Using Solace Event Broker For Modeling & Simulation

by

Xiangyu Chen

A thesis submitted to the Faculty of Graduate and Postdoctoral Affairs in partial fulfillment of the requirements for the degree of

Master of Applied Science

in

Electrical and Computer Engineering

Carleton University
Ottawa, Ontario

Abstract

Distributed simulation allows a simulation program to run across different computers and locations. The benefit of using distributed simulation is that it solves the limited performance of simulating a large-scale system on one computer and enhances the efficiency of Modeling and Simulation (M&S) systems. In this thesis, we focus on how to apply distributed simulation methods to the entire processes of the M&S ecosystem, and we propose an architecture that uses a message broker to connect to all processes within the M&S life cycle. The architecture combines an Event Broker, a cloud platform for design and visualization of buildings, and a discrete-event simulator. The research shows how a message broker can simplify communication between each M&S software through its Publish/Subscribe technique. The architecture has the potential to bridge M&S systems and make various M&S software platforms work in collaborative fashion.

Acknowledgements

Over the course of my researching and authoring this paper, firstly, I would like to I would like to thank my supervisor, Prof. Gabriel Wainer, for his continuous support and patience throughout the past two years. His guidance helped me during the whole researching and writing period. Working on a very busy schedule, he still give my paper a careful reading and detailed comments. Without his advice and guidance, this study could not have been completed.

Also, I would like to thank the members of the Advanced Real-Time Simulation Laboratory, especially Vinu Subashini Rajus and Mitali Patel for their help in technical matters and general supports.

Finally, my thanks to my family members and friends who love and care me and whom I love and care.

Table of Contents

List of Figures

List of Acronyms

Acronyms	**Full Description**
M&S	Modeling and Simulation
DEVS	Discrete Event System Specification
Cell-DEVS	Cell Discrete Event System Specification
API	Application Programming Interface
RTI	Run-Time Infrastructure
HLA	High-Level Architecture
MSaaS	Modeling and Simulation as a Service
MOM	Message-Oriented Middleware
EDA	Event-Driven Architecture
JMS	Java Message Service
SMF	Solace Message Format
VPN	Virtual Private Networks
BIM	Building Information Modeling
URL	Uniform Resource Locator
URN	Uniform Resource Name
CSV	Comma-Separated Values

Chapter 1: Introduction

In recent times, the field of Modeling and Simulation (M&S) has been experiencing rapid growth and development. This surge in interest is primarily driven by the increasing need to comprehend complex real-world systems and predict their behavior accurately. M&S finds its applications in diverse areas such as engineering[18][19][20], economics[59][60], and social sciences[15][16], making it a crucial tool for decision-making and problem-solving.

The M&S process involves several essential steps that work together to create simulations. These steps include creating models that represent real-world systems, running simulations to understand how these models behave, analyzing the results to draw conclusions, and visualizing the outcomes to make them understandable[1][2]. Each of these steps contributes to the overall accuracy and reliability of the simulation.

However, making all these steps function seamlessly within an M&S system can be challenging. This is because different software components are involved, and coordinating their interactions can be complicated. For instance, software used for modeling might need to communicate with simulation software, and the analysis software might need to exchange data with visualization tools. To build a good M&S system, people have to use different software for different uses. But the developers cannot be good at every software, and only one software is hard to cover all the steps as well.

To address these challenges, the concept of distributed simulation has emerged. Distributed simulation involves breaking down a simulation into smaller parts and running them on different computers that are connected[3][4]. There are different types of distributed simulation: distributed computing systems, distributed information systems, and pervasive systems[4]. By using these architectures, the computers could tackle the complexity of simulations that require a lot of computational power and resources[21][22][35][36]. It also allows simulations to be carried out across different geographical locations[25][26][27].

With the advancement of distributed simulation technology, the role of middleware has become crucial. Middleware acts as a mediator between different software components, facilitating communication and coordination between them[58][8][4][5]. Among the available options, message-oriented middleware [7][8][9][10], often referred to as a message brokers, could provide a better choice. Message brokers provide a Publish/Subscribe architecture that allows software share and access information by publishing or subscribing to a topic (the message address). This architecture has been used in many different fields such as Big Data [37][63][64], Internet of Things [61][62], and others. Many developers employ this technique to access real-time data from databases or sensors for decision-making.

A message broker enables communication between different software components that might operate at different times and places. It acts as a messenger that passes messages between components without requiring them to directly interact. Each step can function independently, meaning that if one part encounters an issue, it won't affect the other parts of the simulation. This enhances the overall robustness and stability of the simulation.

Therefore, each process within the M&S life cycle could keep the independence, meanwhile, they could directly communicate with each other as well. This means that using message broker in an M&S ecosystem could reduce the necessity of mastering all the associated software within the M&S field. For example, developers focused on the simulation process do not need to learn how to use modeling software for building scenario models, and those concentrating on visualization do not have to learn about simulating executions.

Considering the above, the objective of this thesis is to define and implement an architecture for facilitating communication throughout the M&S lifecycle. This architecture aims to enable seamless interaction between simulation software and modeling and visualization tools. To achieve this goal, we introduce a system in which all software components are integrated through a message broker. The architecture's validity is assessed by constructing an M&S project, followed by testing the communication interfaces between these components.

One contribution of this work is the establishment of a bridge that enhances interactions within M&S software. Within this model, we implement the fundamental functionalities of a message broker to ascertain the practicability and advantages of incorporating a message broker across various stages of the M&S lifecycle. Additionally, the message broker provides support for several supplementary features, thereby enhancing the overall interoperability of the system.

In our model, we integrate the Solace event broker [29] into both the Cadmium [51] simulator and the Autodesk Forge platform. The Solace Event Broker is a message broker, as previously discussed, that supports a variety of messaging protocols and patterns. Notably, it offers features such as message persistence, high availability, and different messaging approaches. On the other hand, the Cadmium simulator is a simulation tool for users to build and define Discrete Event System Specification (DEVS) models [43], a type of M&S that allows the system to be modeled as a series of discrete events occurring over time. Finally, Autodesk Forge [41] is a cloud-based development platform created by Autodesk that provides a set of Application Programming Interfaces (APIs) and tools for integrating Autodesk products. It enables developers to integrate 3D modeling, visualization, and collaboration capabilities into their applications and workflows.

By means of the architectural framework we introduce, both Autodesk Forge and the Cadmium simulator are poised to facilitate streamlined access to their respective essential data. In instances where the Cadmium simulator endeavors to retrieve the BIM model from Autodesk Forge, or when Autodesk Forge seeks to acquire the simulation results generated by the Cadmium simulator, the complex steps can be greatly reduced allowing for a direct exchange of messages between the two components.

1.1 Thesis Organization

The remainder of the thesis is structured as follows: In Chapter 2, we provide a concise background on distributed simulation. This includes an exploration of its development, applications, as well as an overview of the message-oriented middleware and the specific message broker employed in this study. Additionally, we offer insights into various modeling and simulation software tools, namely Autodesk BIM 360, Autodesk Forge, and Cadmium simulator, alongside a presentation of the simulation technique employed: Discrete Event System Specification (DEVS), including its extension known as Cell Discrete Event System Specification (Cell-DEVS) formalisms. Moving on to Chapter 3, we introduce and formally define the system architecture proposed in this thesis. Chapter 4 consists of results and case study of the described model in chapter 3. Finally, Chapter 5 draws conclusions, summarizes the key findings, and outlines potential avenues for future research endeavors.

Chapter 2: Background

2.1 Distributed Simulation

Distributed simulation is a concept that allows a simulation program to be divided into smaller programs that run on multiple processors [3][4]. There are several reasons why people want to use distributed simulation. Firstly, it can reduce the time to execute the simulation and allow the execution of large-scale simulation programs that cannot run on a single computer. Secondly, distributed simulation allows different simulators to run in the same environment. Rather than creating a new model which contains all submodels, it can create a new simulation environment to connect different models. In addition, distributed execution can be executed over a broad geographical area via Internet, which saves costs and resources. The simulation is executed on many machines and communicates with each other by the network. Fujimoto [3] discusses two widely used architectures of distributed simulation: client-server and peer-to-peer. For the client-server approach, the simulation runs on one or more server machines, and the user can get the results from the servers. In Peer-to-peer networks, every user could be a client or a server.

Distributed simulations usually need to rely on the support of middleware to run in a distributed computing environment. For instance, the Run-Time Infrastructure (RTI) middleware software [5] can provide real-time functions based on High-Level Architecture (HLA). HLA[11][12] is an architecture initially designed by the US Department of Defense (and later adapted as an IEEE standard) for the reuse and interoperation of the simulations. It divides into three major components: the HLA rules (the standard that simulations must obey), the interface specification (the way that simulations interact with RTI), and the object models (the template for describing simulation elements). Other distributed simulation middleware have been developed; for instance, ARTIS [35], based on the HLA to support heterogeneous and distributed models of a complex-system simulation. *Lasnier et al*. [36] combined Ptolemy, an M&S tool, and HLA for distributed simulation of the cyber-physical system.

In parallel with the development of distributed simulation methodologies, the rapid development of web technology allowed the definition of web-based simulation software, tools, and standards [13][14]. This technology has been used in different fields, including medical applications [15], mathematical models [16][17], engineering [18][19][20], etc. In [15], the author presents several web-based simulation platforms for nursing education, such as The Neighborhood and Second Life. These platforms offered several simulation programs for nursing students to learn CPR, mental health management, intravenous cannulation, etc. *Dormido et al*.[16] designed a web-based tool for control education, students can learn many nonlinear behavior phenomena in control engineering. *Lazaridis et al*.[17] built a web-based learning tool to solve linear problems using animation and visualization techniques. In [18], the authors present a tool for control engineering using Java applets. This tool allows for the analysis and simulation of the single-input-single-output linear control system. *Dominguez et al*.[19] proposed a web-based simulation environment that centralizes the tasks for testing different electrical distribution system configurations and facilitates the detection of defects in a design system.

The advance of distributed system technology and the emergence of cloud computing introduced a new concept: Modeling and Simulation as a Service (MSaaS). Many cloud-simulation tools have been developed following these ideas. In [21], *Calheiros et al*.

developed a CloudSim toolkit for modeling and simulating extensible Clouds. It provides simulation-based approaches for researchers to test their methods and mechanisms for efficiently managing Cloud infrastructures. *Sriram*[22] designed a simulation tool for Elastic Cloud Infrastructures named SPECI. This tool can simulate and predict the performance and behavior of data centers according to different sizes and middleware policies. *Lim et al*. [23] built a flexible and scalable simulation platform for conducting and analyzing multi-tier data center performance and power issues.

Since researchers hope that distributed simulation provides plug-and-play interoperability, web-services-based simulation became a useful method for distributed simulation systems *Ma* and *Tian* [25] proposed a web service-based Multi-Disciplinary Collaborative Simulation System(MDCSS). In MDCSS, various engineering application programs are integrated as a web service federation. This temporary federation provides an alternative to using the software tools in a "pay-as-you-use" method. *Wei She et al*. [26] developed a web service simulation system called WS-Sim. It is based on realistic data from actual web services and applications. In [27], the authors proposed a distributed simulation environment for a System-on-Chip design based on a Simple Object Access Protocol(SOAP). The communication will be only based on HTML and XML protocol. In [28], *Wainer et al*. designed a distributed simulation engine (D-CD++) based on CD++, a modeling and simulation toolkit for DEVS and Cell-DEVS simulation. They use web service technology like SOAP and XML to enable CD++ execution in a distributed environment.

Al-Zoubi and Wainer [6] proposed the RISE (RESTful interoperability simulation environment) middleware based on RESTful Web Services. In RISE, each resource uses the uniform resource identifier (URI) template, and these resources will use uniform channels to exchange information using XML. Compared with the distributed simulation using SOAP-based WS, it has been shown this modeler-oriented middleware greatly improves the simulation interoperability and model reuse.

2.2 Message-oriented Middleware

Besides the different types of middleware for distributed simulation discussed in the previous section, this research is interested in applying Message-Oriented Middleware (MOM) to distributed simulations. MOM software enables many different users to send and receive messages in a distributed architecture. Specifically, it allows many applications to exchange data in the form of a message [7]. There are two types of MOM middleware [8]: Message Passing/Queue and Message Publish/Subscribe. In Passing/Queue architecture, applications send a message by using a client MOM, and a MOM server selects the request in a specific order from the queue. The MOM server acts as a router to the message and does not usually interact with it. In a Publish/subscribe architecture, the MOM is event-driven. We will detail this pattern in section 2.3.

MOM is a family of software or middleware platforms, a technology that can be used to implement an Event-driven architecture (EDA), Service Oriented Architecture, or other architectures. EDA [9][10] has the ability to detect events and react to these events. An event is a change of state that will cause the system to take action. In an event-driven architecture, an event that happens in a system will notify people who are interested in this system, and then people can take action based on this event. For example, when smoke is detected by a sensor, this event triggers the smoke alarm, which will ring and issue a signal to a fire alarm control panel.

2.3 Solace Event Broker

The Solace PubSub+ platform [29] provides a variety of open standard protocols and APIs for developers to create their applications. In particular, the Solace event broker [30] is a middleware appliance, software, or SaaS, based on the event-driven architecture. It is used to transmit information between event senders and receivers, and it facilitates and mediates the interactions of different applications and platforms. Solace event broker is available for installation into various environments such as clouds, PaaS, iPaaS, and on-premises, and it supports many programming languages such as C/C++, C#, Java, JavaScript, etc. [31]. Solace event broker allows information or events to be transmitted from one application to another application or from one platform to another platform. For example, the producer can publish the messages through MQTT, and the others can subscribe to the messages through SpringBoot.

The event broker is built using some basic design concepts discussed next.

1. Messaging: the term *messaging* refers to the technology that lets computer systems share information without requiring either direct connections or awareness of each other's location. The architecture of Solace messaging contains five components, as shown in Fig. 1 [32].

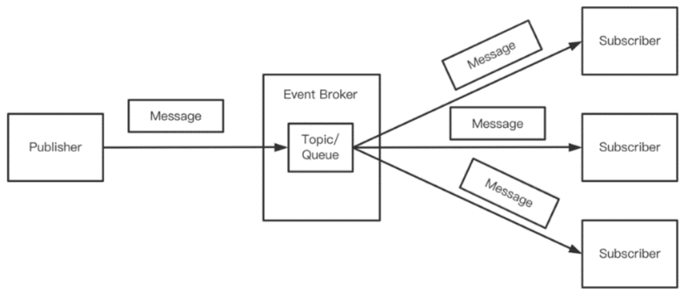


Figure 1. The Architecture of Solace Messaging

Publisher: The application or entity that sends a message to a topic;

Subscriber: The application or entity that receives the message from a topic;

Message: The data exchanged between applications. The Messages often contain events, queries, commands, or other information;

Topic: A type of address that is used to exchange messages. It is used when more than one subscriber is intended to consume the message;

Queue: A type of address that is used to exchange messages. It is used when the message is consumed by at most one subscriber.

2. Exchange Patterns: there are three message exchange patterns available in the event broker. a) Point-to-Point, b) Publish-Subscribe, and c) Request-Reply [33]. Point-to-Point messages sent by the Producer are processed by a single Consumer. With Publish-Subscribe messaging, messages sent by the Producer are processed multiple times by different consumers. Each consumer receives their own copy of the message for processing. For the Request-Reply pattern, applications achieve two-way communication using separate point-to-point channels: one for requests and another for replies.

3. Delivery Mode: the event broker supports Directed and Guaranteed message delivery modes [34]. Direct messaging is meant for use with high-speed applications that can tolerate occasional message loss. Producers can publish messages to a topic, and when these messages are received by the event broker, they are delivered to Consumers with matching topic subscriptions. For the Guaranteed mode, clients can publish messages to an event broker. When the messages are received by the event broker, they are saved in the event broker's message spool. Then, the acknowledgment is confirmation to the publisher that the event broker accepted the message. Those messages are persisted on the event broker until they expire, or the consuming client acknowledges the messages. Guaranteed Messaging can be used to ensure the delivery of a message between two applications, even in cases where the receiving application is offline, or there is a failure of a piece of network equipment. It keeps a copy of the message until successful delivery to all clients and downstream event brokers has been verified.

Besides the Solace event broker, there are many other MOM tools, such as IBM MQ[54], Apache Kafka[37][38], Rabbit MQ[55][56], and Active MQ[39][40]. We will simply introduce you to Apache Kafka and ActiveMQ. Apache Kafka is a distributed event stream platform developed by LinkedIn [37]. It uses a public-subscribe messaging architecture. Kafka is built from three main components: Kafka Cluster, Kafka Connect, and Kafka Stream. Kafka Cluster consists of one or more Kafka brokers organized by topics. The Brokers store data that are sent by producers. Kafka Connect is used to transmit data from Kafka and external applications. Kafka Stream is for data stream processing [38]. ActiveMQ [39][40] is an open-source message broker built on top of Java. It implements

Java Message Service (JMS), a Java API that allows applications to send and receive data asynchronously. ActiveMQ currently has two versions: Classic and Artemis. ActiveMQ must be installed in the Java Runtime Environment.

2.3.1 Architecture and Workflow of Solace Event Broker for Interaction

The Solace event brokers provide a foundation for messaging across multiple protocols and standards, including Solace Message Format (SMF), JMS1.1, MQTT3.11, REST, and AMQP1.0. As events move through the broker, they are translated from an ingress messaging protocol to an egress messaging protocol for each consumer that receives messages. This section will discuss how the Solace SMF protocol makes use of the event broker's messaging components to move data from the producers to the broker and from the broker to the consumers only. The figure 2 shows how applications interact with a Solace event broker.

Step 1: An application initiates the Solace API and the context that acts as a container in which sessions are created and session-related events can be handled. A session is the communication channel between the API and the broker.

Step 2: The app creates a session and configures the properties of this session. In this process, the username, password, IP address, and Message Virtual Private Networks (VPNs) will be set. Message VPNs are managed objects on Solace event brokers that enable the segregation of the topic space and the clients. They also group clients connecting to a network of event brokers, so that messages published within a particular group are only visible to that group's clients. Clients in different Message VPNs are allowed to subscribe to the same topics, and two clients in different Message VPNs are allowed to publish messages to topics that correspond to those clients' subscriptions. Based on the membership of Message VPN, only a client connected to the same Message VPN as a particular subscriber can receive messages from that publisher.

Step 3: The Solace Session connection opens up a flow-controlled and bi-directional TCP connection from the application to the broker. All communications for producers and consumers to send and receive messages for all Message Exchange Patterns are based on this bi-directional connection.

Step 4: The broker authenticates the connection. The basic authentication scheme is the default client-authentication scheme for a Message VPN. For client applications, it's available using any of the Solace messaging APIs. In a basic authentication scheme, a client can authenticate with an event broker by providing a valid client username and password as its credentials.

Step 5: The broker authorizes the connection. Once a session is connected to the required services and authenticated, we proceed to the authorization stage. Based on the client username in the connection credentials, two profiles are applied: the Client Profile and the ACL Profile. The Client Profile sets the broker resources that are allowed to access by clients. And the ACL Profile defines which data events the client can produce and consume.

Step 6: The application's software development kit(SDK) creates a publisher flow. In this process, SDK sends "last messageID sent" and "last messageID ACK received", then, the broker sends "last messageID received" and "last messageID AQCK sent". This state is maintained by the application as a message flow to allow for correct re-transmissions as required on reconnects

Step 7: When a message is published, Application SDK sends the message and decrements the available window by one. After receiving, routing, and persisting this message, the broker sends ACK. The application SDK then receives ACK and increments the available window by one.

Step 8(Optional): This step deals with the establishment of application subscriptions. Application Subscribes by adding a subscription to the queue. In this step, the application sends a SUBSCRIBE message with a topic and an endpoint. And API forwards that message. Based on the subscription endpoint, the broker adds that subscription to the connection or the backing queue and then sends SUBACK.

Step 9: The application SDK creates a consumer flow and connects to the flow. The application API handshakes with this flow and binds it to a Queue/Endpoint. Then, the Message is able to move from broker to subscriber.

Step 10: The Application starts to flow. When the broker sends a message, it decrements the available window by one. The client API then receives the message and passes it to the application.

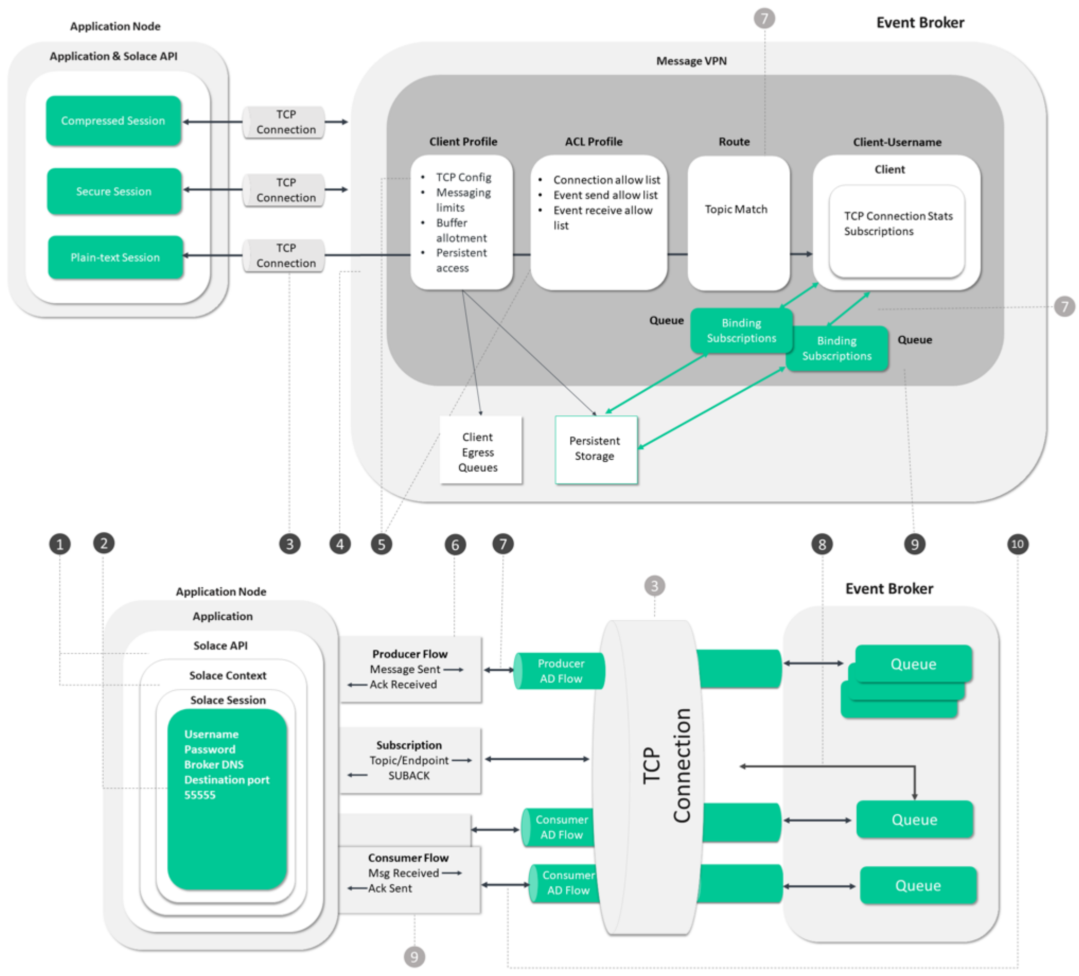


Figure 2. The Architecture of Solace event broker for interaction[57]

2.4 Modeling and Simulation Software

In the rest of the research, we have used different simulation platforms that are described in this section.

2.4.1 BIM 360

Autodesk BIM 360 [52] provides tools for modeling and visualization. It is a cloud-based platform that allows people to create BIM (Building Information Modeling) projects. It not only allows design and construction teams to work more efficiently, but it allows them to capture the data they create during the process to benefit operations and maintenance activities. By using BIM360, people can design and manage all projects in a single platform, and they can easily access their projects anywhere.

2.4.2 Forge Platform

Autodesk Forge [41][42] is a platform of web service APIs that allow you to integrate Autodesk SaaS products (such as BIM 360, PowerBI, etc.) into your workflows, and it helps people to access and use their design and engineering data through the cloud.

Forge is a cloud platform that allows users to build custom applications to access Autodesk SaaS products. It mainly contains various foundational web services APIs, namely: Authentication, Data Management, Data visualization, Model Derivative, Design Automation, Viewer, BIM 360, and Reality Capture. These services will be described in detail next.

The Authentication API, also called OAuth (Open Authorization), is for token-based authentication and authorization. The API generates tokens based on the OAuth 2.0 standard for authenticating requests made to the Forge APIs and SDKs. OAuth 2.0 that is a method for granting access to a set of resources that uses access tokens (a token is a piece of data that represents the authorization to access resources on behalf of the end user). The basic workflow is as follows: a client (application) makes an HTTP call to its Forge authorization server and provides its credentials. A token is then returned to the client. Then, the client can make subsequent calls to various APIs on the platform. At call time, the platform keeps track of what resources are allowed or denied access by the token's permitted range. Once the token expires, the client will have to retrieve a new token by going through all of these steps again. After the users obtain the token, they are allowed to access the services that are provided by the Forge platform.

The Data Management API provides a consistent way to access data across BIM 360 Team, Fusion Team, Object Storage Service (OSS), and others. BIM 360 Team and Fusion Team are cloud-based collaboration tools that enable users to work efficiently together in a central workspace. Users are able to view, share, and review 2D and 3D file formats from any device such as browsers and mobile devices. BIM 360 Team is used for sharing BIM models while Fusion Team is for 3D engineering products like electronics and Printed Circuit Board (PCB) design. OSS is a service within the Data Management API that is responsible for managing and storing files from user's Forge application (this service we will be explained in detail later). Using this API, a number of workflows can be built, including workflows for accessing a BIM model in BIM 360 Team and obtaining an ordered structure of elements, IDs, and properties to generate a list of materials in a 3rd party process. The Data Management API is composed of the following services: (1) Project Service: Navigate to a project that is from a hub. A hub is a collection of one or more users with their related projects. The project represents the entry point to all the related information and data for BIM 360 projects and Fusion projects; (2) Data Service: Navigate and manage the 2D and 3D designs in terms of folders, items, and versions, as well as the relationships between these entities. An item could be one or more versions of files. Each item can have multiple versions; (3) Schema Service: This allows your application to understand the structure and semantics of extended datatypes; (4) Object Storage Service: This allows your application to download and upload raw files (such as

PDF, XLS, DWG, or RVT) that are managed by the Data Service. These files are independent of any Autodesk SaaS application and come from your app on the Forge platform. As shown in Figure 3, block A and block B represent the hub and the project, respectively. C is the folder of a project. A project could have one or more folders. D is an item in the House folder and E is a version of this item which is a file that could be downloaded using the endpoints exposed by the OSS.

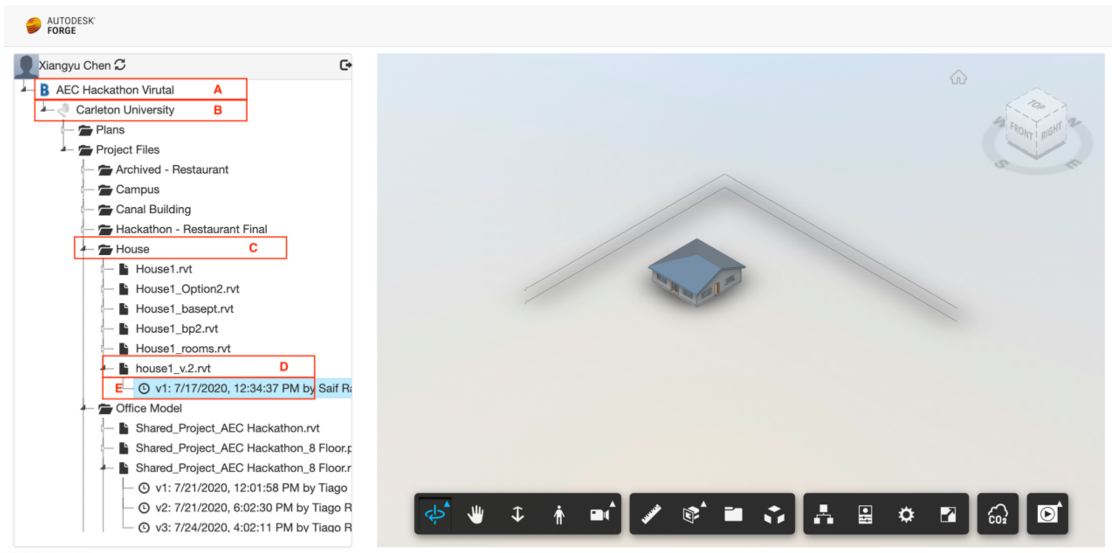

AUTODESK FORGE
Xiangyu Chen
AEC Hackathon Virutal A
Carleton University B
Plans
Project Files
Archived - Restaurant
Campus
Canal Building
Hackathon - Restaurant Final
House C
House1.rvt
House1_Option2.rvt
House1_basept.rvt
House1_bp2.rvt
House1_rooms.rvt
house1_v.2.rvt D
E v1: 7/17/2020, 12:34:37 PM by Saif R
Office Model
Shared_Project_AEC Hackathon.rvt
Shared_Project_AEC Hackathon_8 Floor.
Shared_Project_AEC Hackathon_8 Floor.
v1: 7/21/2020, 12:01:58 PM by Tiago
v2: 7/21/2020, 6:02:30 PM by Tiago R
v3: 7/24/2020, 4:02:11 PM by Tiago R
TOP
FRONT
RIGHT

Figure 3. The hierarchy of a BIM360 project

The BIM 360 API allows users to access, store, and download Revit models on the Forge platform in such a way that users do not need to directly access the BIM data. The use of the BIM 360 API allows us to integrate and expand our applications with the services of Autodesk BIM 360 . This allows us to access the BIM projects that the modelers create in the Autodesk BIM 360 platform. This API consists of many services such as BIM 360 Account Admin, BIM 360 Asset, BIM 360 Checklists, BIM 360 Cost Management API, and so on and so forth. In our article, we will explain these three APIs that are used in our project: BIM 360 Account Admin, BIM 360 Data Connector, and BIM 360 Document

Management API. The BIM 360 Account Admin API enables you to manage your Autodesk BIM 360 account and its projects and members. It could create, update, and retrieve information about BIM 360 projects and project users. The BIM 360 Data Connector API exports data from the BIM 360 service modules in order for users to create their own custom analytics and reports. Users are able to request data (such as Admin information, issues, locations, cost, and so on.) from one or more of the BIM 360 services. The BIM 360 Document Management API could access, upload, and share 2D plans, 3D BIM models, and any other project documents. Most of the BIM 360 endpoints relating to the storage and management of BIM 360 documents are housed within the Forge Data Management API. The BIM 360 Document Management API is a part of the Data Management API and for this reason, we are able to manage and access our BIM 360 project through Data Management API. And users are able to use the Forge Viewer API to visualize the BIM model in their applications.

The Viewer is a client-side JavaScript library based on WebGL for rendering 3D and 2D models. It is strongly dependent on Three.js API. The Viewer allows design patterns to be viewed and shared on the web from a wide range of products including AutoCAD, BIM 360, Revit, and many others. And developers can customize the Viewer's appearance and behavior through the Extension API within Viewer. The user would need to initialize the Viewer and create a Viewer instance before loading a model onto the Viewer. Then the Viewer communicates with the Model Derivative API who translates the model into the SVF/SVF2 format so that the model can be displayed on the Viewer.

The Model Derivative API allows users to represent and share their designs in different formats and to extract valuable metadata. This API mainly provides file translation and data extraction services. In the case of file translation, the Model Derivative could translate designs into the SVF, STL, OBL, and so forth formats and generate thumbnails of different sizes from design files. In the case of the data extraction service, this API could extract design metadata (such as object properties and hierarchy) and build it into our workflows. It could extract the parts of the model and export their geometric representations as well. One of the most common uses of the Model Derivative API is to translate designs into SVF or SVF2 formats in order to be able to display them on the Viewer that we mentioned above.

2.4.3 DEVS and Cadmium Simulator

We will combine different platforms with models defined using the DEVS formalism, which was created for the modeling and simulation of discrete-event dynamic systems. An introduction to the DEVS simulation is presented in [43]. First of all, the DEVS formalism can describe a system as a composition of atomic and coupled components. The atomic model consists of the set of input events (X), the set of output events (Y), the set of sequential states (S), the external state transition function (δext), the internal state transition function (δint), output function (λ) and time advance function (ta). The external state transition function could combine the current state and input events to generate a new state. The internal state transition function changes the state after a period of time. When time expires, the model outputs the value by the output function, and then it changes to a new state by internal function. Therefore, an atomic model can be specified as M=< X, Y, S, δint, δext, λ, ta>. The coupled model can be defined as CM=< X, Y, D, Md, EIC, EOC, IC, select>. In this function, D is the set of component names. Md is a basic DEVS model.

EIC is the set of external input couplings. EOC is the set of external output couplings. IC is the set of internal couplings, and select is the tiebreaker function. Because DEVS formalism is appropriate for event-driven models[44], DEVS could be used in many simulation fields. In article [45], a simulation to evaluate how the cloud and the fog can work together to enhance the user experience by using DEVS to model a cloud-only and a cloud-fog scenario. In [46], DEVS was used to build an agricultural model based on an equipment model and a field model based on DEVS to model a variety of agricultural fields with different shapes and different equipment. Similarly, in [47], the authors presented an electrical power system for energy planning based on DEVS which has a Storage module, Generator module, Load module, Dispatch module, and Transmission module.

In addition, there are several extensions of the DEVS model, like Cell-DEVS [43] and Parallel DEVS (PDEVS) [48], etc. The PDEVS formalism proposed a parameter called confluent transaction function to solve the collision behavior and reflect the co-occurrence of events in the system, which cannot be modeled by DEVS formalism [48].

A Cell-DEVS model is a defined as an n-dimensional cell space. Each cell is an atomic model. hence, the atomic model can be defined as TDC=< X, Y, S, N, delay, δint, δext, t, λ, D >. X is the set of input events. Y is the set of output events. S is the state set. N is the set of input values. Delay is the type of delay. δint is the internal transition function. δext is the external transition function. λ is the output function. t is the local computing function. D is a state's duration function. When the external event occurs, the local computing function is executed, and then the results will be transmitted when the state changes in a delay. There are two types of delay which are transport delay and inertial delay. For transport delay, state changes must be informed in the futures, their values and scheduled times are stored in a local queue. Whereas, the inertial delay prevents any scheduled change from taking place upon receiving an external event from a neighbor cell before the scheduled time. In addition, the coupled model of Cell-DEVS is GCC=< Xlist, Ylist, I, X, Y, η, {t1,...,tn}, N, C, B, Z, select>. X is the set of input events. Y is the set of output events. Xlist is the list of input couplings. Ylist is the list of output couplings. η is the neighborhood size. {t1,...,tn} is the size of the cell space. N is the neighborhood set. I is the set of state. C is the cell space set where are Cell-DEVS atomic components. B is the border set. Z is translation function. select is the tie-breaking selector function.

The Cadmium simulator is a C++17 header-only simulator implemented as a library. It allows users to define and execute DEVS models [51]. The architecture of Cadmium has four levels: Simulation, Abstract modeling, Bridge, and Concrete modeling. The simulation level is where all the modules in charge of running the simulation of the implemented model are defined [51]. There are three types of classes at this level: Simulator classes, Coordinator classes, and Runner classes. The simulator class manages the abstract algorithm for atomic models. The coordinator class manages the abstract algorithm for coupled models. It’s a top-down implementation where Runner sends an advance request to all Coordinators. Then, each Coordinator receiving one of these messages will send requests to all its Simulators. The simulator runs the simulation of its model sequentially and sends results to its parent Coordinator until all the results are collected at the Root level. Finally, the root coordinator routes the message to Runner. The Abstract modeling level defines the modeling class interfaces used by the simulation level. The Bridge level includes the Atomic wrapper and link implementation modules for connecting the model and abstract model class. Finally, the concrete modeling level is for defining all the concrete models.

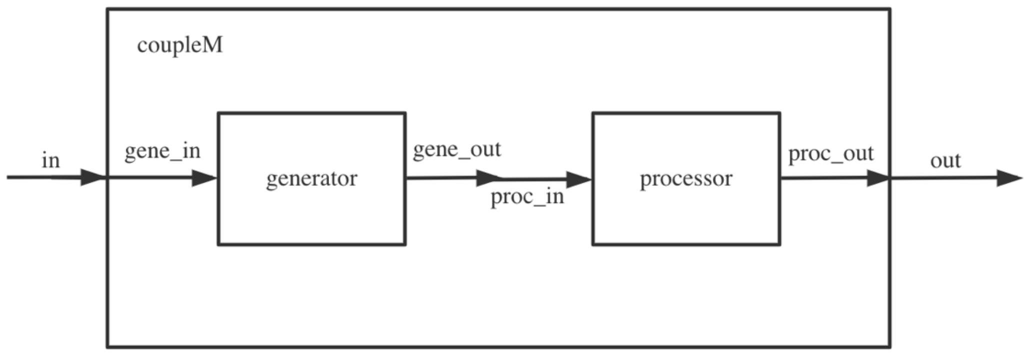


Figure 4. An example of a DEVS Coupled model

Figure 4 shows a sketch of a DEVS coupled model. In this model, the generator will send a "gene_out" message to the processor when the generator receives specific instructions. The processor then will process the "gene_out" message and generate the output messages. The "coupleM" model (coupled model) can be defined as follows:

X = {in}

Y = {out}

D = {generator, processor}

EIC = {(in, gene_in) }

EOC = {(proc_out, out)}

IC = {(gene_out, proc_In)}

As mentioned above, we can define the generator model (atomic model) as follows:

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

X = {gene_in}

Y = {gene_out}

S = { Active, Passive }

δint(Active) = {Passive; }

δext(gene_in, Passive) = {Active;}

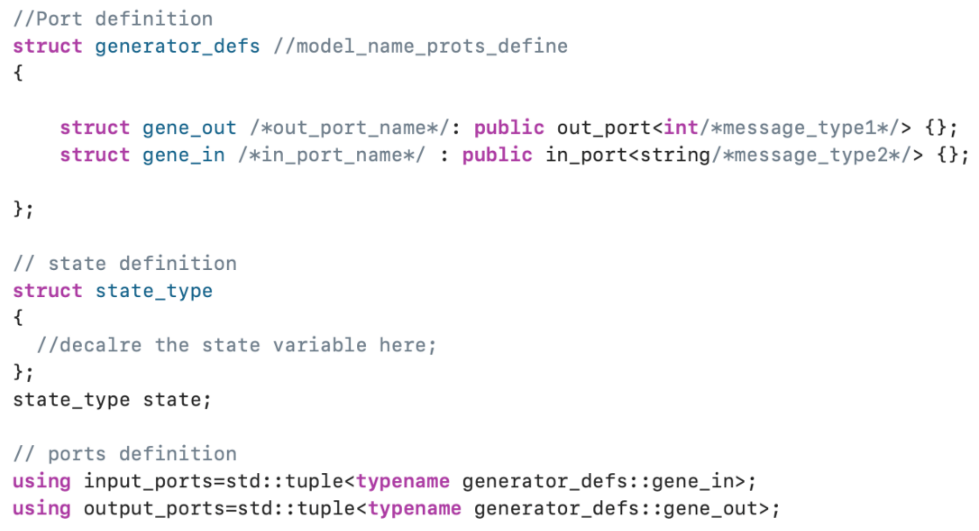
λ(Active) = {gene_out; }

ta(Active) ={0}

ta(Passive) ={inf;}

The format of the processor model's definition is the same as the generator model.

Based on these DEVS formalism structures, we can define them in Cadmium. The atomic model is defined in an hpp file. Figure 5 shows the templates for defining the ports and the states. Cadmium defines ports as a structure that contains all the input and output ports of this model. It's called *model_name_ports_define*. Since the ports were declared in structure, to access the input and output ports, we need to use *model_name_ports_defs::in_port_name* and *model_name_ports_defs::out_port_name*. The state variables of the model are declared in a structure called *state_type*. All the state variables of the model must be declared inside the structure. Figure 6 shows the templates for the internal transition function, external transition function, output function, and time advance function. First of all, the internal transition function is defined as a void method called *internal_transition()*, which takes no parameters. Then, the external transition function is defined as a void method called *external_transition*. The method takes two parameters, the elapsed time (e) and a bag of input messages (mbs). There is one bag of messages per input port. *make_message_bags<>* is a template data type declared in the Cadmium library used to declare a bag of messages for input or output ports. Next, the output function called *output()* is defined as a constant method that returns a bag of messages in the output ports. And the time advance function is defined as a constant method that returns the time of the next internal transition and takes no parameters. It is called *time_advance()*.



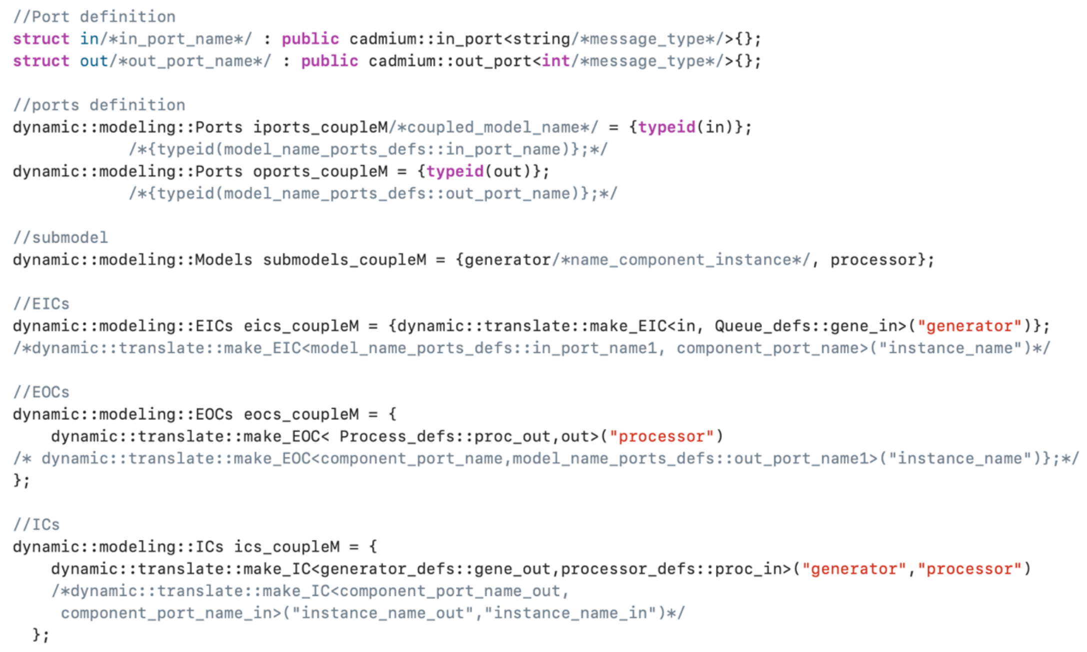
```
//Port definition
struct generator_defs //model_name_prots_define
{

    struct gene_out /*out_port_name*/: public out_port<int/*message_type1*/> {};
    struct gene_in /*in_port_name*/ : public in_port<string/*message_type2*/> {};

};

// state definition
struct state_type
{
  //decalre the state variable here;
};
state_type state;

// ports definition
using input_ports=std::tuple<typename generator_defs::gene_in>;
using output_ports=std::tuple<typename generator_defs::gene_out>;
```

Figure 5. The templates for defining ports and states

```
// internal transition
void internal_transition()
{
     //define internal transition function here;
}

// external transition
void external_transition(TIME e, typename make_message_bags<input_ports>::type mbs)
{
    //define external transition function here;
}


// output function
typename make_message_bags<output_ports>::type output() const
{
    typename make_message_bags<output_ports>::type bags;
   //Define output function here
    return bags;
}

// time_advance function
TIME time_advance() const {
    TIME next_internal;
    //Define time advance function here;
    return next_internal;
}
```

Figure 6. Templates for transition functions

Coupled models are defined inside the main function in a cpp file. Figure 7 shows how to define the “coupleM” model. In coupled models, we can omit grouping all the ports in a single structure and declare them as shown. To assign the input and output ports that are already declared to a coupled model, Cadmium uses the data type *Ports*. *Ports* is a data type used to define input and output ports. For the definition of submodels, they are stored inside a variable of type *Models*. *Models* is used to define the components of a coupled model, which is defined in the cadmium library. It is a vector that takes as elements pointers to models. The *name_ component _instance* is the name given to the variable that stores the instance of the component of the coupled model or atomic model. Then, Cadmium provides a similar method to define EICs, EOCs, and ICs. The sets are stored as a vector with elements of type *EIC*, *EOC,* and *IC,* respectively. The function *make_EIC<>()* is used to create an EIC structure, and it returns an element of type EIC. Function *make_EOC<>()* is similar to *make_EIC<>()*, but for the External Output Couplings. And *make_IC<>()* is used to create the internal couplings. they all take template parameters in a specific order(from-to)



```
//Port definition
struct in/*in_port_name*/ : public cadmium::in_port<string/*message_type*/>{};
struct out/*out_port_name*/ : public cadmium::out_port<int/*message_type*/>{};

//ports definition
dynamic::modeling::Ports iports_coupleM/*coupled_model_name*/ = {typeid(in)};
            /*{typeid(model_name_ports_defs::in_port_name)};*/
dynamic::modeling::Ports oports_coupleM = {typeid(out)};
            /*{typeid(model_name_ports_defs::out_port_name)};*/

//submodel
dynamic::modeling::Models submodels_coupleM = {generator/*name_component_instance*/, processor};

//EICs
dynamic::modeling::EICs eics_coupleM = {dynamic::translate::make_EIC<in, Queue_defs::gene_in>("generator")};
/*dynamic::translate::make_EIC<model_name_ports_defs::in_port_name1, component_port_name>("instance_name")*/

//EOCs
dynamic::modeling::EOCs eocs_coupleM = {
    dynamic::translate::make_EOC< Process_defs::proc_out,out>("processor")
/* dynamic::translate::make_EOC<component_port_name,model_name_ports_defs::out_port_name1>("instance_name")};*/
};

//ICs
dynamic::modeling::ICs ics_coupleM = {
    dynamic::translate::make_IC<generator_defs::gene_out,processor_defs::proc_in>("generator","processor")
    /*dynamic::translate::make_IC<component_port_name_out,
     component_port_name_in>("instance_name_out","instance_name_in")*/
  };
```

Figure 7. the template of the coupled models

Chapter 3: System Architecture

3.1 Overall Architecture

As discussed in the previous chapter, we can use a message broker as a method for distributed simulation. As discussed in Chapter 2, some applications for message brokers focus on the communications and transactions between simulation models. In those cases, different models run on different servers and the message broker is used to send and receive messages. Specifically, these applications split a large complex simulation into one or more federations that collaborate. The federations could run on different servers, on the web, desktop, or cloud computing devices. The communication between each federation is conducted through the message broker. For example, *Al-Zoubi* and *Wainer* [6] built a DCD++ system that could execute Parallel DEVS simulation on the web via RISE, which is a Message-Oriented Middleware. The message broker can be used not only in the communication of distributed models, but also throughout the entire process of the M&S lifecycle (problem definition, conceptual modeling, specification, implementation, and experimentation).

In an M&S ecosystem, the modelling tools are used for the transformation of the conceptual model into a simulation model based on a certain formalism such as DEVS, Petri Nets, Finite State Automata, etc. In order to study the behavior and performance of a real or theoretical system, simulation tools can assume many forms and experimental environments. And visualization tools are used for presenting the simulation data in the form of geographical maps, graphs, heat maps, statistical charts, etc.

During these processes, each piece of software is independent, which means that interaction between different tools could be a problem. Developers of an M&S project are required to install many different pieces of software and spend time learning about the usages of the software. To extend software capabilities and simplify development costs, we introduced the Solace broker as a core component of distributed simulation in M&S projects.

The use of the Solace means that developers do not need to install any other software. Users could obtain appropriate data such as model configuration files, statistical analysis results via the broker.

The modeling, simulation, visualization, and analysis software could connect with each other through the message broker as shown in Figure 8.

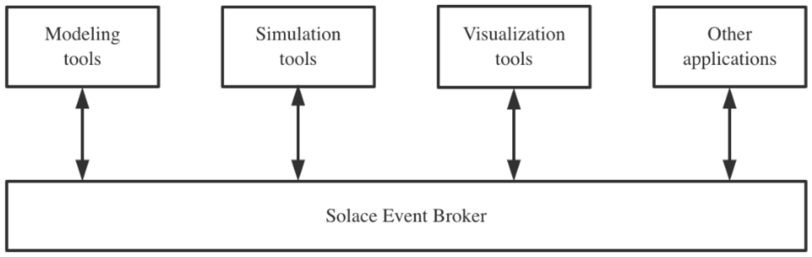


Figure 8. System architecture

As seen in the figure, the software with different functions interact with each other through the broker. In our case, the modeling software could send a BIM or GIS model to the simulation engine. The simulation software would send simulation results to the visualization or analysis software as well. The modeling software can publish scenario models related to the topic they create, and then any application subscribing to the corresponding topic can receive the model and use it for simulation or other uses. A simulation tool that runs a program can also publish the results to another topic so that other applications can use these results for visualization and analysis, etc.

In the next chapters we introduce a prototype implementation, in which we used Cell-DEVS as the formalism of the simulation model, the Cadmium simulator as the tool for Cell-DEVS simulation, and Autodesk Forge as the tool for modeling and visualization, nevertheless the chosen formalisms and tools can differ. Figure 9 shows the three major components of the system architecture that were defined and implemented during this research, namely, the Forge platform, the Solace Event Broker, and the user's desktop.

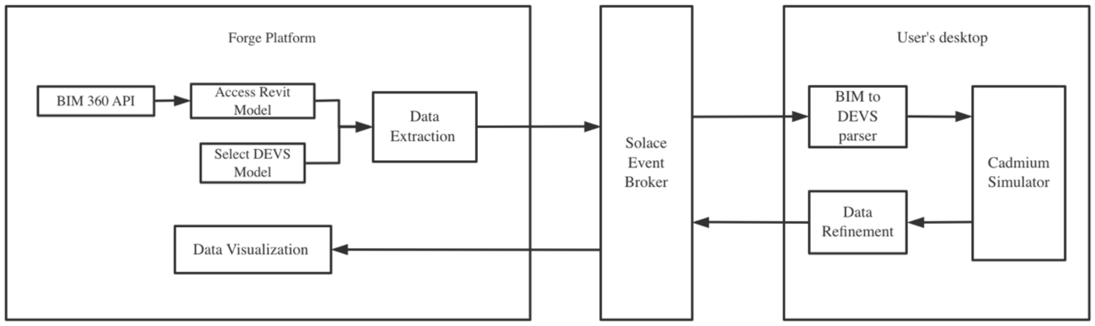


Figure 9. The Proposed Architecture

As noted above, Forge is a web service API platform that allows users to integrate Autodesk SaaS products. It is a web-based application and runs on the Forge Server. Since it allows people to access BIM 360 services, it is used as a model tool. And because Forge's Viewer service is a WebGL for rendering 2D and 3D models, we use it as a visualization tool in our project as well.

Cadmium is a C++17 header-only simulator implemented as a library. It allows users to define and execute DEVS and Cell-DEVS models. In our project it works on the local desktop. When we use Cadmium for Cell-DEVS simulation, we need a Cell-DEVS configuration file. Therefore, after obtaining the BIM model file from Forge, the BIM-to-DEVS parser converts the BIM model file into the appropriate format for Cadmium. In addition, for the Forge application to be capable of reading and displaying simulation results from Cadmium, the data refinement program is required to convert the simulation results to a Forge-readable format.

Using this method, one or more clients can receive the data when the publisher decides to send the message. Next, we will detail the structure of each of the components.

3.2 The Forge Component

The Communication between Solace and Autodesk Forge is based on the Forge BIM-to-DEVS architecture presented [52], which uses an architecture that allows interaction between a BIM model and a DEVS model using the Forge platform. Based on this original idea, we introduced new methods used to connect Forge publishes the model to the Solace broker, introduced in Figure 10.

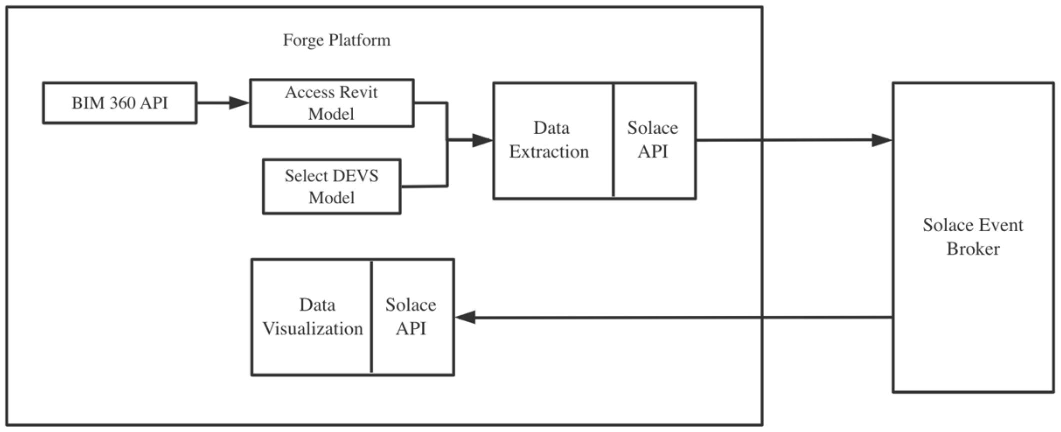


Figure 10. Software architecture using the Forge platform

Our architecture uses some of the foundational services of the Forge cloud platform (which allows users to build custom applications using various web services), namely Authentication, Data Management, Model Derivative, Viewer, and BIM 360.

To publish a model to the Solace broker, our first step is accessing the Revit models on the Forge platform. This stage occurs on the server side of the platform. To do so, we must use authentication, BIM 360 and Data Management services. The Authentication service is used for token-based authentication and authorization, and it enables users to call Forge APIs and use its services. The BIM 360 service enables users to integrate BIM 360 services onto the Forge platform. And the Data Management service can access the data from BIM 360, Fusion team and OSS and display them on Forge. We should first obtain permission to access BIM 360 data. Once we login to our Forge platform through the Authentication API, it will redirect the user to an Autodesk login page where we can authorize our forge platform to access the BIM 360 data. The authentication API will then return an authorization code through a callback URL (Uniform Resource Locator). With this authorization code, we can send a request to Forge server for an access token. Once the token has been obtained, we are then allowed to access the BIM 360 data (Revit Model) via the Forge platform.

Next, we use the Data Management API to navigate and access which project and model we want to use. In order for our model to be presented in the browser, we need the Viewer and Model Derivative APIs. As discussed in Chapter 2, the Viewer service is a JavaScript library that allows us to view and share BIM models on the web. This API is the client side of the Forge application. As mentioned earlier, we would need to translate the model into SVF or SVF2 format if we wish to present the model through the Viewer. Therefore, the

Viewer API will communicate locally with the Model Derivative API to translate and retrieve data in the model.

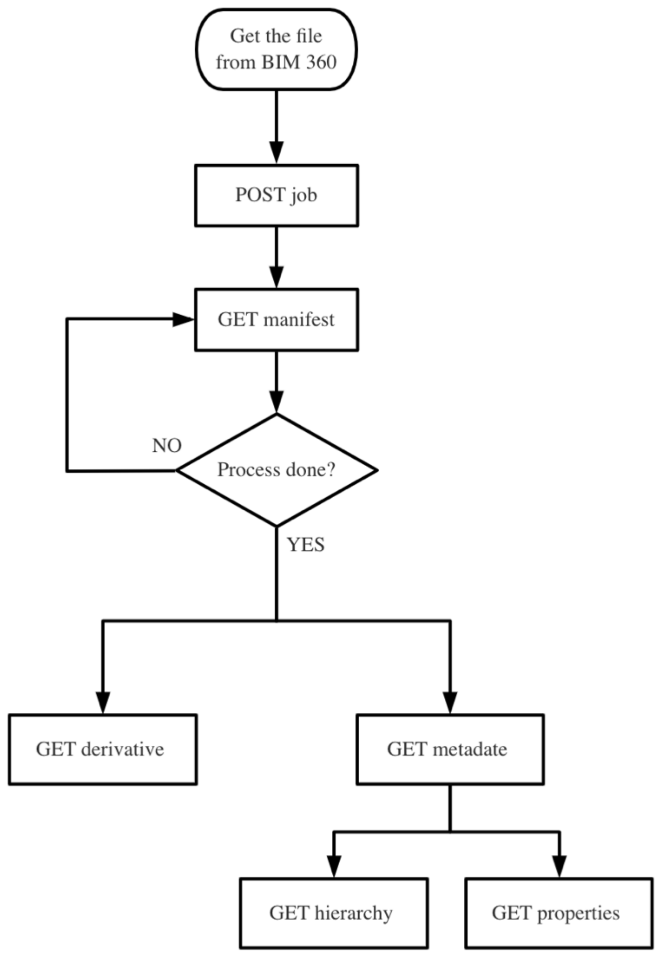


Figure 11. The process of fetching data

Figure 11 illustrates how to use the Model Derivative API for translating and retrieving data. After we obtain the BIM 360 file through the Data Management service, a request for translation will be sent to the Forge server via "POST job" endpoint of the Model Derivative API. Then, the user could get a manifest file. The manifest file is a JSON document that contains all the information about all the model's translations that you've requested so far. Once the translation process completes, we can locate information about the SVF-format translation such as the Uniform Resource Name (URN) of the model by "GET derivative" endpoint. Then, the model could be displayed on the Forge platform by embedding the string value of the URN of the model into an instance of the Viewer object.

When the user has decided the model that they want to publish in a model database, the process proceeds to the Data Extraction stage (shown in figure 3). We will extract corresponding data from the Revit Model. The Model Derivative service and the Viewer service has also been used in this stage. To extract the model, the Model Derivative API will be firstly used to get the whole objects' manifest file. Then, the Viewer API will scan the whole model to make sure which objects should be fetched and then communicate with the Model Derivative API to extract the information from the manifest file.

A variety of data scanning methods are available through the Viewer API: (1) bounding boxes, (2) ray interactions and (3) box colliders. A bounding box is an invisible three-dimensional space that contains an entire object and all the edges are facing the cardinal direction. In this method, a bounding box is returned for each object by fetching the ID from the object's database as a parameter. Complex shapes can be accurately extracted using the ray intersection method. This method generates a ray for each pixel intersecting the bounding box of the BIM model and determines the objects by selecting each object's database ID in an array and a ray, an instance of ray caster. Then, the extraction records each ray intersection as points to a voxel format with the object type. The box-collider uses an SQLite database for extracting the metadata from the BIM geometry. We send queries from the client-side viewer with the model's properties such as model's id, and the type of objects (walls, doors, windows, etc.). Their respective database ids are transmitted to the server for extracting the BIM data. Next, the server downloads the list of files related to the project and the database of properties from Forge.

At this stage, the Model Derivative API is used for extracting information about all objects within the model first. As shown in figure 4, the model derivative service will return the information such as the Database ID and the type of the object through "GET metadata" endpoint. Once the fetch objects are finished, we use the bounding box method to obtain the overall length and width of the model and then divide this configuration into smaller cells. We now turn to the ray intersection method. Each Database ID of ray intersection points will be compared to the Database ID of the objects we extracted in the first place. If the intersection point has the same Database ID, then the position of the intersection point, Database ID, and the type of the object will be recorded. After going through all the intersection points, we can get a whole configuration file. Once the model has been extracted, we need to connect to the Solace event Broker for model publication. This process will be explained later.

For the data visualization component, we used the Viewer service and D3 JavaScript library which allows people to bind data to a document object model to show the Cell-DEVS simulation results, which maps these results on the BIM model.

As we discussed in Chapter 2, Cell-DEVS combines DEVS and cellular automata where each cell acts as a DEVS atomic model. It is natural to use the point-cloud technique to visualize Cell-DEVS models, because we can bind each point to a cell in a Cell-DEVS model. A point-cloud is a dataset of points under a coordinate system. Each of these points could contain varied information, including three dimensional coordinates, color, classification, intensity, time, and so on. Therefore, we can map Cell-DEVS simulation results into the point clouds, as the simulation results of a cell in Cell-DEVS can contain varied information such as status, position, time, etc., and a point in a point cloud could carry this information and show it using different point sizes, colors, and etc.

Considering the fact that our models are built using Cell-DEVS, and having seen that the point cloud technique is well suited to show the results of the Cell-DEVS simulation, we now need to address how to use this technique in BIM model so that the Cadmium simulation results will be shown on the Forge platform. We will now detail how we use this technique to let Cell-DEVS simulation shows on the Forge's BIM model.

Before visualizing the results that we received from the Solace broker directly, we need to use D3.js to manipulate the simulation results first. D3.js is a tool that helps you create interactive graphics and charts using data. It works with common web languages like HTML, SVG, and CSS. It combines visual elements with a data-driven approach to make your graphs and charts engaging and dynamic. The d3.csv() function can load and parse the CSV (Comma-Separated Values) file that is sent to the Forge server and turn that data into an array of objects. Then, we can use these data for visualization. In order to match the filtered data to the BIM geometry, the coordinates of the cells in the array must have the appropriate offset. In this case, we use the THREE.BufferGeometry() function. BufferGeometry is a representative of mesh or point geometry. It stores information such as vertex position, colors within a buffer. The THREE.BufferGeomery() method is used to create a BufferGeometry. When we obtain the BufferGeometry of the BIM geometry, each cell can find a corresponding position in the geometry. Once the data has been mapped, it can be read and displayed within the Forge Viewer as a point cloud geometry. We then use the THREE.PointCloud()function to generate the points. This function is based on the point's material which is the size and color of the points and the BufferGeometry to create a point cloud geometry. This function will use two parameters: the point's material which is the size and color of the points, and the BufferGeometry to generate the points. And each point will be in a corresponding cell in BufferGeometry. Finally, the d3.select().append() function can append a new element with a specific name as the last child of each element. We use it to append information, cell's status, into each point, so the point could show the corresponding color and attributes within the visualization. The point clouds then will update at every timestamp as part of the animation. The UI provides a countdown timer to allow the Viewer to know how much time remains until the end of the simulation.

3.3 The Cadmium Component

Figure 12 shows the Cadmium component, which runs on the user's local devices. It includes three subcomponents: a BIM-to-DEVS parser, the Cadmium simulator, and the Data Refinement module.

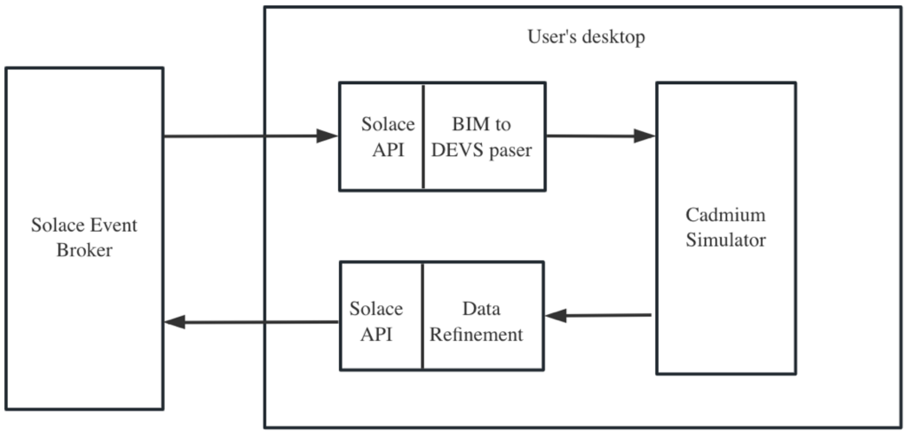


Figure 12. Software architecture using Cadmium simulator

The BIM to DEVS Parser is built as a python script which allows the received data to be converted into a format that can be used for the Cadmium simulator. The parser makes use of the coordinates system provided by the BIM data extraction to create a Cadmium configuration file. This tool analyzes the ray trace of the BIM model and generates the corresponding configuration file for the simulation of the Cell-DEVS model. The script first detects the scenario dimension based on the point set extracted from the BIM model. Then, it divides the space into smaller cubes and classifies each ray trace point with the cube that contains it. For each of these cubes, the script counts the number of points inside and classifies them according to their object type. Then, this converted JSON file could be used for Cadmium Cell-DEVS simulation. For each of these cubes, we first count the number of points in a cube and all these points are stored as tuples with the types of points and object IDs. Then, we count the number of point types that are in this cell and the number of points in each type. Finally, we can get the cell's type by comparing the number of points in each type. The type of point that has the most occurrences is used as the cell type. After we define the type of each cell, we add a Cell-DEVS default configuration (for instance, the default concentration of CO2 in a cell). Then, we use a dictionary to store the value of each cell and the whole cells will be stored in a list. Once the value is stored, we save the outcome in a JSON file.

When the simulation finishes, it will generate a text file containing the simulation results that we need. This state file logs the status of every cell at each timestamp, using the format: "*State for model CO2_lab_(x, y) is <counter, concentration, type>*". For instance, when we get "*State for model CO2_lab_(1,7,0) is <-1,500,-100>*", *"(1,7,0)*" represents the position of a cell as Cartesian coordinates, and "<*-1,500,-100*>" represents the cell's status. In this case, the first number is a counter that records the number of people in the simulation scenario and -1 is a default number. The second number is the concentration of CO2. And the third number represents the types of the cell (for instance, if it is a wall, a window, air, and so on).

However, the Forge platform needs to use CSV files to visualize the simulation results. These files should contain 6 parameters: time, x coordinate, y coordinate, previous value (for instance, the previous CO2 level in a CO2 simulation), current value (i.e., CO2 level) and the cell type. We use the Data Refinement subcomponent to convert the previous text file into a new CSV file.

The Data Refinement subcomponent is a python script that receives the results file from Cadmium, and it parses each line. Since each output event is represented in a separate line in the file, we can use regular expression operations to distinguish it from the cell's state line. Then, as explained above, the format of cells' position is "(x, y)" and the format of cells' state is "<counter, concentration, type>". Therefore, we could use find() method to determine the position of each parameter. Once we fetched all these parameters in one line, we output this line to an output file. Then continue until the end of the input file.

3.4 The Solace Component

Once the Forge or the Cadmium component decide to publish or subscribe a message, we need to connect them to the Solace Event Broker. Any software that wants to connect and communicate with the Solace broker must integrate the Solace API services because, as explained earlier, an application needs to connect to the Solace session to send and receive messages. The Solace session is the basis for all clients to communicate with the Solace message router. And Solace API can help users to create and connect Solace session.

Based on the Section 2.3.1, when we now discuss how to interact with the Solace broker. To do this, there are 4 main steps: (1) API Initiation; (2) Session Creation; (3) Session connection; and (4) Flow creation. In the first step, we use the *solace.init ()* function to initiate the Solace API. This function allows the Solace session to be created. Next, the Solace session allows the application to send and receive messages from the Solace broker. The second step focuses on creating and configuring the session with username, password, localhost address, etc. through the function *solace.createSession(session's properties)*. In the next step, we use the *session.connect()* function to let our application connect to the session. This function initiates the TCP session and send CONNECT message to the broker. Once the broker authorizes the connection, we are able to create publish flow and subscribe flow through the *publisher.session.send(message, topic)* method, which allows users to send and receive event messages. The parameter of the message could be the BIM data or simulation results and the topic is the address we want to send so that the broker knows where the destination is. On the subscriber side, we use the *subscriber.session.subscribe(topic)* function to receive messages and the event handler function *message.getBinaryAttachment()* provided to extract the model or simulation data from the event message. After the communication is complete, we use the function *session.disconnect()* to close the channel and then the applications cannot send and receive messages again.

We now show how these three components connect together and interact with each other. The entire communication process between Cadmium and Forge is shown in Figure 13.

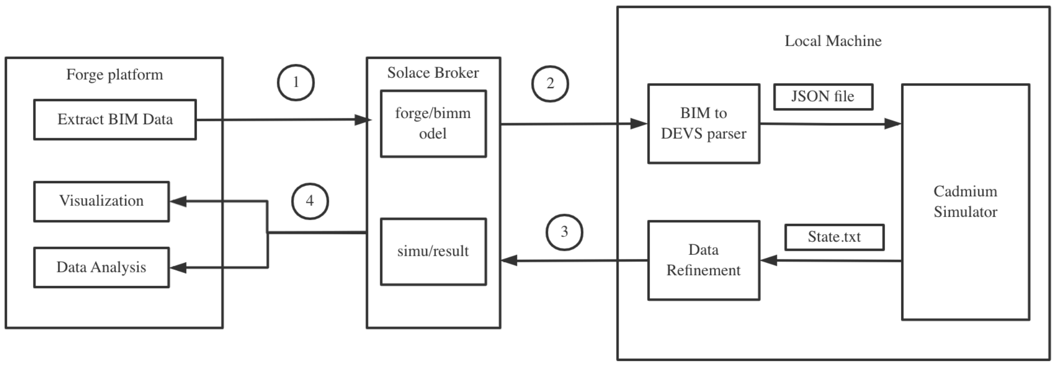


Figure 13. The process of communication between Forge and Cadmium

Initially, the modelers create their models on BIM360, and these models can then be uploaded to the Forge platform. As mentioned above, if the modelers wish to share models, the Forge platform can retrieve data from the BIM project via the Model Derivative API. Once the Forge platform has finished extracting the data, it publishes the data to a topic such as "forge/bimmodel" (Step 1 in figure 3). The topic has the format: a/b/... /c, where a, b, c, and so on are identifiers in a hierarchical scheme that you've designed that allows you to classify your data.

Step 2 executes on the Cadmium user’s side. To do so, we integrate the Solace API into the BIM to DEVS Parser, which allows the received data to be converted into a format that can be used for the Cadmium simulator. The BIM to DEVS parser will first connect to the broker and obtain the message by subscribing to the corresponding topic. The raw JSON file will be fetched from the message payload. This file will be stored on the user's local machine and then be converted to Cadmium Cell-DEVS configuration file. Users can use either the converted file for the Cadmium simulation or the raw file for other applications. Then, the Cadmium simulator runs the simulation based on the configuration file and generates a "state.txt" file that records the states of every cell at every time step.

Next, we extract the information such as time, coordinates, cell's state, and cell's type and then generate a CSV file at Data refinement stage. Next, we connect to the Solace broker and create a new Solace session and publish the CSV file to a new topic ("simu/result"), which is the step 3 presented in figure 13.

Finally, the Forge application connects to the same Solace message router and subscribes to the topic "simu/result" in order to obtain the CSV file and store it on the server. The Forge application visualizes the simulation results through the Forge Viewer or analyze the results through other Autodesk products. In our project, we use the Forge Viewer to visualize these results on BIM model by point-cloud technique that we explained above.

In conclusion, this is the entire process of allowing the Forge application and the Cadmium simulator to interact with each other through the Solace Event Broker.

Chapter 4: Study Cases

In the previous chapter, we presented the structure of our proposed framework using SOLACE for distributed simulation. In this chapter, we will discuss an example of using the software developed and an implementation of the framework using Cadmium, the Solace broker, and the Forge application. Additionally, we will discuss the proposed framework from an application point of view and include a case study implementation to support it.

As described in Chapter 3, the process of model transmission involves three steps – extracting the BIM data, publishing/subscribing the message, and converting the data to a Cell-DEVS configuration. Initially, the users choose a model to share on the Forge platform, and the application scans the model to fetch all objects and record them in a JSON file. Upon completion of the extraction process, the Forge application will send a JSON file to the Solace broker and wait for other users to subscribe to it. Upon receiving the message, the parser will convert the data into an appropriate format for the Cadmium simulator. The process for transmitting simulation results is similar – the application will convert the format of the simulation results, send them to the Forge platform through the Solace broker, and visualize them using the point-cloud technique.

We integrated the Solace API into the Forge platform and Cadmium simulation tool. The following section will delve into the implementation details, demonstrating how the processes work in practice.

4.1 BIM Model Transmission

As discussed in section 3.2, when users decide to send a BIM model, the Forge application will access its Revit model through the BIM 360 service and the Data management service. Then, the Forge application will send a request to its Model Derivative API to access all the information about that BIM project. All the processes described above are based on their original architecture or application. However, in the Data Extraction step, it is necessary to determine which information needs to be extracted and how the Solace API can be integrated to enable data transmission. This is the primary focus of our current implementation efforts. Figure 14 shows an excerpt of the whole architecture (explained earlier in Figures 9 and 13), in which we focus on the "Extract BIM Data" box, discussed in this section. This component retrieves the model through the Model Derivative Service to get whole objects' manifest file and then it scans the BIM model to extract the appropriate information from the manifest file.

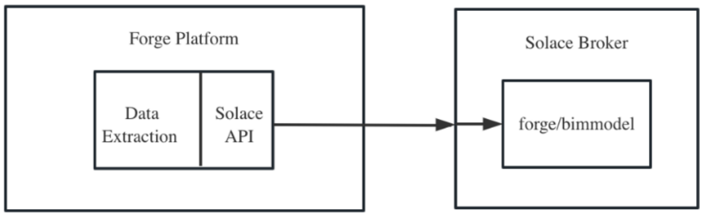


Figure 14. The cut of the whole system

The methods have been implemented according to the definitions provided in section 3.2. As discussed in that section, the data scanning methods were implemented using the bounding box and ray-shooter methods. To do this, the Data Extraction process, records the total length and width of the scenario for calculate the bounds of the room.

```
var cellSize  = 0.25;
const bounds = viewer.model.getBoundingBox();

const width = Math.floor((bounds.max.x - bounds.min.x) / cellSize);

const height = Math.floor((bounds.max.y - bounds.min.y) / cellSize);
```

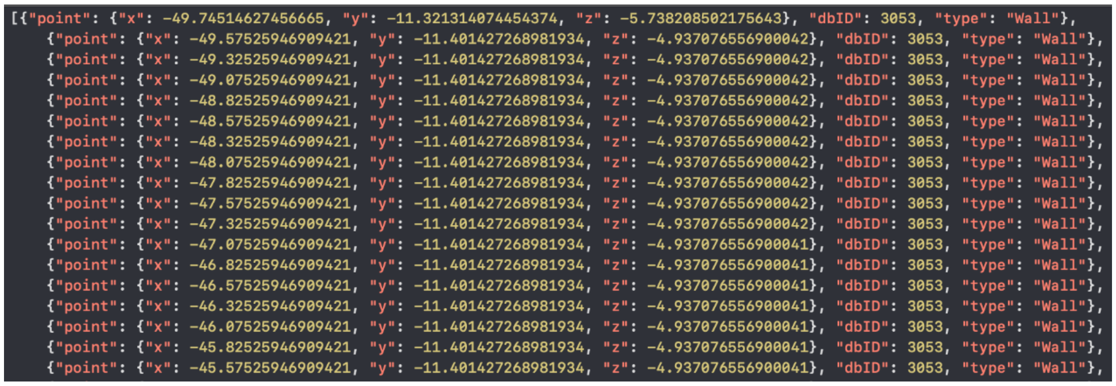
We first use the getBoundingBox() function for returning the overall length and width of the model. This is calculated by subtracting the minimum coordinate from the maximum coordinate (*bound.max-bound.** methods). In order to record all objects efficiently, we divide the layout into smaller cells of size 0.25m×0.25m×0.25m. Then, the layout of length and width will be represented as the number of cells, i.e., the *height* and the *width* above. In addition, the *Math.floor()* method is used for rounding down and returning the largest integer less than or equal to a given number.

After this step, we must record all the objects in the model. Then, the ray-shooter method can generate the corresponding number of rays for those cells. We generated rays to scan the bounds to determine the object type of each cell by comparing the objects with database ID. Once we make sure the object type and position of a cell, we store their information, and add color for the cell based on its object type (for example, "Wall" is black, and "Window" is red). The colors are represented as a combination of red, green, blue, and opacity values, each ranging from 0 to 255, as follows.

```
for (let j = 0; j < height; j++) {

   for (let k = 0; k < width; k++) {

        ray.origin.x = bounds.min.x + k * cellSize;

        ray.origin.y = bounds.min.y + j * cellSize;

        const intersection = viewer.impl.rayIntersect(ray, false, dataID);
        if (intersection){
        //getting the intersected points to be used for Cadmium input
            for(let c = 0; c < data.length; c++) {
                if(data[c].id == intersection.dbId)
                   intersectionData.push({"point":intersection.
                     intersectPoint, "dbID":intersection.dbId,
                          "type":data[c].type});
```

For each cell, we set the x and y coordinates of the origin property of the ray object. Then, **.rayIntersect()* is called with the ray object as the first argument, *false* as the second argument (which means the method should not perform any occlusion tests), and *dataID* as the third argument. The *dataID* is the database ID of an object from the manifest file. If the *rayIntersect()* method returns an object, it means that the ray intersected an object in the viewer. In this case, the code loops through an array of *data* (which is a data list extracted from the manifest file), and checks if the *dbId* property of the intersection object matches the id property of any of the *data* objects. If there is a match, the code pushes an object to the *intersectionData* array, which contains the intersection point of the ray with the object, the *dbId* of the object, and the type of the object.

Then, we add the color information into this intersection data. Different type of objects will have different value. Figure 15 shows an example of the output from this module as a JSON file. It records the position coordinates of each intersection point we explained above, the databaseID from BIM360 database, and the object type. Each object type has a specific databaseID, for example, “Wall” is 3053, “Door” is 3127, “Window” is 3170.



```
[{"point": {"x": -49.74514627456665, "y": -11.321314074454374, "z": -5.738208502175643}, "dbID": 3053, "type": "Wall"},
    {"point": {"x": -49.57525946909421, "y": -11.401427268981934, "z": -4.937076556900042}, "dbID": 3053, "type": "Wall"},
    {"point": {"x": -49.32525946909421, "y": -11.401427268981934, "z": -4.937076556900042}, "dbID": 3053, "type": "Wall"},
    {"point": {"x": -49.07525946909421, "y": -11.401427268981934, "z": -4.937076556900042}, "dbID": 3053, "type": "Wall"},
    {"point": {"x": -48.82525946909421, "y": -11.401427268981934, "z": -4.937076556900042}, "dbID": 3053, "type": "Wall"},
    {"point": {"x": -48.57525946909421, "y": -11.401427268981934, "z": -4.937076556900042}, "dbID": 3053, "type": "Wall"},
    {"point": {"x": -48.32525946909421, "y": -11.401427268981934, "z": -4.937076556900042}, "dbID": 3053, "type": "Wall"},
    {"point": {"x": -48.07525946909421, "y": -11.401427268981934, "z": -4.937076556900042}, "dbID": 3053, "type": "Wall"},
    {"point": {"x": -47.82525946909421, "y": -11.401427268981934, "z": -4.937076556900042}, "dbID": 3053, "type": "Wall"},
    {"point": {"x": -47.57525946909421, "y": -11.401427268981934, "z": -4.937076556900042}, "dbID": 3053, "type": "Wall"},
    {"point": {"x": -47.32525946909421, "y": -11.401427268981934, "z": -4.937076556900042}, "dbID": 3053, "type": "Wall"},
    {"point": {"x": -47.07525946909421, "y": -11.401427268981934, "z": -4.937076556900041}, "dbID": 3053, "type": "Wall"},
    {"point": {"x": -46.82525946909421, "y": -11.401427268981934, "z": -4.937076556900041}, "dbID": 3053, "type": "Wall"},
    {"point": {"x": -46.57525946909421, "y": -11.401427268981934, "z": -4.937076556900041}, "dbID": 3053, "type": "Wall"},
    {"point": {"x": -46.32525946909421, "y": -11.401427268981934, "z": -4.937076556900041}, "dbID": 3053, "type": "Wall"},
    {"point": {"x": -46.07525946909421, "y": -11.401427268981934, "z": -4.937076556900041}, "dbID": 3053, "type": "Wall"},
    {"point": {"x": -45.82525946909421, "y": -11.401427268981934, "z": -4.937076556900041}, "dbID": 3053, "type": "Wall"},
    {"point": {"x": -45.57525946909421, "y": -11.401427268981934, "z": -4.937076556900041}, "dbID": 3053, "type": "Wall"},
```

Figure 15. The JSON file of BIM data

When the extraction finishes, the file is sent to the Solace broker, as seen in in Figure 14 (the arrows from the Solace API to forge/bimmodel).

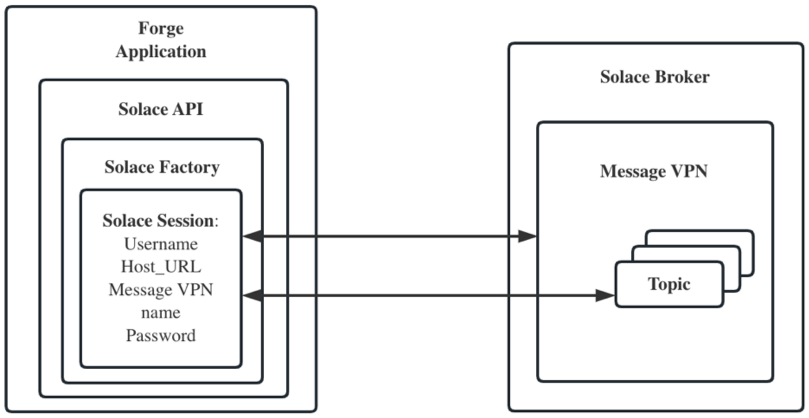


Figure 16. The architecture of sending message

As mentioned in section 3.4, to send or receive a BIM model through Solace, we must connect to Solace first (the process of building connection between the Solace session and the Message VPN is described in figure 16). There are three steps to connecting to the Solace event broker: loading and initiating the Solace factory, creating a Solace session, and connecting to the session.

The first step is loading and initiating the factory (or context). The factory acts as a container for sessions, as seen in figure 16. As discussed in Chapter 3, the Solace session is embedded inside of the Solace factory. It encapsulates the network IO and event notification threads, and it is the first entry point to the Solace API. The following snippet shows part of the implementation of this process. We generated an instance of Solace Factory named *window.factoryProps* at the global scope. As explained earlier, to generate the Solace session, we first need to generate the factory . Before initiating the factory, we set the version of Solace API that the factory will use. And then, the **.init()* method from Solace API will help us to initiate the factory as shown in figure 16.

```
// Initiate factory with the most recent API defaults
window.factoryProps = new solace.SolclientFactoryProperties();

factoryProps.profile = solace.SolclientFactoryProfiles.version10;
solace.SolclientFactory.init(factoryProps);
```

The second step is creating the Solace session part in Figure 16. The session is the communication channel between the API and the broker. It is responsible for client connection, publishing, and receiving messages. The factory is used to create a session by defining the properties of a session. The following code snippet shows a part of this implementation

```
try {

    publisher.session = solace.SolclientFactory.createSession({
    // solace.SessionProperties

    url: "ws://localhost:8008",
    vpnName: "default",

    userName: "cl1",
    password: "",

         });
}

// connect the session

try {
    publisher.session.connect();
}
```

We create a Solace session instance called *publisher*. Then, the **.createSession()* creates a session with the parameters: URL, VPN name, and username.

The last step is connecting the session to a Solace message router. For this, we only need to call the **.session.connect()* function and this will create the connection shown figure 16 between the Solace session and the Message VPN. This is also the connection from Data Extraction to Solace broker in Figure 14.

Based on this session, we added the topic name “forge/bimmodel” to the publisher. As mentioned in section 3.4, the topic has a hierarchical scheme to classify the data. In our case, “forge” is the root topic, and “bimmodel” is the subtopic. The date from BIM 360 will be published to this topic through our Forge app. Then, the Forge application connects to this specific topic automatically when the messages are published, as shown in Figure 16.

As for the process of connecting the BIM-to-DEVS parser to the Solace broker, this is basically done in the same way as the connection of Forge platform to the Solace Broker. After the Forge platform and the BIM-to-DEVS parser connect to the Solace broker, these two applications can communicate with each other at different times and locations.

Next, we will discuss the implementation of the BIM-to-DEVS parser, which allows the received BIM data to be converted into Cadmium scenario configuration, as shown in Figure 17.

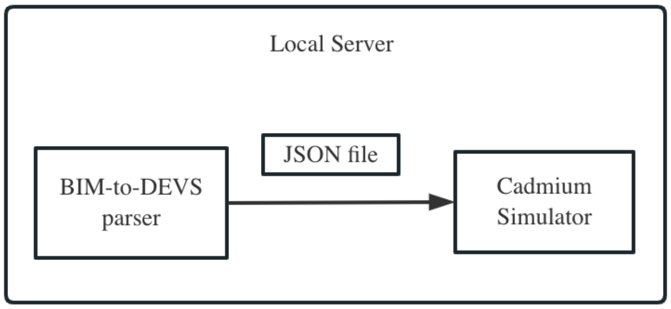


Figure 17. Cadmium component structure (excerpt)

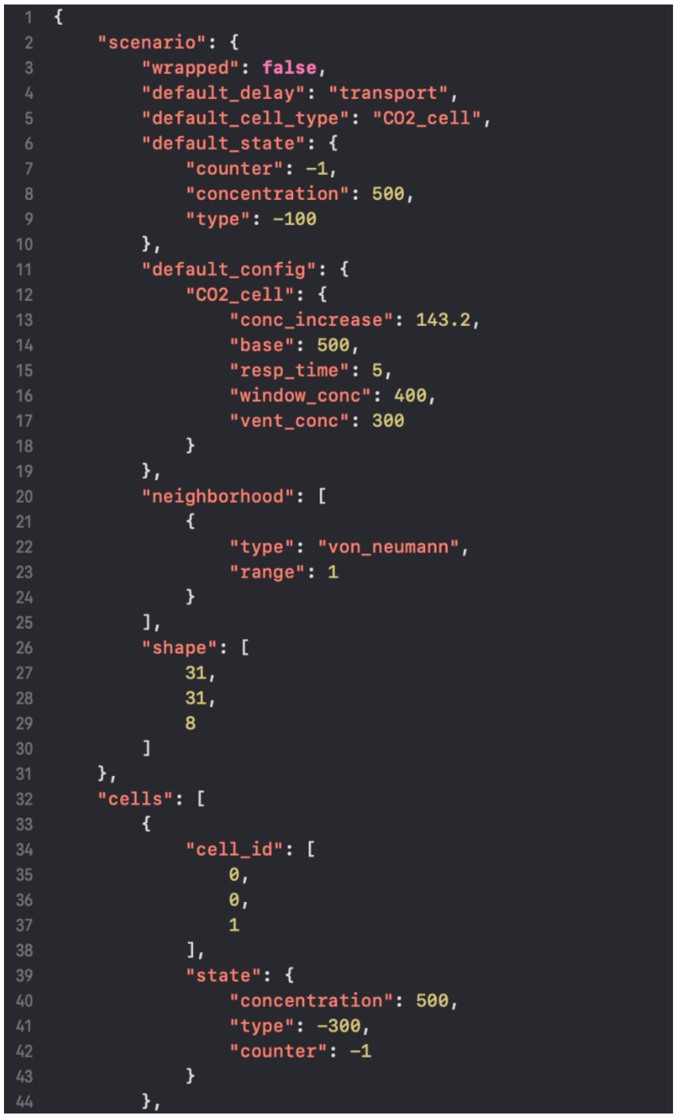
As discussed in section 3.3, The BIM to DEVS Parser allows the BIM data to be converted into a format that can be used for the Cadmium simulator. We adapted the data to a corresponding format and change the offset of all coordinates by finding the minimum coordinate for each dimension, so that we could get an unbiased BIM data and store it in a dictionary named *data_unbiased*. Then we went point by point from *data_unbiased.items()* as shown in snippet to identify which cell this point corresponds to and appended this point into the list: *data_cell[cell],* in this case, each cell is a cubic meter.

```
data_cell = dict()
 for coordinate, data in data_unbiased.items():

    cell = tuple(int(coordinate[i] // cell_size[i]) for i in range(len(cel_size)
))
     if cell not in data_cell:

         data_cell[cell] = list()
     data_cell[cell].append(data)
```

After computing the scenario shape (which is the transform from the original points position to cell position) and adding all the cells to the scenario configuration, we added the custom information for those cells such as the concentration of CO2, which is the *aux* in snippet. The *get_cell_type()* function will count how many points of each-type points are in the cell to determine the cell's type based on the list *data_cell[cell]* we obtained above.

```
cells = list()

for cell_id, data in data_cell.items():
    aux = {

        'cell_id': cell_id,
        'state': {

            'concentration': 500,
            'type': get_cell_type(data),

            'counter': -1,
        },

    }
    cells.append(aux)
```

Figure 18 shows the output (or the JSON file) from the BIM-to-DEVS parser module in figure 13/17. Compared with the output file from the model-extraction component, it becomes more readable. The file adds the default configuration such as the type of the CO2 spread, CO2 level, number of people, and so on. In addition, it converts the x, y, z coordinates into the more readable cell's position which is the "cell_id".



```
{
    "scenario": {
        "wrapped": false,
        "default_delay": "transport",
        "default_cell_type": "CO2_cell",
        "default_state": {
            "counter": -1,
            "concentration": 500,
            "type": -100
        },
        "default_config": {
            "CO2_cell": {
                "conc_increase": 143.2,
                "base": 500,
                "resp_time": 5,
                "window_conc": 400,
                "vent_conc": 300
            }
        },
        "neighborhood": [
            {
                "type": "von_neumann",
                "range": 1
            }
        ],
        "shape": [
            31,
            31,
            8
        ]
    },
    "cells": [
        {
            "cell_id": [
                0,
                0,
                1
            ],
            "state": {
                "concentration": 500,
                "type": -300,
                "counter": -1
            }
        },
```

Figure 18. The converted-configuration file

In Figure 18, the object "scenario" represents the default set of the entire model. Within this object, the parameter "wrapped" denotes the type of border cell used. If the border cells are wrapped, they can communicate their results to neighbors on the opposite border. On the other hand, non-wrapped borders have fixed boundaries. If a cell is outside the cellular space, it will return an undefined value. The value "transport" corresponds to one type of delay used for the Cell-DEVS model. As we mentioned in Chapter 2, there are two types of delay: transport delay and inertial delay. In the case of transport delay, state changes must be communicated in the future. The values and scheduled times are stored in a local queue. In contrast, inertial delay prevents any scheduled change from occurring upon receiving an external event from a neighboring cell before the scheduled time

The "default_cell_type" represents the cell type defined in the Cadmium atomic model file. It can be assigned any name based on the modeler's definition used in the atomic model file. The "default_state" refers to the default cell states. In the default case, the cell type is air (represented by the number -100 in the figure) with a CO2 concentration of 500 ppm and no occupants (counter = -1). In the "default_config" setting, "CO2_cell" represents the default concentration of CO2 in a cell. Here, "conc_increase" indicates the incremental increase in CO2 level within a cell that contains a CO2 source. The base level of CO2 is 500 ppm, the time used to calculate the concentration increase is 5 seconds, the CO2 level around windows is 400 ppm, and the CO2 level around vents is 300 ppm.

Next, the "neighborhood" defines a finite set of nearby cells for each cell. In our scenario, we used von Neumann's neighborhood, which enables communication with four nearby cells (North, West, South, East), and "range : 1" means that a cell can only affect cells that are one grid away from it. The "shape" represents the overall configuration of cells in the scenario, with dimensions of 31×31×8 cells. Finally, "cells" is an array that includes all the cells in the scenario and their initial configuration. For instance, in Figure 18, the cell at position (0, 0, 1) has no occupant (or CO2 source), a CO2 level of 500ppm, and it is a wall (identified by the type -300).

After discussing the overview of the entire process, we will now show how this process is presented within the Forge app. Figure 19 shows the user interface of the Forge app. As previously discussed in section 2.4.2, the sidebar serves as the directory that enables users to locate and access the BIM 360 project through Forge. The right canvas represents the Viewer, which allows for visualization of the project. Within the Viewer, there is a default toolbar (labeled as D) that provides various functionalities for inspecting the entire BIM model, such as moving and rotating the view. Additionally, a customized bar (labeled as E) is dedicated to rendering our 3D model, and this bar will be further explained in section 4.2. Similarly, buttons A, B, and C consist of a customized bar. These buttons facilitate the connection to the Solace broker and the publication of the BIM models. Specifically, button A establishes the connection to the broker, button B is responsible for publishing the models, and button C allows for disconnecting from the broker.

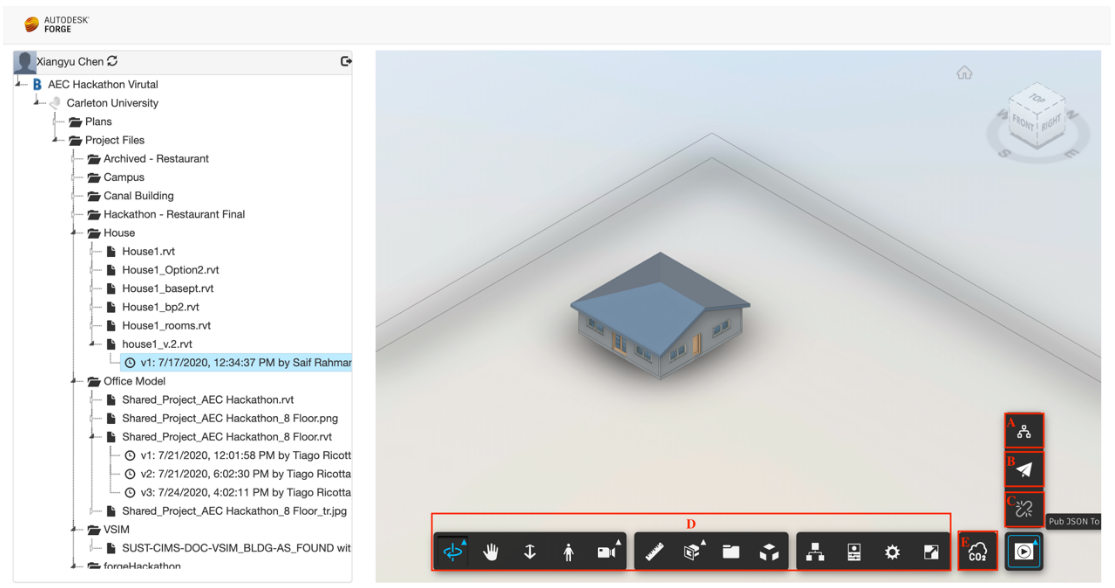
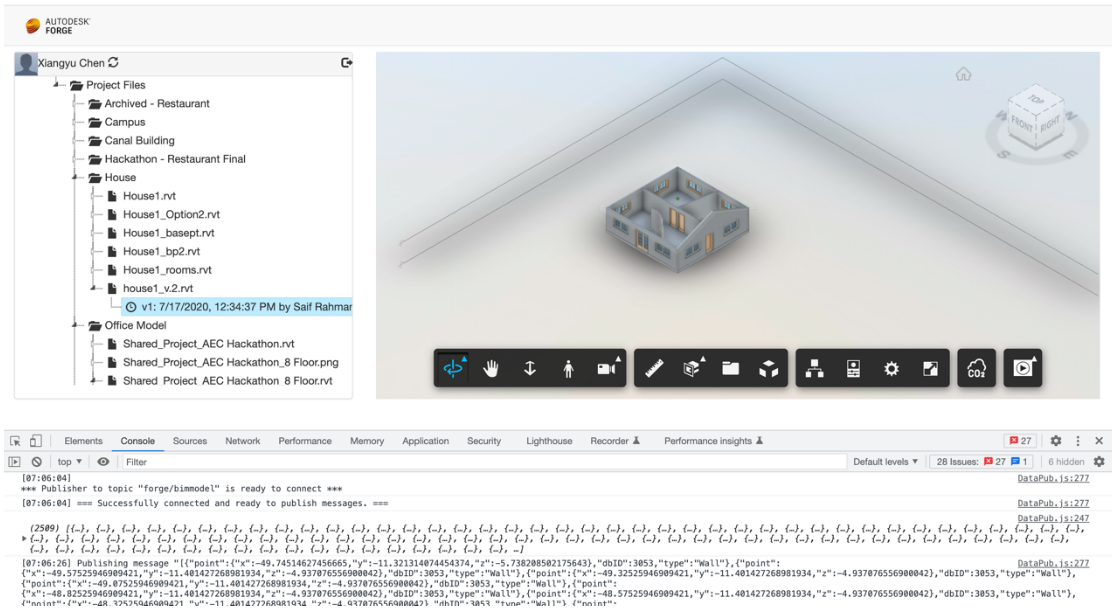

AUTODESK FORGE
Xiangyu Chen
AEC Hackathon Virutal
Carleton University
Plans
Project Files
Archived - Restaurant
Campus
Canal Building
Hackathon - Restaurant Final
House
House1.rvt
House1_Option2.rvt
House1_basept.rvt
House1_bp2.rvt
House1_rooms.rvt
house1_v.2.rvt
v1: 7/17/2020, 12:34:37 PM by Saif Rahmar
Office Model
Shared_Project_AEC Hackathon.rvt
Shared_Project_AEC Hackathon_8 Floor.png
Shared_Project_AEC Hackathon_8 Floor.rvt
v1: 7/21/2020, 12:01:58 PM by Tiago Ricott
v2: 7/21/2020, 6:02:30 PM by Tiago Ricotta
v3: 7/24/2020, 4:02:11 PM by Tiago Ricotta
Shared_Project_AEC Hackathon_8 Floor_tr.jpg
VSIM
SUST-CIMS-DOC-VSIM_BLDG-AS_FOUND wit
forgeHackathon
A
B
C
D
E
Pub JSON To

Figure 19. The interface of the Forge app

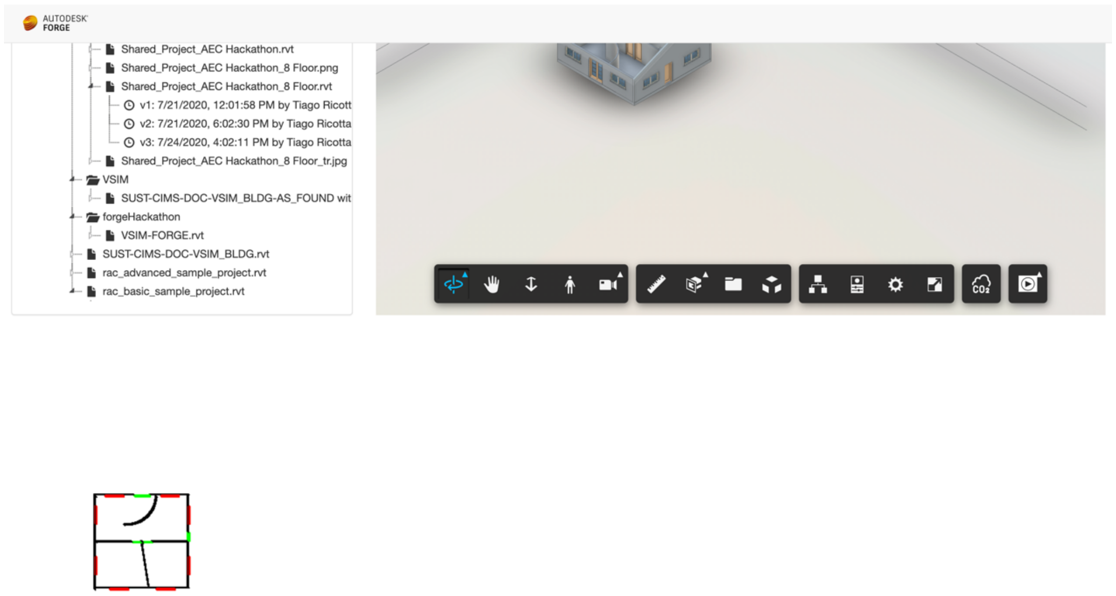
On the publisher side, we choose this specific house as the model to be transferred. To initiate the process, we establish a connection with the broker. Once the connection is established, we can proceed with sending the model. As depicted in Figure 20, the console section displays the app's successful connection to the broker and the designated topic, which in this case is "forge/bimmodel." Furthermore, the console indicates that the app is sending the message to the broker.



```
[07:06:04]                                                                    DataPub.js:277
*** Publisher to topic "forge/bimmodel" is ready to connect ***
[07:06:04] === Successfully connected and ready to publish messages. ===      DataPub.js:277
                                                                              DataPub.js:247
  (2509) [{…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…},
▸ {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…},
  {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, …]
[07:06:26] Publishing message "[{"point":{"x":-49.74514627456665,"y":-11.321314074454374,"z":-5.738208502175643},"dbID":3053,"type":"Wall"},{"point":   DataPub.js:277
{"x":-49.57525946909421,"y":-11.401427268981934,"z":-4.937076556900042},"dbID":3053,"type":"Wall"},{"point":{"x":-49.32525946909421,"y":-11.401427268981934,"z":-4.937076556900042},"dbID":3053,"type":"Wall"},
{"point":{"x":-49.07525946909421,"y":-11.401427268981934,"z":-4.937076556900042},"dbID":3053,"type":"Wall"},{"point":
{"x":-48.82525946909421,"y":-11.401427268981934,"z":-4.937076556900042},"dbID":3053,"type":"Wall"},{"point":{"x":-48.57525946909421,"y":-11.401427268981934,"z":-4.937076556900042},"dbID":3053,"type":"Wall"},
{"point":{"x":-48.32525946909421,"y":-11.401427268981934,"z":-4.937076556900042},"dbID":3053,"type":"Wall"},{"point":
```

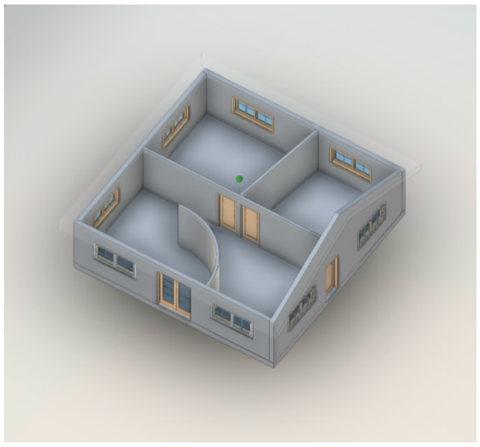
Figure 20. The connection of Forge and Solace

Once the publishing process is successfully completed, the console log will display the message "Message Published." Additionally, a new canvas will appear below the screen, presenting the model that has been extracted from the Revit model (as shown in Figure 21). And Figure 22 provides a comparison between the original model (Figure 22.a) and the extracted model (Figure 22.b).

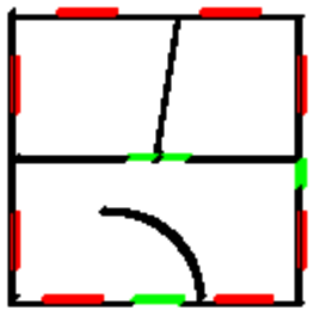


Shared_Project_AEC Hackathon.rvt
Shared_Project_AEC Hackathon_8 Floor.png
Shared_Project_AEC Hackathon_8 Floor.rvt
v1: 7/21/2020, 12:01:58 PM by Tiago Ricott
v2: 7/21/2020, 6:02:30 PM by Tiago Ricotta
v3: 7/24/2020, 4:02:11 PM by Tiago Ricotta
Shared_Project_AEC Hackathon_8 Floor_tr.jpg
VSIM
SUST-CIMS-DOC-VSIM_BLDG-AS_FOUND wit
forgeHackathon
VSIM-FORGE.rvt
SUST-CIMS-DOC-VSIM_BLDG.rvt
rac_advanced_sample_project.rvt
rac_basic_sample_project.rvt

Figure 21. The interface after the publishment



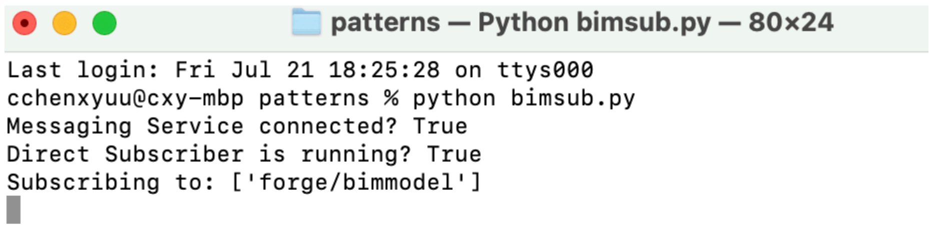
(a) The original BIM model



(b) The extracted model

Figure 22. The comparison of the original model and the extracted model

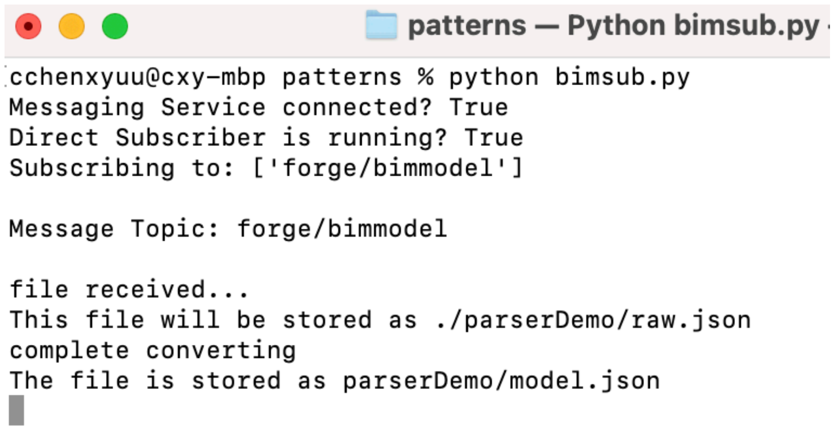
On the subscriber side, the process is simpler. As shown in Figure 23, our terminal serves as a “subscriber”. Once users run the BIM-to-DEVS parser, the terminal will display that we have successfully connected to the Solace broker using direct messaging. The topic name remains ‘forge/bimmodel,’ as mentioned earlier for the publish side. Subsequently, the parser enters a listening mode for the messages that are sent to the topic.



```
Last login: Fri Jul 21 18:25:28 on ttys000
cchenxyuu@cxy-mbp patterns % python bimsub.py
Messaging Service connected? True
Direct Subscriber is running? True
Subscribing to: ['forge/bimmodel']
```

Figure 23. The terminal of running BIM-to-DEVS Parser

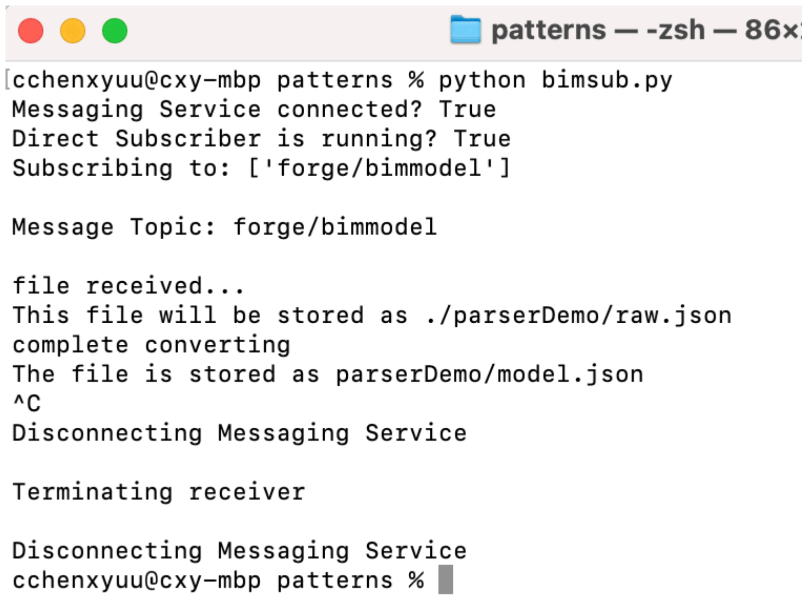
When the publisher (the Forge app) sends the BIM data to the broker, the parser receives this raw data and converts it into a Cadmium configuration, as depicted in Figure 24. The converted files will be stored in the "parserDemo/.." directory. The "raw.json" file contains the BIM data shown in Figure 15, while the "model.json" file holds the converted configuration displayed in Figure 18.



```
cchenxyuu@cxy-mbp patterns % python bimsub.py
Messaging Service connected? True
Direct Subscriber is running? True
Subscribing to: ['forge/bimmodel']

Message Topic: forge/bimmodel

file received...
This file will be stored as ./parserDemo/raw.json
complete converting
The file is stored as parserDemo/model.json
```

Figure 24. Receiving the files

The parser will continue listening to the topic until we choose to exit the application. When we decide to stop messaging, the parser will disconnect from the broker and terminate the application, as illustrated in Figure 25.



```
[cchenxyuu@cxy-mbp patterns % python bimsub.py
Messaging Service connected? True
Direct Subscriber is running? True
Subscribing to: ['forge/bimmodel']

Message Topic: forge/bimmodel

file received...
This file will be stored as ./parserDemo/raw.json
complete converting
The file is stored as parserDemo/model.json
^C
Disconnecting Messaging Service

Terminating receiver

Disconnecting Messaging Service
cchenxyuu@cxy-mbp patterns %
```

Figure 25. Teminating the service

4.2 Simulation Results Transmission

As discussed in the previous parts of this chapter, we implemented an application for using Solace to transmit BIM models, and later, we performed the process of sending data from Cadmium to Forge through the Solace broker as mentioned earlier. This section of the chapter will focus on the simulation results from the Cadmium simulator and how these results are transmitted and displayed on the Forge platform.

From section 3.3, once the Cadmium simulator receives the scenario configuration and runs the simulation, it generates two files, one containing the model’s output messages, and the other the internal state changes, as shown in Figure 26. The output message file can be used to debug as well as to identify the ports of an output stream. On the other hand, the simulation results can be studied by analyzing the state changes.

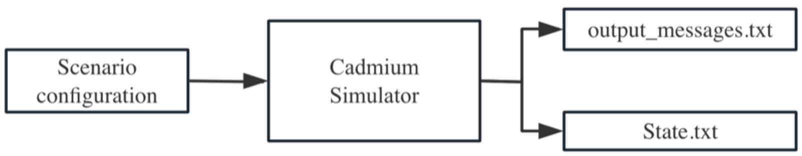
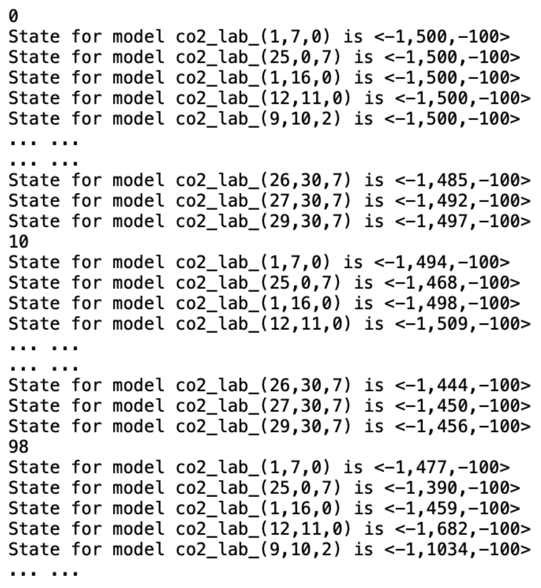


Figure 26. Cadmium simulation execution

Figure 27 illustrates parts of the simulation output at 0 seconds, 10 seconds, and 98 seconds. The file records the time on the first line and sequentially lists each cell's state, including its position, counter, CO2 level, and cell type.



```
0
State for model co2_lab_(1,7,0) is <-1,500,-100>
State for model co2_lab_(25,0,7) is <-1,500,-100>
State for model co2_lab_(1,16,0) is <-1,500,-100>
State for model co2_lab_(12,11,0) is <-1,500,-100>
State for model co2_lab_(9,10,2) is <-1,500,-100>
... ...
... ...
State for model co2_lab_(26,30,7) is <-1,485,-100>
State for model co2_lab_(27,30,7) is <-1,492,-100>
State for model co2_lab_(29,30,7) is <-1,497,-100>
10
State for model co2_lab_(1,7,0) is <-1,494,-100>
State for model co2_lab_(25,0,7) is <-1,468,-100>
State for model co2_lab_(1,16,0) is <-1,498,-100>
State for model co2_lab_(12,11,0) is <-1,509,-100>
... ...
... ...
State for model co2_lab_(26,30,7) is <-1,444,-100>
State for model co2_lab_(27,30,7) is <-1,450,-100>
State for model co2_lab_(29,30,7) is <-1,456,-100>
98
State for model co2_lab_(1,7,0) is <-1,477,-100>
State for model co2_lab_(25,0,7) is <-1,390,-100>
State for model co2_lab_(1,16,0) is <-1,459,-100>
State for model co2_lab_(12,11,0) is <-1,682,-100>
State for model co2_lab_(9,10,2) is <-1,1034,-100>
... ...
```

Figure 27 the fragment of the simulation results

Once the simulation is complete, the Data Refinement program converts the output file into an appropriate CSV format, as depicted in Figure 28 (or Figure 13).

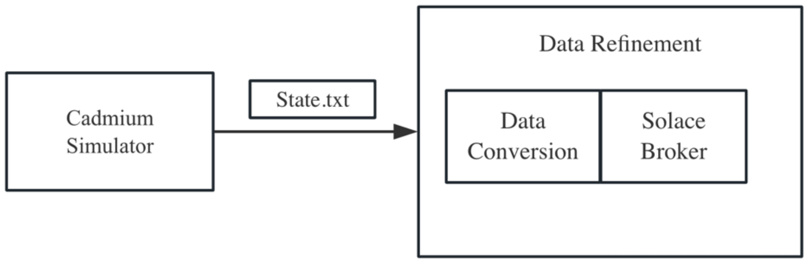


Figure 28. The process of data conversion

As discussed in section 3.3, the Data Refinement is a Python script. When this module receives the text file, it first converts it into a CSV file. Since simulation results can be quite large and the message size for Solace transportation is limited, we divide the data into several chunks and send them through the broker one by one. We will explain this aspect later; for now, let's focus on illustrating how we perform the data conversion.

The "convert_to_csv" function takes two file objects, "in_file" and "out_file," as arguments, representing the input and output files, respectively. The function reads each line of the input file and processes it based on its content.

During the conversion process, each line of the output file contains the following columns: time, x, y, previous_state, current_state, and type. To handle this, we create a dictionary called "states". This dictionary is used to store previous states associated with specific coordinates (x, y, and z). When processing new data, the function checks if the current coordinates are already present in the states dictionary. If found, it records the previous state's value for inclusion in the CSV output.

Regular expressions are employed to differentiate between time values and data values. Time values are identified by a pattern of consecutive digits, enabling the function to set the current time appropriately. Non-time lines are further processed to extract relevant data.

From the format of the state file, the function extracts the three values, x, y, and z, from the substring between the first occurrence of '(' and the first occurrence of ')'. It then converts these values to integers using list comprehension. Similarly, it extracts the parameters 's1', 's2', and 's3' from the substring between the last occurrence of '<' and the last occurrence of '>'. The function then converts these values to integers using list comprehension.

After completing the data conversion, we adopted a strategy to divide the large CSV data into smaller chunks to enhance transfer efficiency. Each data chunk was encapsulated in a Solace message and subsequently published by the Solace broker. The entire process is illustrated in Figure 29.

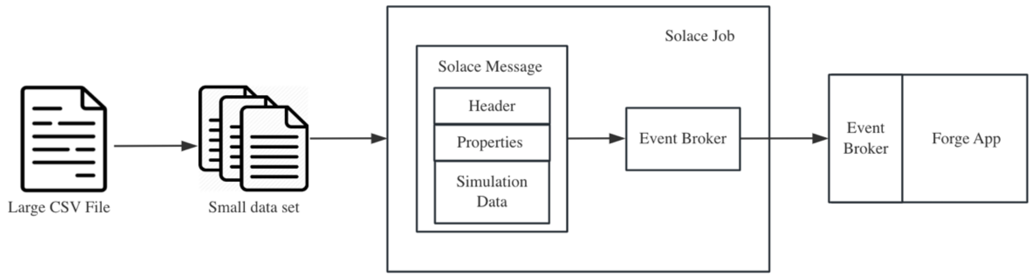


Figure 29. The process of message transportation

The first step in the process involved loading the CSV data into memory using the pandas library, creating a DataFrame to store the dataset. The pandas library is an open-source data manipulation and analysis library for the Python programming language. It provides data structures for working with structured data, such as tabular data, time series, and heterogeneous data. This approach was chosen for its ease of data manipulation due to its intuitive and user-friendly API that simplifies common data operations, such as filtering, sorting, joining, and grouping. It enables users to perform data transformations and analysis without the need for complex coding. The DataFrame was then transformed into a CSV string. This string represented the payload of the Solace messages that would be transmitted through the messaging service.

Next, we created a message builder responsible for constructing the Solace messages. The builder was initialized with properties such as "sample_id", "application" and "language" providing relevant metadata to the messages being published. These are custom properties that can be attached to the outbound message being built. These properties provide additional metadata or information related to the message content. In our program, the property "application" represents the name of the application that generated the message. The "language" indicates the programming language we used to create the message content.

With the setup complete, we proceeded to publish the data.

```
# Prepare outbound message payload and body

outbound_msg_builder = messaging_service.message_builder().with_application_message_id(
"sample_id").with_property("application", "samples").with_property("language", "Python"
)
TOPIC_PREFIX = "simu/result"
# Get the size of each chunk in bytes (approximately 500KB)

chunk_size_bytes = 500 * 1024
try:
   count = 1

   #while count <= 5:
   for chunk_start in range(0, len(message_body), chunk_size_bytes):

       #if count > 5:
           #break

       chunk_end = chunk_start + chunk_size_bytes
       chunk = message_body[chunk_start:chunk_end]


       topic = Topic.of(TOPIC_PREFIX + f'/{count}')

       outbound_msg = outbound_msg_builder \
                       .with_application_message_id(f'NEW {count}')\

                       .build(f'{chunk}')
       direct_publisher.publish(destination=topic, message=outbound_msg)
```

```
        print(f'Published chunk {count} on {topic}')

        count += 1
        time.sleep(1)
```

As show in the code snippet, to prevent large message sizes, the CSV data was divided into smaller chunks. The start and end indices of each chunk were defined based on the predetermined chunk size of 500KB, and the chunk was extracted from the CSV string representation accordingly.

With the Solace topic ready, we built a Solace message using the previously configured message builder. The current chunk was set as the payload for the message, and a unique application message ID was assigned to each message to facilitate efficient message routing and management.

This was achieved by using a predefined topic: "simu/result," followed by an incremental count value to ensure a distinct topic name for each chunk. For example, topics such as "simu/result/1," "simu/result/2," and so on were created for the respective chunks.

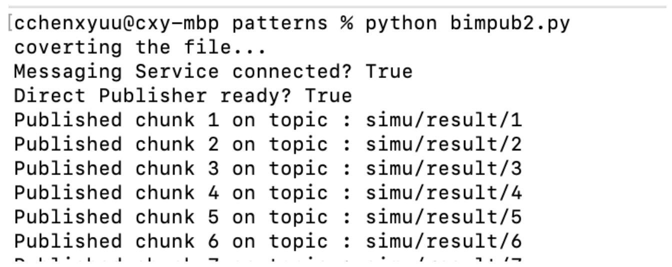
As mentioned in Section 3.4, the topic structure followed a hierarchical scheme with the format "a/b/... /c". Consequently, on the subscriber side, users could subscribe to the topic "simu/result/*" to receive all the chunks from the publisher.

Finally, we published the outbound message to the respective Solace topic using a direct publisher.

On the subscriber side, once the Forge app receives the complete simulation results, it proceeds to save the file and utilizes the D3.js library for CSV file manipulation and the point cloud technique for visualizing simulation data within the Forge Viewer, as elaborated in Section 3.2.

Figure 30 displays the terminal console of the Data Refinement program, functioning as a publisher. When the Cadmium simulator concludes the simulation, this program converts the simulation results into a CSV file. Following completion of the conversion, it establishes a connection with the Solace Broker and progressively publishes the chunked data via the directed publish method. Upon the successful publication of the entire simulation results, the app unsubscribes from the topic and disconnects to the broker.

Figure 30 displays results obtaining by the Data Refinement program functioning as a publisher. The application starts converting the simulation results into a CSV file, and following completion of the conversion, it establishes a connection with the Solace Broker and progressively publishes the chunked data via the directed publish method. Chunks are sent to the subtopics of “simu/result/”, and upon the successful publication of the entire simulation results, the app unsubscribes from the topic.



```
cchenxyuu@cxy-mbp patterns % python bimpub2.py
coverting the file...
Messaging Service connected? True
Direct Publisher ready? True
Published chunk 1 on topic : simu/result/1
Published chunk 2 on topic : simu/result/2
Published chunk 3 on topic : simu/result/3
Published chunk 4 on topic : simu/result/4
Published chunk 5 on topic : simu/result/5
Published chunk 6 on topic : simu/result/6
```

Figure 30.Data refinement program sending the simulation data

The Forge app functions as the subscriber in this scenario, as illustrated in Figure 31. Within the figure, there are options for the visualization of the simulation results, for subscribing to the Solace broker and receiving the simulation data, and to disconnect the Forge app from the Solace Broker.

In figure 31, the left side shows the BIM projects available and the right side is the Viewer. We can start visualization of the simulation results, subscribing and disconnecting to the Solace broker, etc.

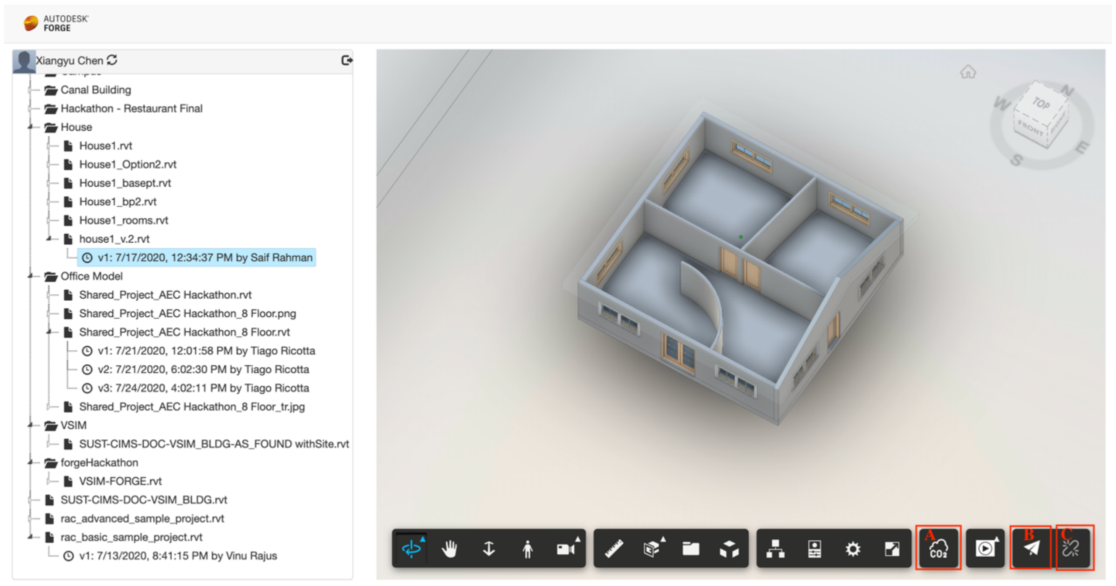
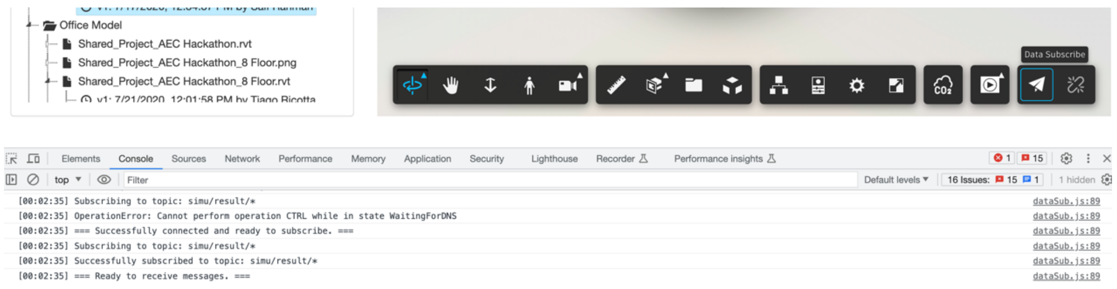

AUTODESK FORGE
Xiangyu Chen
Canal Building
Hackathon - Restaurant Final
House
House1.rvt
House1_Option2.rvt
House1_basept.rvt
House1_bp2.rvt
House1_rooms.rvt
house1_v.2.rvt
v1: 7/17/2020, 12:34:37 PM by Saif Rahman
Office Model
Shared_Project_AEC Hackathon.rvt
Shared_Project_AEC Hackathon_8 Floor.png
Shared_Project_AEC Hackathon_8 Floor.rvt
v1: 7/21/2020, 12:01:58 PM by Tiago Ricotta
v2: 7/21/2020, 6:02:30 PM by Tiago Ricotta
v3: 7/24/2020, 4:02:11 PM by Tiago Ricotta
Shared_Project_AEC Hackathon_8 Floor_tr.jpg
VSIM
SUST-CIMS-DOC-VSIM_BLDG-AS_FOUND withSite.rvt
forgeHackathon
VSIM-FORGE.rvt
SUST-CIMS-DOC-VSIM_BLDG.rvt
rac_advanced_sample_project.rvt
rac_basic_sample_project.rvt
v1: 7/13/2020, 8:41:15 PM by Vinu Rajus
TOP
FRONT
N
E
S
W
CO2
A
B
C

Figure 31.The UI of the subscriber side

When the Solace message encapsulate the simulation data and send them to the specific topic: “simu/result/*”, the Forge app integrates the Solace API, and it will connect to the same message broker and subscribe to the same topic to receive the simulation data for visualizing them on BIM model (shown in Figure 29/ step 4 in Figure 13).

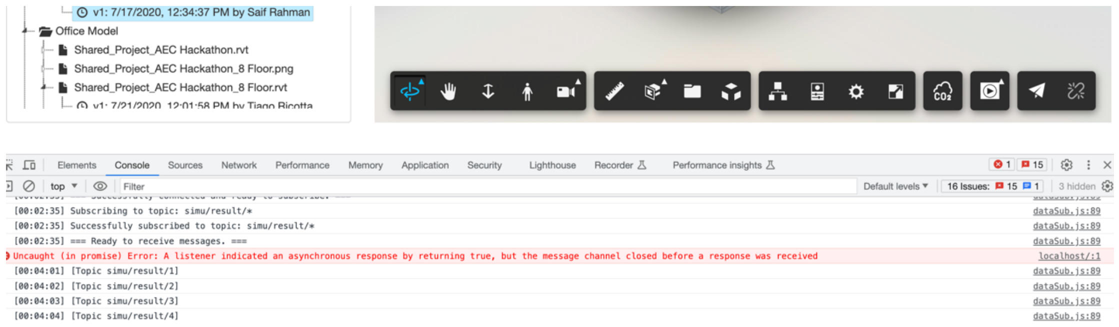
In Figures 32 and 33, when users subscribe, the Forge app establishes a connection with the broker, subscribes to the "simu/results/*" topic, and waits to receive data from the sending side.

Figure 32 shows how the Forge app connects to and subscribes to the broker. At timestamp “00:02:35”, Forge subscribes to “simu/topic/*” and is ready to receive data from the publisher side. In Figure 33, at "00:04:01", Forge received the first message from the topic "simu/result/1." Subsequently, it receives the remaining messages at a rate of one message per second.



```
[00:02:35] Subscribing to topic: simu/result/*                                  dataSub.js:89
[00:02:35] OperationError: Cannot perform operation CTRL while in state WaitingForDNS   dataSub.js:89
[00:02:35] === Successfully connected and ready to subscribe. ===               dataSub.js:89
[00:02:35] Subscribing to topic: simu/result/*                                  dataSub.js:89
[00:02:35] Successfully subscribed to topic: simu/result/*                      dataSub.js:89
[00:02:35] === Ready to receive messages. ===                                   dataSub.js:89
```

Figure 32. Connecting to the Broker


v1: 7/17/2020, 12:34:37 PM by Saif Rahman
Office Model
Shared_Project_AEC Hackathon.rvt
Shared_Project_AEC Hackathon_8 Floor.png
Shared_Project_AEC Hackathon_8 Floor.rvt
v1: 7/21/2020, 12:01:58 PM by Tiago Ricotta
Elements
Console
Sources
Network
Performance
Memory
Application
Security
Lighthouse
Recorder
Performance insights
top
Filter
Default levels
16 Issues: 15 1
3 hidden

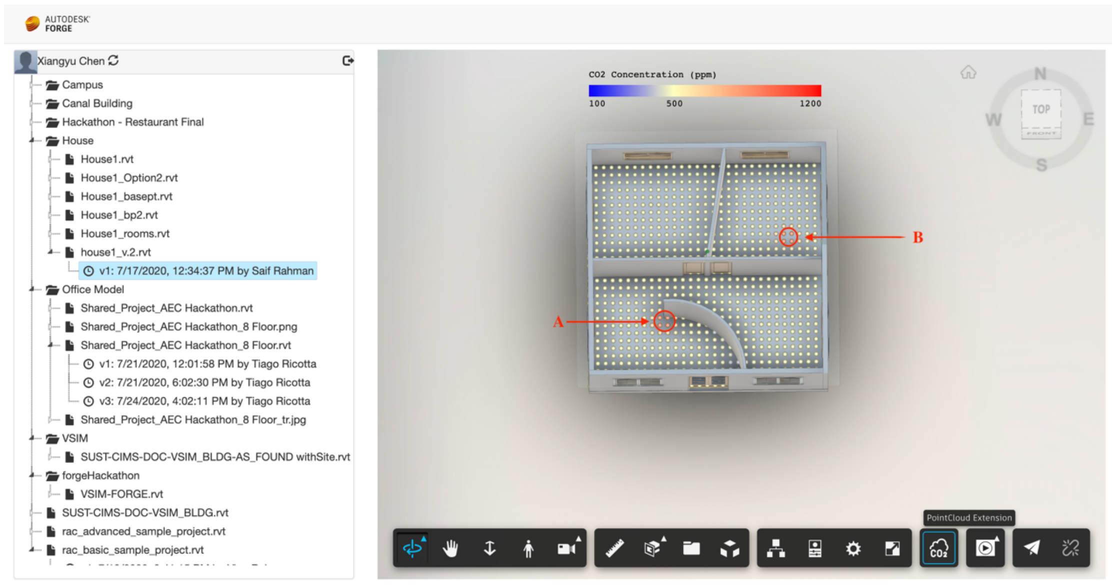
```
[00:02:35] Subscribing to topic: simu/result/*                                  dataSub.js:89
[00:02:35] Successfully subscribed to topic: simu/result/*                      dataSub.js:89
[00:02:35] === Ready to receive messages. ===                                   dataSub.js:89
Uncaught (in promise) Error: A listener indicated an asynchronous response by returning true, but the message channel closed before a response was received   localhost/:1
[00:04:01] [Topic simu/result/1]                                                dataSub.js:89
[00:04:02] [Topic simu/result/2]                                                dataSub.js:89
[00:04:03] [Topic simu/result/3]                                                dataSub.js:89
[00:04:04] [Topic simu/result/4]                                                dataSub.js:89
```

Figure 33. Receiving the simulation results

Upon receiving the data, users can proceed to visualize the simulation results, as seen in Figure 35. Unlike Figure 31, this visualization generates and maps multiple points above the BIM model. Each point corresponds to a CO2 concentration state within a cell. The color schema in the status bar indicates the CO2 level in the visualization, with darker red denoting higher CO2 concentrations. In our experiment, we designated places A and B as two CO2 sources within the house. Figure 36 illustrates the changing CO2 levels within the room.

Upon receiving the data, users can proceed to visualize the simulation results, as seen in Figure 35. There is a matrix of points above the BIM model, generated using the point cloud method introduced in section 3.2, obtained from the Cell-DEVS simulation results that Forge just received. Each point is mapped to one cell of the BIM model and corresponds to a CO2 concentration state within a cell. On the top of the Viewer, it is a color schema which indicates the CO2 level in the visualization (a CO2 concentration from

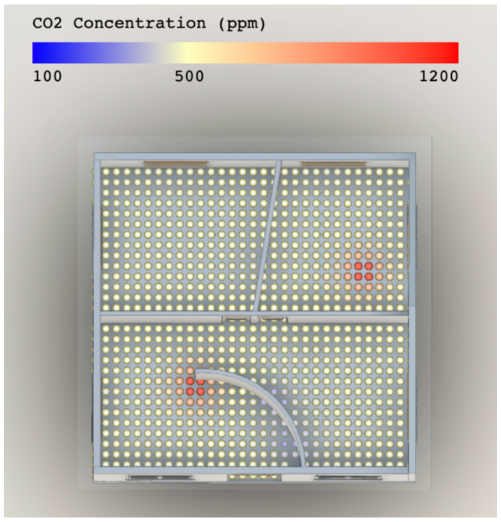
100ppm to 1200ppm, with darker red denoting higher CO2 concentrations). A and B represent two locations designated as CO2 sources within the house in the Cell-DEVS simulation.



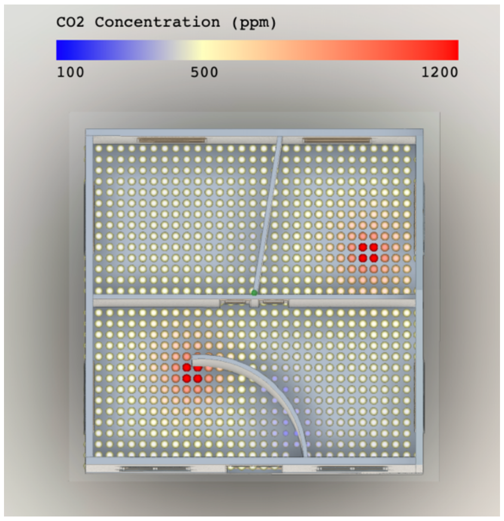
- Campus
- Canal Building
- Hackathon - Restaurant Final
- House
  - House1.rvt
  - House1_Option2.rvt
  - House1_basept.rvt
  - House1_bp2.rvt
  - House1_rooms.rvt
  - house1_v.2.rvt
    - v1: 7/17/2020, 12:34:37 PM by Saif Rahman
- Office Model
  - Shared_Project_AEC Hackathon.rvt
  - Shared_Project_AEC Hackathon_8 Floor.png
  - Shared_Project_AEC Hackathon_8 Floor.rvt
    - v1: 7/21/2020, 12:01:58 PM by Tiago Ricotta
    - v2: 7/21/2020, 6:02:30 PM by Tiago Ricotta
    - v3: 7/24/2020, 4:02:11 PM by Tiago Ricotta
  - Shared_Project_AEC Hackathon_8 Floor_tr.jpg
- VSIM
  - SUST-CIMS-DOC-VSIM_BLDG-AS_FOUND withSite.rvt
- forgeHackathon
  - VSIM-FORGE.rvt
- SUST-CIMS-DOC-VSIM_BLDG.rvt
- rac_advanced_sample_project.rvt
- rac_basic_sample_project.rvt

Figure 35. The visualization of the simulation data

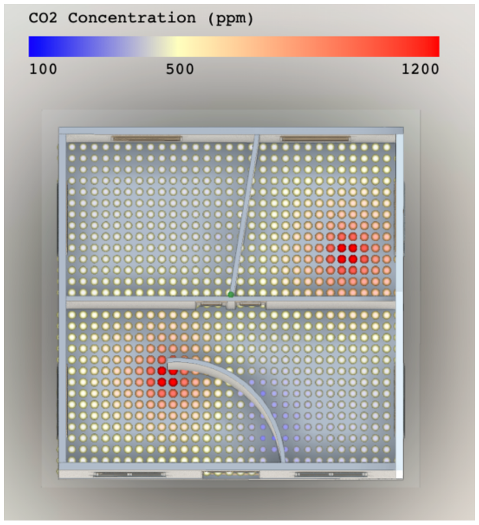
Figure 36 illustrates the changing CO2 levels within the room. In Figure 36.a, at the beginning of the simulation, the majority of cells exhibit normal CO2 concentrations (around 500ppm), except for the two CO2 sources. In the middle of the simulation (Figure 36.b), the CO2 concentrations increase in cells surrounding the sources due to inhalation and exhalation, while cells near doors and windows remain at lower levels, indicated by the color blue. At the end (Figure 36.c), the CO2 levels stabilize. Rooms with CO2 sources maintain higher CO2 levels, whereas rooms without sources exhibit significantly lower concentrations, such as the upper-left corner room.



(a) Early stage of the simulation



(b) Middle stage of the simulation



(c)Final stage of the simulation

Figure 36. Change in the CO2 concentration

Chapter 5: Conclusion

Modeling and Simulation is a complex field with many different processes. When we work on M&S projects, people like modelers, programmers, and analysts all play important roles. M&S researchers proposed distributed simulation methods, which allow different models to be executed on different computers using middleware techniques, which solved the limited resources for large simulation projects.

Although distributed simulation bridges a gap between simulation models, the gap between the different parts of the M&S life cycle still exists. This is because the independence of the software causes communication complexities. It is inconvenient to exchange information, such as scenario models and simulation results, between each software. Also, the requirement for developers to possess knowledge of multiple software adds a layer of complexity to the workflow.

In this thesis, we proposed a new method to apply distributed simulation to connect the modeling software, simulation software, and visualization software together by using a message broker.

As discussed in the document, the architecture consists of three components: Solace Event Broker, Autodesk Forge and the Cadmium simulator. We integrate Solace broker into Forge platform and Cadmium simulator so that the BIM models from Forge and the simulation results from Cadmium could be obtained by each other through subscribing to the same topic that we defined in Solace broker.

The case studies showed that we can use an event broker like Solace to build this integration. The broker's API provides a wide range of functions to facilitate data transmission, logging, and customization through property parameters that permits seamless integration of the components, allowing developers to use this architecture based on their original system without minimal changes.

In addition, some people may be concerned that the size of data may exceed the payload of a Solace message. The research introduced a way to deal with this problem by cutting a large file into smaller pieces. A subscriber who wants to receive the messages can easily get all the data by connecting to their parent topic.

In future work, an error feedback mechanism needs to be established so that the system can handle situations where subscribed messages do not match the messages that users require. And we could try other delivery modes provided by message brokers to enhance the asynchronicity of the system in various contexts.

References

[1] Balci, Osman. "A life cycle for modeling and simulation." Simulation 88.7 (2012): 870-883.

[2] Cetinkaya, Deniz & Verbraeck, Alexander & Seck, Mamadou. (2011). MDD4MS: A model driven development framework for modeling and simulation. Proceedings of the 2011 Summer Computer Simulation Conference. 113-121.

[3] Fujimoto, "Distributed simulation systems," Proceedings of the 2003 Winter Simulation Conference, 2003., 2003, pp. 124-134 Vol.1, doi: 10.1109/WSC.2003.1261415.

[4] R. Fujimoto, "Parallel and distributed simulation," 2015 Winter Simulation Conference (WSC), 2015, pp. 45-59, doi: 10.1109/WSC.2015.7408152.

[5] McLean, Thom, Richard Fujimoto, and Brad Fitzgibbons. "Middleware for real-time distributed simulations." *Concurrency and Computation: Practice and Experience* 16.15 (2004): 1483-1501.

[6] Al-Zoubi, Khaldoon, and Gabriel Wainer. "RISE: A general simulation interoperability middleware container." Journal of Parallel and Distributed Computing 73.5 (2013): 580-594.

[7] Yongguo, Jiang, et al. "Message-oriented middleware: A review." 2019 5th International Conference on Big Data Computing and Communications (BIGCOM). IEEE, 2019.

[8] Bishop, Toni A., and Ramesh K. Karne. "A Survey of Middleware." CATA. 2003.

[9] Taylor, Hugh, et al. Event-driven architecture: how SOA enables the real-time enterprise. Pearson Education, 2009.

[10] Michelson, Brenda M. "Event-driven architecture overview." Patricia Seybold Group 2.12 (2006): 10-1571.

[11] Dahmann, Judith S., Richard M. Fujimoto, and Richard M. Weatherly. "The department of defense high level architecture." Proceedings of the 29th conference on Winter simulation. 1997.

[12] Dahmann, Judith S. "The High Level Architecture and beyond: technology challenges." Proceedings Thirteenth Workshop on Parallel and Distributed Simulation. PADS 99.(Cat. No. PR00155). IEEE, 1999.

[13] Fishwick, Paul A. "Web-based simulation: some personal observations." Proceedings of the 28th conference on Winter simulation. 1996.

[14] Page, Ernest H., et al. "Web-based simulation: revolution or evolution?." ACM Transactions on Modeling and Computer Simulation (TOMACS) 10.1 (2000): 3-17.

[15] Cant, Robyn P., and Simon J. Cooper. "Simulation in the Internet age: The place of Web-based simulation in nursing education. An integrative review." Nurse Education Today 34.12 (2014): 1435-1442.

[16] Dormido, S., et al. "An interactive tool for introductory nonlinear control systems education." IFAC Proceedings Volumes 35.1 (2002): 255-260.

[17] Lazaridis, Vassilios, et al. "Visual LinProg: A web-based educational software for linear programming." Computer Applications in Engineering Education 15.1 (2007): 1-14.

[18] Méndez, J. Albino, et al. "A web-based tool for control engineering teaching." Computer Applications in Engineering Education 14.3 (2006): 178-187.

[19] Dominguez, Xavier, et al. "Web-Based Simulation Environment for Vehicular Electrical Networks." Energies 14.19 (2021): 6087.

[20] Saygin, Can, and Firat Kahraman. "A web-based programmable logic controller laboratory for manufacturing engineering education." The International Journal of Advanced Manufacturing Technology 24.7 (2004): 590-598.

[21] Calheiros, Rodrigo N., et al. "CloudSim: a toolkit for modeling and simulation of cloud computing environments and evaluation of resource provisioning algorithms." Software: Practice and experience 41.1 (2011): 23-50.

[22] Sriram, Ilango. "SPECI, a simulation tool exploring cloud-scale data centres." IEEE International Conference on Cloud Computing. Springer, Berlin, Heidelberg, 2009.

[23] Lim, Seung-Hwan, et al. "MDCSim: A multi-tier data center simulation, platform." 2009 IEEE International Conference on Cluster Computing and Workshops. IEEE, 2009.

[24] Sinha, Utkal, and Mayank Shekhar. "Comparison of various cloud simulation tools available in cloud computing." International Journal of Advanced Research in Computer and Communication Engineering 4.3 (2015): 171-176.

[25] Ma, Songhua, and Ling Tian. "A web service-based multi-disciplinary collaborative simulation platform for complicated product development." The International Journal of Advanced Manufacturing Technology 73.5 (2014): 1033-1047.

[26] She, Wei, I-Ling Yen, and Bhavani Thuraisingham. "WS-Sim: A web service simulation toolset with realistic data support." 2010 IEEE 34th Annual Computer Software and Applications Conference Workshops. IEEE, 2010.

[27] Meftali, Samy, et al. "SOAP based distributed simulation environment for System-on-Chip (SoC) design." Forum on Specification and Design Languages, FDL'05. 2005.

[28] Wainer, Gabriel A., Rami Madhoun, and Khaldoon Al-Zoubi. "Distributed simulation of DEVS and Cell-DEVS models in CD++ using Web-Services." Simulation Modelling Practice and Theory 16.9 (2008): 1266-1292

[29] What is Solace PubSub+ Platform. https://docs.solace.com/Solace-PubSub-Platform.htm

[30] What is an Event Broker? https://solace.com/what-is-an-event-broker/

[31] PubSub+ Event Broker. https://solace.com/products/event-broker/

[32] What is Messaging? https://docs.solace.com/Basics/Core-Concepts-Messaging.htm

[33] Message Exchange Pattern. https://docs.solace.com/Get-Started/Core-Concepts-Message-Models.htm

[34] Message Delivery Modes. https://docs.solace.com/Get-Started/Core-Concepts-Message-Delivery-Modes.htm

[35] Bononi, Luciano, et al. "ARTIS: a parallel and distributed simulation middleware for performance evaluation." International Symposium on Computer and Information Sciences. Springer, Berlin, Heidelberg, 2004.

[36] Lasnier, Gilles, et al. "Distributed simulation of heterogeneous and real-time systems." 2013 IEEE/ACM 17th International Symposium on Distributed Simulation and Real Time Applications. IEEE, 2013.

[37] B. R. Hiraman, C. Viresh M. and K. Abhijeet C., "A Study of Apache Kafka in Big Data Stream Processing," 2018 International Conference on Information , Communication, Engineering and Technology (ICICET), 2018, pp. 1-3, doi: 10.1109/ICICET.2018.8533771.

[38] Sax, Matthias J. "Apache Kafka." (2019).

[39] Justin Reock, “What is Apache ActiveMQ?” https://www.openlogic.com/blog/what-apache-activemq

[40] Apache ActiveMQ. https://activemq.apache.org/

[41] Sanjana Chand, George Busecan, “What is Forge?” https://forge.autodesk.com/blog/what-forge

[42] Adam Nagy, “What the Heck is Forge?” https://www.autodesk.com/autodesk-university/class/What-Heck-Forge-2017

[43] Wainer G., 2009. Discrete-Event Modeling and Simulation: A Practitioner’s Approach. 1st ed. CRC Press.

[44] Zeigler, B.P. Theory of Modelling and Simulation. N.p., 1976. Print.

[45] M. Etemad, M. Aazam and M. St-Hilaire, "Using DEVS for modeling and simulating a Fog Computing environment," 2017 International Conference on Computing, Networking and Communications (ICNC), 2017, pp. 849-854, doi: 10.1109/ICCNC.2017.7876242.

[46] Toba, Ange-Lionel, L. Michael Griffel, and Damon S. Hartley. "Devs based modeling and simulation of agricultural machinery movement." Computers and Electronics in Agriculture 177 (2020): 105669.

[47] A. Toba, M. Seck, M. Amissah and S. Bouazzaoui, "An approach for DEVS based modeling of electrical power systems," 2017 Winter Simulation Conference (WSC), 2017, pp. 977-988, doi: 10.1109/WSC.2017.8247848.

[48] A. C. H. Chow and B. P. Zeigler, "Parallel DEVS: a parallel, hierarchical, modular modeling formalism," Proceedings of Winter Simulation Conference, 1994, pp. 716-722, doi: 10.1109/WSC.1994.717419.

[49] Wainer, Gabriel A., and Norbert Giambiasi. "Application of the Cell-DEVS paradigm for cell spaces modeling and simulation." Simulation 76.1 (2001): 22-39.

[50] Al-Habashna, Ala'A., and Gabriel Wainer. "Modeling pedestrian behavior with Cell-DEVS: theory and applications." Simulation 92.2 (2016): 117-139.

[51] Belloli, Laouen, et al. "Building devs models with the cadmium tool." 2019 Winter Simulation Conference (WSC). IEEE, 2019.

[52] BIM 360 docs. https://construction.autodesk.com/sitecon/bim-360-autodesk-docs-demo/?_ga=2.129061708.1735661905.1658261335-1026838067.1635442483

[53] Rajus, Vinu Subashini, et al. "A cloud-based workflow for the integration of BIM to DEVS." Proc. SimAud2021 (2021).

[54] Introduction to IBM MQ. https://www.ibm.com/docs/en/ibm-mq/8.0?topic=overview-introduction-mq

[55] What can RabbitMQ do for you? https://www.rabbitmq.com/features.html

[56] Lovisa Johansson, Part 1: RabbitMQ for beginners-What is RabbitMQ? https://www.cloudamqp.com/blog/part1-rabbitmq-for-beginners-what-is-rabbitmq.html

[57] Overview: How Apps Interact with PubSub+ Messaging Components. https://docs.solace.com/API/Component-Maps.htm

[58] Bernstein, Philip A. "Middleware: a model for distributed system services." Communications of the ACM 39.2 (1996): 86-98.

[59] Ouyang, Min. "Review on modeling and simulation of interdependent critical infrastructure systems." Reliability engineering & System safety 121 (2014): 43-60.

[60] Cocco, Luisanna, and Michele Marchesi. "Modeling and Simulation of the Economics of Mining in the Bitcoin Market." PloS one 11.10 (2016): e0164603.

[61] da Silva, Ana C. Franco, et al. "Opentosca for iot: automating the deployment of iot applications based on the mosquitto message broker." Proceedings of the 6th International Conference on the Internet of Things. 2016.

[62] Oryema, Brian, et al. "Design and implementation of an interoperable messaging system for IoT healthcare services." 2017 14th IEEE annual consumer communications & networking conference (CCNC). IEEE, 2017.

[63] Rao, T. Ramalingeswara, et al. "The big data system, components, tools, and technologies: a survey." Knowledge and Information Systems 60 (2019): 1165-1245.

[64] Cheng, Bo, et al. "Industrial cyberphysical systems: Realizing cloud-based big data infrastructures." IEEE Industrial Electronics Magazine 12.1 (2018): 25-35.