid fetching large amounts of it and returning it through the network which can cause high latencies and low throughput and can bring down our systems.

Maintain good security practices: A lot of the communication between client and server contains private information. Therefore, using SSL/TLS which establishes a secured encrypted link between the client and the server is necessary. Furthermore, appropriate roles should be assigned to users and checked when resources are being requested. For example, a normal user should not be able to access data of administrators.

Al-Zoubi and Wainer (2013) applied these concepts to design and developed the RESTful Interoperability Simulation Environment (RISE), the first existing distributed simulation platform built following the Representational State Transfer (REST) architecture. Although REST is now a reasonably well-established practice in modern web-based applications, at the time, it was still in its infancy. RISE's main feature is to expose any simulator as an easily accessible REST web service. The platform sought to, and accomplished four clear goals:

- Decouple the design and implementation while supporting composition scalability. It allowed any number of systems to join or leave the environment as needed.
- Serve as a container for any other simulation environment and therefore, be independent from any given simulator implementation, synchronization algorithm or formalism.
- Provide an experiment-oriented environment through archival of experiment related artifacts.
- Bring web-based resources within the simulation loop.

RISE exposes its tools through a uniform interface, consumers access them using a set of resource oriented Uniform Resource Identifiers (URI). Client applications execute REST service calls and, upon reception on the server-side, they are routed through a tree-like conceptual structure to determine the correct functions to call on the server. RISE stores the results of any simulation it executes under workspaces assigned to individual users. To this end, users must be authenticated through a standard username and password combination. The admin branch of the platform contains the resources required to achieve this. Users can be listed, created, deleted or updated, passwords can be created, deleted or updated. It also contains other resources to manage the server configuration or logging subsystem. The other notable branch is the one that manages simulation experiments. Through variable portions of the URI, users can specify their workspace and simulator to use (serviceType) to run their model. Again, they can list, create, update or delete experiments. Requests are sent using the GET, POST, PUT or DELETE methods depending on the type of operation to accomplish and, when necessary, users send data in XML format. However, a library of models is not available.

In (St-Aubin, Menard, and Wainer, 2021), the authors made a first attempt to create a library of models. This work builds on top of that research.

3 LIBRARY OF MODELS: DESIGN AND IMPLEMENTATION

In this research, we focus on designing and implementing a library of models. The library of models solves the problem statement that modelers need a continuous integration system where they can share, collaborate, comment, organize, and revise their experiments. This in turn speeds up the experimentation process, improves the communication and results, and lowers the cost of operation.

Figure 1 presents a conceptual model of the system containing the entities listed below.

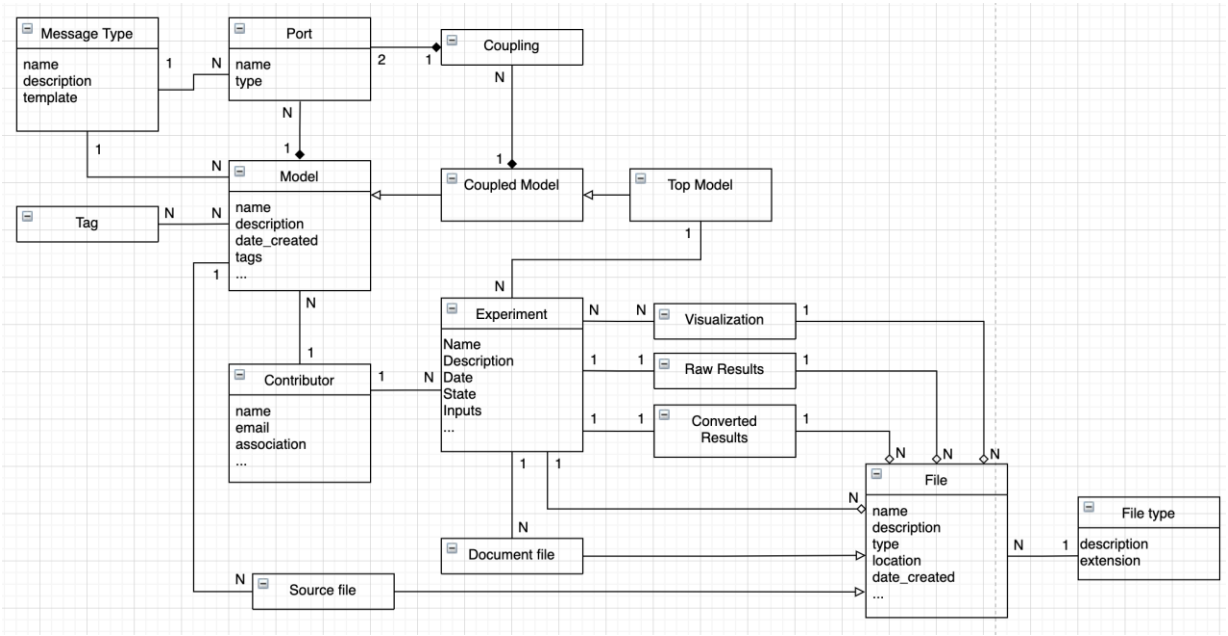

Figure 1: Conceptual Model.

Experiment: represents information about the experiment that is being conducted such as its description, the date of when the experiment was conducted, and its state (e.g. COMPLETED, IN_PROGRESS, INITIAL).

Contributors: These are the people who contributed to an experiment (first name, last name, contact information, etc.)

Model: This is the atomic model and represents simulation data for a single entity, which is the simulation object in the real world (e.g. hospital, car crash statistics, geographic data for a certain area of the world, etc.)

Coupled model: Can be composed of one or more coupled or atomic models

Top Model: This is the top level model of the experiment and is a 1-to-1 mapping to it, it can be composed of one or more sub models (i.e. "coupled models") each sub model representing a subset of the overall data.

Ports: These are available input/outputs locations that models can use to exchange information between each other. That is model #1's output port can be linked to model #2's input port and looking at these two models from a black box perspective is what forms a coupled or top-level model.

Message type: This describes the data template of the ports, which is the format of the data that the port will be reading or writing and a description of what each field means.

Source file: This is the C++ code that modelers use to read the raw data as a goal to extract insights from it.

Tag: This is a way to label models to improve the model search experience for users. (e.g. makes it easier to retrieve all models associated to a healthcare tag).

Visualization files: Determines how to display the simulation data to the user, used by frontend applications that know how to interpret them. An experiment can contain many visualization files, potentially used for different ways to view the same data.

Raw results: Represents all the simulation insights (e.g. this area of the city does not have good healthcare).

Converted results: Users have the ability to change the simulation insights format after the fact if they wish.

Files: Every visualization file, raw result and converted result is a file and that contains a description, a creation date and must specify a file type.

File Type: Determines the extension of the file (XML, JSON, Binary, text, etc.).

3.1 Database Model

The database model is presented in figure 2.

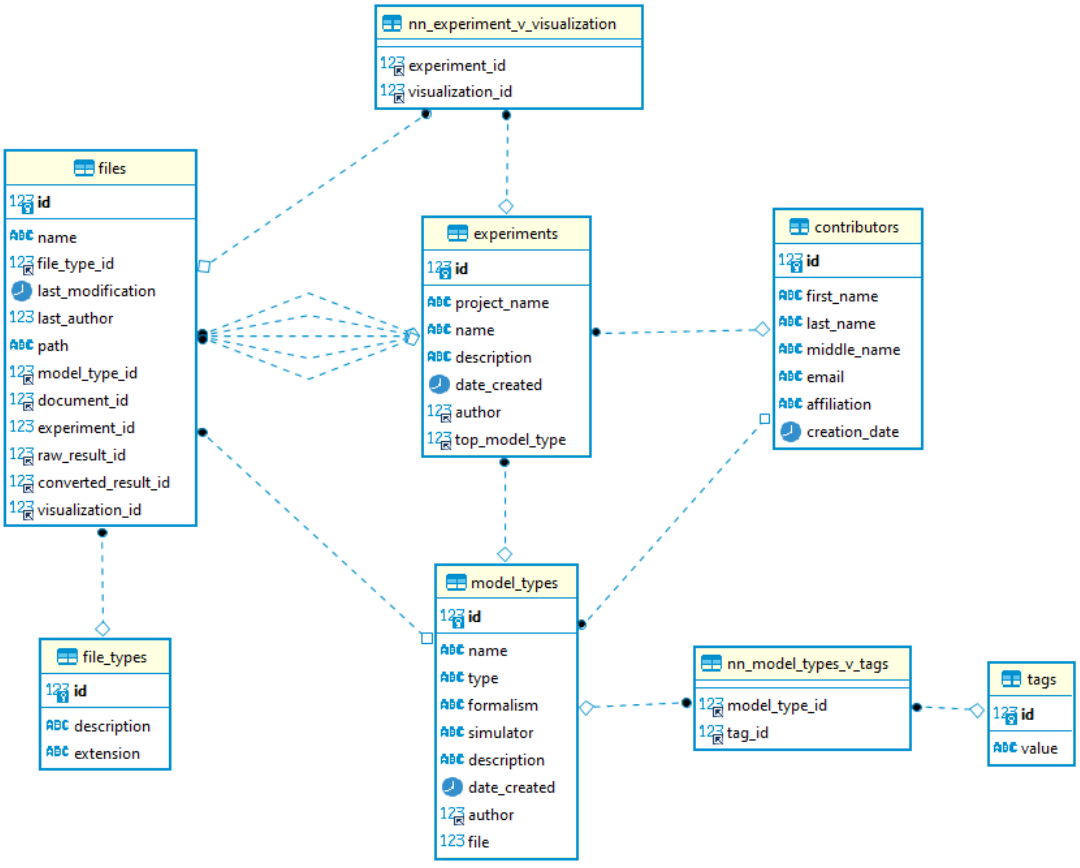

Figure 2: Database model.

As can be seen from figure 2 below, all the files in the conceptual model were grouped in a single table "files" with Boolean columns specifying whether the file is a visualization file, raw_result, converted_result, or model file. These files are all related to an experiment through the mapping defined in the conceptual model. Because an experiment can be associated to many visualization files, we need a separate table "nn_experiment_v_vizualization" to track this N-N relationship. Similarly, all the models were grouped into a single entity "model_types" with a "type" field that determines whether the model is atomic or coupled. Each model can also contain many tags and each tag can be re-used across many models, hence why we need the "nn_model_types_v_tag" relationship table.

3.2 Implementation

The library of models was implemented as a Java virtual machine (JVM) with SpringBoot (2021) backed with Maven in Java. Spring Boot makes it easy to create stand-alone, production-grade Spring based Applications that you can "just run". Maven (2021) is a software project management and comprehension tool.

The implementation of this research work can be found in the following GitHub repository: https://github.com/staubibr/arslab-services-v2

A relational database, namely MYSQL, was used to store the data presented in the conceptual model. This is because the data is relational in nature (an experiment contains many files and contributors, and models can be reused across experiments). Furthermore, MYSQL is ACID-compliant, meaning that it follows the following principles:

- **Atomicity**: All changes to data are performed as if they are a single operation. That is, all the changes are performed, or none of them are.
- **Consistency**: Data is in a consistent state when a transaction starts and when it ends.
- **Isolation**: The intermediate state of a transaction is invisible to other transactions. As a result, transactions that run concurrently appear to be serialized.
- **Durability**: After a transaction successfully completes, changes to data persist and are not undone, even in the event of a system failure. Durability guarantees that the library of models' data will not be lost or corrupted (through a race condition for instance).

3.3 REST APIs available as part of the model

The following REST APIs are available as part of the library of models:

- /api/experiments
    - POST: creates an experiment
    - GET: searches experiments (available query params: name, description, ids, fromDate, toDate)
    - GET /api/experiments/id – retrieves a single experiment uniquely identified by id.
- /api/models
    - POST: create a model.
    - GET: searches models (available query parameters: description, experimentId, contributors, tags)
    - GET /api/models/id – retrieves a single model uniquely identified by id.
- /api/contributors
    - POST: create a user
    - GET: searches users (available query parameters: first name, last name, email, ids, createdFrom, createdTo)
    - GET /api/contributors/id – retrieves a single user uniquely identified by id.
- /api/results

- GET: searches and returns multiple files (available query params: experimentId, modelId, fileType)
- GET /api/results/id – retrieves a single results file uniquely identified by id.

4 HOSPITAL USE CASE

Consider a healthcare system built on two sub-models. The first is an emergency model that involves reading rich geospatial GIS data to determine all types of emergency statistics per area such as the number of people that require critical care. The second sub model represents hospital-based data to determine hospital statistics such as availability, reliability, performance, costs, position, number of treated patients or deaths, and more. These two subsystems can then be linked based on the distance to form a more powerful system able to answer questions such as which area in the city needs more medical coverage or what are the most performant hospitals or potentially which hospitals need to be replaced or relocated?

These insights can help make sure we are making an optimal decision when considering where to build a new hospital for instance and can save governments a lot of money over the long-term. In this case, each one of these two sub models are sub models themselves. The hospital top-model will consist of one model per hospital whereas the emergency area sub model will consist of a number of sub models, each one representing the static data for each sub area. One can now consider implementing a generic model implementation in the programming language of choice that can be used to read any data as long as it conforms to a pre-defined standard. This can provide a scalable way to implement and extend the overall system, resulting in lots of cost reductions, shorter time-to-markets and more robust and intuitive systems.

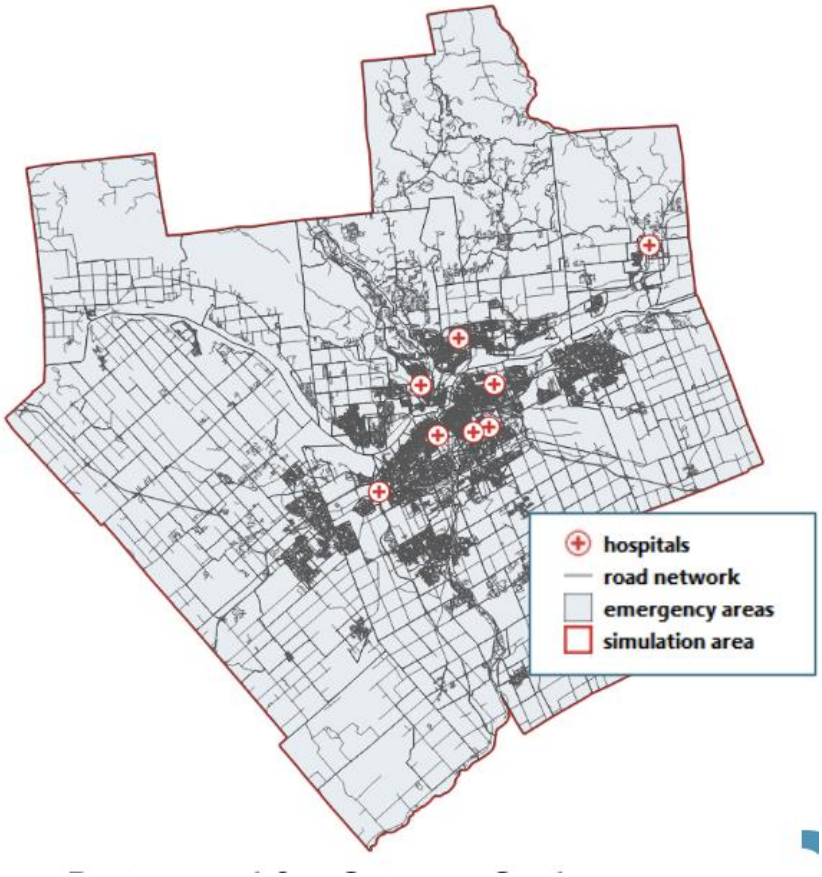

Figure 3: Hospital Use Case Demo.

The library of models' implementation discussed in the previous section can be used to store these hospital sub models and version them so that each sub model can be extended and worked on separately. Consider figure 3 below representing a visualization of the healthcare system described above containing 8 hospitals ("+" signs) and a simulation area divided into a few hundred sub areas.

Each experiment contributor can separately work on gathering the data for a subset of hospitals (e.g. number of treated patients, deaths, costs, availability, etc.) or a subset of areas (e.g. how many people reside in the area, average age, etc.). We then end up with 8 atomic models representing 8 different hospital data gathered by 8 different modelers which can then come together and link their models through the input/output ports to form one coupled model representing all hospital data. Another coupled model representing all geographical data can be obtained by applying the same principle to the simulation areas, where each sub area can be its own atomic models. Combining these two coupled models would form a top-level model capable of showing up the ratio of medical coverage for every area of the city and how critical it is to add new hospitals to a certain area (e.g. if an area is facing lots of medical-related deaths and there are no hospitals nearby, or if a hospital is rarely available and is frequently forced to reject patients in an over-crowded area).

It then becomes clear that the library of models' enable collaborative modelling across a group of modelers and eliminates hazards such as loss of data on a personal computer since data is stored on a secured server that provides data replication across multiple datacenters to guarantee the data never gets lost.

The future vision of this work is for the user to download a set of pairs of emergency area and hospital models, map them to his or her own data, build a workspace where he or she can select which models to keep (i.e. trimming the data) then assembling the atomic models into a coupled model by linking their ports.

A potential bottleneck that can happen when using the library of models is that the database can be very large causing very high API latencies thus a slow rendering on the frontend when displaying the model results to users. These performance bottlenecks can be addressed by:

- Introducing a customizable pagination: API users of frontend applications can then specify how much data they want the server to return, greatly reducing the API latencies since users are now only requesting for the data they need.
- Adding web sockets between clients and the server: Web sockets are bi-directional, full-duplex communications protocol initiated over HTTP between a client and a server. Web socket connections are typically long-lived, meaning the connection will remain open and idle until either the client or the server is ready to send a message. WebSockets thus allow for a higher amount of efficiency compared to REST because they do not require the HTTP request/response overhead for each message sent and received.

5 CONCLUSIONS

In many domains of application, modeling and simulation (M&S) has shown to be useful to study real world systems and to support decision-making through models that abstract the systems under study. This provides decision-makers with a method that requires less resources and involves less risk than studying the system itself. It is a way to evaluate new or upgraded systems without compromising limited resources, interrupting operations, compromising safety, etc. However, building accurate models that adequately represent real-world systems is a difficult and time-consuming task that requires extensive domain knowledge and a deep understanding of the simulation method used. Modeling and simulation and more specifically, DEVS based M&S face significant challenges that limit its approachability. We posit that this is largely due to the lack of an integrated DEVS environment that streamlines and unifies the simulation lifecycle into a coherent and intuitive process for non-expert users. To help with these issues, we introduce a platform that enables collaborative and component-based modelling through a web-based application. This is intended to simplify the simulation and modeling process due to the complex nature of the problem

and the lack of compatibility between models and modelling tools. This helps reduce the time-to-market, high cost and poor-quality insights through a convenient API abstracting all reliability, versioning and shareability issues from modelers. The web-based platform that supports users in the development of domain-specific models, provides a straightforward way to simulate the models and analytical capabilities for comprehensive decision-making. The environment is a means to study ways to overcome these challenges. In the future we will expand this initial effort to a fully featured framework that is approachable by any user and applicable to a wider breadth of scenarios. The framework will be used for applications in simulation-based optimization workflows for urban planning and logistics.

REFERENCES

Al-Zoubi, K., and G. Wainer. 2013. "RISE: A General Simulation Interoperability Middleware Container" *Journal of Parallel and Distributed Computing*, vol. 73, no. 5, pp. 580-594.

Apache Maven Project (2021). "Welcome to Apache Maven". https://maven.apache.org/ Accessed June 1, 2022.

Bernard, Z., P. H. Praehofer, and T. G. Kim. 2000. *Theory of Modeling and Simulation*. Academic Press.

Goldstein, R., S. Breslav, and A. Khan. 2016. "DesignDEVS: Reinforcing Theoretical Principles in a Practical and Lightweight Simulation Environment". In 2016 *Symposium on Theory of Modeling and Simulation (TMS-DEVS)*. Pasadena, CA, USA

Massé, M. 2012. *Rest Api Design Rulebook Designing Consistent RESTful Web Service Interfaces*. O'Reilly.

Sarjoughian, H., and V. Elamvazhuthi. 2009. "CoSMoS: A Visual Environment for Component-Based Modeling, Experimental Design, and Simulation." In *2nd International ICST Conference on Simulation Tools and Techniques*, Rome, Italy.

Spring. 2021. "Spring Boot" https://spring.io/projects/spring-boot Accessed June 1, 2022.

St-Aubin, B., F. Loor, and G. Wainer. 2021. "A Specification for Multi-Simulator, Multi-Formalism Visualization and Analytics on the Web." *Simulation Practice and Theory* (Submitted).

St-Aubin, B., J. Menard, G. Wainer. 2021 "A Web Based Modeling and Environment to Support the DEVS Simulation Lifecycle". *Annual Modeling and Simulation Conference (ANNSIM) 2021*, Fairfax, VA, USA. Society for Modeling & Simulation International (SCS)

Twitter Developer Platform. "Post, Retrieve, and Engage with Tweets" https://developer.twitter.com/en/docs/twitter-api/v1/tweets/post-and-engage/overview. Accessed June 1, 2022.

AUTHOR BIOGRAPHIES

HAMZA QASSOUD is a Master student at the Department of Systems and Computer Engineering at Carleton University. His email address is scs21roeder@gmail.com.

BRUNO ST-AUBIN is a Ph.D. student at the Department of Systems and Compute Engineering at Carleton University in Ottawa. He holds a Master's degree in Geomatics Sciences from Laval University in Quebec.His research interests lie in simulation lifecycle management and simulation visualization, particularly for large scale, spatial scenarios. His email address is staubin.bruno@gmail.com

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.

CRISTINA RUIZ MARTIN is an Instructor II at the Department of Systems and Computer Engineering at Carleton University. Her email address is cristinaruizmartin@sce.carleton.ca