Visualization and testing of an autonomously driving truck's SysML models in a virtual 3D simulation environment

Sanvordenker, R.S.

Award date:
2020

Technische Universiteit
Eindhoven
University of Technology

Department of Mathematics and Computer Science
Model Driven Software Engineering Research Group

Visualization and Testing of an Autonomously Driving Truck's SysML Models in a Virtual 3D Simulation Environment

Master Thesis
M.Sc Automotive Technology

Rudra Shailesh Sanvordenker
Student Id: 1326147

Supervisors:
Dr. ir. Ion Barosan
Dr. ir. Pieter Weterings

Version 1.0

Eindhoven, 9th July 2020

Declaration concerning the TU/e Code of Scientific Conduct for the Master's thesis

I have read the TU/e Code of Scientific Conduct[i].

I hereby declare that my Master's thesis has been carried out in accordance with the rules of the TU/e Code of Scientific Conduct

Date 8/4/20

Name Rudra Shailesh Sanvordenker

ID-number 1326147

Signature 

Insert this document in your Master Thesis report (2nd page) and submit it on Sharepoint

[i] See: http://www.tue.nl/en/university/about-the-university/integrity/scientific-integrity/
The Netherlands Code of Conduct for Academic Practice of the VSNU can be found here also.
More information about scientific integrity is published on the websites of TU/e and VSNU

Abstract

Model-Based Systems Engineering (MBSE) is becoming a standard approach to develop systems yielding complex components. MBSE offers a lot of freedom for interdisciplinary team members to communicate with each other in a way befitting their background, using SysML as a modeling language. However, even using a SysML modeling tool, a developer can be overwhelmed by the sheer amount of data he is exposed to due to limited visualization, leaving the door open to potential expensive development mistakes. The introduction of the Digital Twin gives developers the possibility to model, test, visualize, validate and execute various aspects of their SysML models in a 3D virtual environment, saving cost and offering an advanced comprehension of complexity involved. In this report, we present a method to integrate a 3D virtual environment, based on Unity3D with a added plugin named Prespective, with a SysML modeler tool, IBM Rhapsody.

Following the method, the developers can integrate and visualize SysML models inside a 3D virtual environment to execute, test, control, validate, and debug various aspects of their models. Also, a two way live-link is realized between Unity3D and IBM Rhapsody. To validate the method a prototype of an autonomously driving truck's SysML model is presented. The integration of the SysML modeler and the 3D virtual environment offers developers a better understanding and insight into the behavior and structure of the SysML models. Also, our approach can be applied to any SysML models, which increases the applicability of the method in any domain and not only automotive. Based on our methodology and research we frame and answer several research questions that are part of this report. We conclude that a basic prototype can be implemented that can help model based system engineers to visualize, control and test their SysML models in a virtual 3D environment.

Preface

I would like to express my sincere gratitude to my daily supervisor Pieter Weterings for guiding me throughout my graduation project. During these nine months, he has always been supportive and has always helped me reach my full potential by keeping a constant watch on me and my thesis. I would also like to thank Koen Grotenhuis, Bas Botermans and Wouter Vanmulken for providing me with the necessary direction to implement various parts of my thesis. Without your combined knowledge and expertise it would be impossible for me to carry out my research and implementation. I would also like to extend my sincere thanks Dr. Ion Barosan for firstly introducing me to Unit040 and giving me a direction to work on my critical thinking and analytical skills with the help of all the weekly meetings we had. Thank You for believing in me and pushing me harder to achieve more. Finally I would like to thank my family in India and friends here who have always supported me overcome every challenge and obstacle. This thesis has been a roller-coaster ride for me; however I emerged very contented and satisfied with my efforts combined with the support of all my mentors and friends.

Abbreviations

List of abbreviations:

- FMU - Functional Mock-up Unit
- FMI - Functional Mock-up Interface
- API - Application Programmable Interface
- HTML - Hypertext Markup Language
- MQTT - Message Queuing Telemetry Transport
- ActiveMQ - Active Message Queuing
- CAD - Computer-Aided Design
- SysML - System Modelling language
- XML - Extensible Markup Language
- TCP/IP - Transmission Control Protocol/Internet Protocol
- UDP - User Datagram Protocol
- UML - Unified Modeling Language.
- IOT - Internet of Things

Definitions

List of definitions:

- **Digital Twin** - A digital twin is a digital replica of a living or non-living physical entity.

- **Internet of Things** - The Internet of Things (IoT) is a system of computer devices and other types of machines that transfer data over a network without needing human interference. Each device on the internet of things are provided with unique identifiers.

- **Machine Learning** - Machine learning is an application of artificial intelligence that provides systems the ability to automatically learn and improve from experience without requiring the need to program them.

- **PRElogic** - The PRElogic module inside Prespective is a feature that help the user to connect to a mechanical component or an assembly with control logic systems outside Unity3D such as programmable logic circuits and other formal modelling tools.

- **RestAPI** - A RESTful API is an application program interface (API) that uses HTTP requests to get, put, post and delete data.

Contents

Contents vii

List of Figures ix

List of Tables x

1 Chapter 1 - Introduction 1
- 1.1 Problem Context . . . 2
- 1.2 Project Context . . . 2
- 1.3 System Context . . . 3

2 Chapter 2 - Research Problem 5
- 2.1 Research Questions . . . 5
  - 2.1.1 Question One . . . 5
  - 2.1.2 Question Two . . . 6
  - 2.1.3 Question Three . . . 6

3 Chapter 3 - Related Work 7
- 3.1 Functional Mock-up Interface . . . 7
- 3.2 Functional Mock-up Unit . . . 8

4 Chapter 4 - Methodology 11
- 4.1 Step 1 - Visualization of SysML Models . . . 13
  - 4.1.1 Visualization Tools . . . 14
- 4.2 Step 2 - Communication of IBM Rhapsody . . . 16
- 4.3 Step 3 - Embedding the Web Pages inside Unity3D . . . 18
- 4.4 Step 4 - Live-Link between IBM Rhapsody and Unity3D . . . 19
  - 4.4.1 MQTT . . . 20
  - 4.4.2 ActiveMQ . . . 20
  - 4.4.3 TCP/IP . . . 21
  - 4.4.4 UDP/IP . . . 23
  - 4.4.5 Comparison Study between TCP and UDP . . . 25
- 4.5 Implementation . . . 27

5 Chapter 5 - Discussion and Interpretation of Results **31**
5.1 State and Sub-state Machines in the System 31
5.2 SysML Block Definition Diagram 32
5.3 Rhapsody's Web Services and Live-Link between IBM Rhapsody and Unity3D 33

6 Chapter 6 - Conclusions and Future Work **35**

Bibliography **37**

Appendix **41**

A State Machine (Appendix A) **41**

B Sub-State Machine (Appendix B) **59**

C Block Definition Diagram (Appendix C) **73**

D Live-Link between IBM Rhapsody and Unity3D (Appendix D) **83**
D.1 IBM Rhapsody 83
D.2 Unity3D 87

List of Figures

3.1 Functional Mock-up Interface consisting of two components 7
3.2 Functional Mock-up Unit is an integration of two individual parts 8

4.1 Research Plan Overview and choices made along with research areas to focus 12
4.2 Web Server generated to publish model data from IBM Rhapsody on a localhost 13
4.3 Snippet of XML Data generated from models inside IBM Rhapsody 14
4.4 Publish and Visualize Data 16
4.5 Web Pages provided by Rhapsody's Web Service to manage and control models 17
4.6 Aggregate Table used to send values and commands to Rhapsody remotely 17
4.7 Script to input the URL, width, height and base color of the webpage . . . 18
4.8 Script to enable various inputs and control the drag threshold on the web page 19
4.9 Communication between various layers of the TCP/IP Protocol 22
4.10 Components present inside a TCP/IP data packet 22
4.11 Components present inside a UDP datagram along with the functions . . . 24
4.12 Communication between various layers of the UDP Protocol 24
4.13 Segment Fields of TCP and UDP [10]. 25
4.14 Pub/Sub Architecture where several clients communicate via one broker [28]. 27
4.15 Architecture of prototype along with logical flow of data and command . . 27
4.16 Components present inside the implemented prototype 28

5.1 State Machine and Sub-state Machine in Rhapsody 31
5.2 Truck Lab inside a Distribution Center with models 32
5.3 State and Sub-state Machine in Unity3D 32
5.4 Block Definition Diagram containing the blocks of the System 33
5.5 Rhapsody's Web Service embedded inside Unity3D 33
5.6 Values published and subscribed by Unity3D on the message broker 34

D.1 To Unity Block 84
D.2 Initialization Function 85
D.3 Send and Receive Function 86

List of Tables

4.1 Comparison between TCP and UDP [10]. 26

Chapter 1 - Introduction

As technology becomes more complicated so does the leading development process. The interdisciplinary domains in systems such as airplanes and automotive need to be integrated together so that they function effectively while keeping energy and cost of production under control. Using traditional methods of system development that involve a more document centric approach often leads to developers loosing track of the progress and design decisions made. Ultimately this results in the development of a system that is far from close to what was intended. The solution to this challenge is to use a systems engineering methodology that is more model centric and visual. From an industrial perspective, a methodology is considered to be useful and cost-efficient if it is possible to reuse solutions in multiple projects or products [29].

Model-based systems engineering, a multi-disciplinary systems engineering method addresses the complexity of this problem. MBSE uses modelling languages that help a developer to model, visualize, communicate and test models before deploying them, thus ensuring that every aspect of the model is validated [15]. The fundamental difference between systems engineering and MBSE is that in systems engineering the focus is on producing and controlling documentation about the system; whereas in MBSE the focus is on developing, managing and controlling the model of the system. Hence, Model based systems engineering aids in simplifying the design process thus helping in increasing both efficiency and productivity of the system [29].

With the introduction of the Digital Twin at the beginning of the 21st century system modelling has received more significance. A Digital Twin is a virtual 3D copy of an actual component that can be used to enact the same behaviour as that of its's real world twin with the help of the Internet of Things (IoT) and Machine learning (ML). Hence, a physical component and it's virtual 3D twin can transmit data between each other making it easier for the virtual 3D twin to detect potential errors and threats to the actual machine. This helps in reducing losses and increasing efficiency. Thus developer can fabricate an entire machine in a digital environment and test their high level control software virtually in 3D.

1.1 Problem Context

The integration of Model based systems engineering and Digital Twin can open several opportunities for model based systems engineers to test and visualize large models. A developer can perform the entire process of testing and debugging in a virtual 3D environment. By simulating their models in the virtual 3D environment and testing every assumption made a model based systems engineer knows that their system is valid. Currently there is limited support for introduction of formal models inside a virtual 3D environment through supporting Functional Mock-up Unit(FMU) [1] exchange models.

The major drawback of using Functional Mock-up Unit's for simulation is that the developer is quite unaware of how the models look and function inside an Functional Mock-up Unit. The user is only aware of the inputs given to the Functional Mock-up Unit and the outputs obtained. Hence, it is quite difficult as a developer who wants to validate the models and the controlling software to debug, which is using only Functional Mock-up Unit's. Secondly in case there are complications when an Functional Mock-up unit is exported or if the Functional Mock-up Unit was not exported properly from a particular software tool it can lead to bugs that are untraceable.

There are some validation tools available that help validate Functional Mock-up Unit's but they cannot be used with complete accuracy. The tools only serve the purpose of validating the integrity of the Functional Mock-up Unit algorithm. However there is no means of knowing if the algorithm gives the correct set of outputs given a set of inputs. Thirdly, there can be times when the number of functions performed by the Functional Mock-up Unit is larger than Unity3D can handle which may cause disruptive slowdowns of the Digital Twin. Also when marshalling needs to be done over different platforms it can result in a difference of memory layouts and may lead to a reduction in performance [1]. Thus cross platform interaction can be less efficient and more cumbersome to debug. Hence, we need to develop a system that is more real time in nature, is better to debug and offers good trace ability.

1.2 Project Context

A commonly used modelling language in model based systems engineering is SysML [3]. Using SysML a developer can model and validate every aspect of their system and also reuse parts of their validated models. SysML helps in improving the compatibility between various domains due to increased communication between different teams that are a part of the system integration. For our research and implementation we choose IBM Rhapsody [2]; a modelling environment based in SysML, that helps developers to automatically generate real time embedded code, from their SysML models. Inside IBM Rhapsody we will limit our research to state-machines at the meta model level and sub-state machines that are one hierarchical level below the meta model level. The animation of models inside

IBM Rhapsody is beyond the scope of our research. In addition we also consider block definition diagrams however; we limit our research to only the blocks in the BDD and omit the relations that each block has with the other.

A good platform to make a Digital Twin is Unity3D [5], a world leader in game engines. When it comes to augmented and virtual reality Unity3D dominates the market as it has several developer bases where the latest technological developments can be accessed. With the help of Prespective [4]; a plugin for Unity3D, a Digital Twin can be generated with the help of CAD Data. Unity3D has a very large developer base, ensuring access to the latest technological developments. On top of the Unity3D Architecture, Prespective adds a list of features making the creation of complex and realistic simulations possible. Features – on top of Unity3D – include: simulation of physics (Newtonian laws), smart 3D Geometry importing (e.g. SolidWorks STEP files), I/O coupling, scenario building for testing, multi user viewing, logging of simulations, and replaying issues [4]. Prespective is a software platform that is built on top of Unity3D where users can continuously test systems. A developer can connect to a logic control software using the Internet of Things. The PreLogic component inside Prespective helps a digital twin to connect to an external server or gateway for the exchange of data. A machine or a factory setup can be fully modelled at the digital level, reproducing the dynamic processes and behavior of system components. Simulation and Virtualization provide overview and insight, allowing model driven system engineers to test assumptions early in the development process [4]. For our testing and implementation we use a Digital Twin of an autonomously driving truck inside a distribution center that can load and unload goods.

1.3 System Context

This project focuses on an alternative way of introducing formal models into a digital twins is by creating a 2-way live-link between a IBM Rhapsody and Unity3D engine that can transfer data both ways and also provide a visualization and command prompting mechanism inside this virtual 3D environment. For our project I firmly believe that Bi-directional data transfer will allow for easier integration of digital twinning/prototyping in the development process since model assumptions can be validated on both the creating- and visualizing side. The project can be broken down into five steps to be followed as given below

- Extract data from SysML model inside IBM Rhapsody.
- Visualize the extracted SysML models.
- Establish a two way live-link between IBM Rhapsody and Unity3D.
- Implement a command prompting mechanism inside IBM Rhapsody that can be controlled remotely.

- Embed the visualizations of SysML models and the command prompting mechanism inside Unity3D.

The individual steps will be integrated together in the end to form our complete system prototype. The related work along with the research problem and methodology followed is formulated based on the drawbacks obtained from the problem context. The research along with the implementation and results are clearly discussed and a conclusion is drawn along with ideas for future work.

Chapter 2 - Research Problem

In this chapter we further elaborate on the drawbacks of using Functional Mock-up Unit and the need for research and implementing the proposed prototype. The research questions that we define for the project are enumerated in section 2.1. The fundamental difference between a live-link and using Functional Mock-up Unit's is that the Functional Mock-up Unit approach does not give the developer an option to debug the model in early stages of system development. The Functional Mock-up Unit acts like a black box where you put input and get output out of it. There is no means to know how the model looks and behaves or whether the Functional Mock-up Unit was exported properly. Furthermore, there is a great need for flexibility and short development loops, hence embedding the Digital Twin with an Functional Mock-up Unit is not an ideal option. The virtual 3D environment offered to developers should have real time behaviour as the aim to implement virtual 3D testing is to create an almost same scenario that a developer would have testing models on a real machine.

But in order to implement the proposed prototype it is necessary to know whether a basic prototype can be researched and implemented. In addition, if we are successful in implementing a basic prototype can it be scaled up; so that developers can visualize and debug even larger models containing larger amounts of data. To answer these questions we formulate a research and adopt a sound methodology. As virtual 3D visualization can directly show visually, acoustically and tactile a product in its later life phases along with its context; it's usage is deemed quite beneficial. Product's later life situations and product's interaction with actor(s) and environment can be evaluated virtually by means of product use-case scenarios [27]. Based on the research problem we formulate the following research questions.

2.1 Research Questions

2.1.1 Question One

How do we create a system for a Digital Twin in Unity3D that will facilitate the transfer of SysML models from the SysML modeler in the 3D environment and also provide a two-way live link between the SysML modeler and Unity3D for the exchange of data?

2.1.2 Question Two

Can such a system be scaled up; so that a developer can visualize larger models and send and receive more amounts of data via the live-link?

2.1.3 Question Three

Can the live-link be abstracted, such that it can be easily extended to other tools, for example Modelica and Simulink?

Based on the research questions we present a research plan. First, we study the possibilities to visualize the SysML models in the 3D virtual space, using Unity3D. Second, we look at the communication between the SysML modeler, IBM Rhapsody, and the Unity3D, prospecting the ways in which these two tools can communicate. Third, we determine how the SysML models can be embedded inside the Unity3D's 3D virtual space. Finally, we research how a live-link communication protocol can be implemented between the modeler and the 3D virtual space, respectively between IBM Rhapsody and Unity3D.

Chapter 3 - Related Work

In this chapter we present our research on Functional Mock-up Interface and Functional Mock-up Unit and how Prespective supports Functional Mock-up Unit using the PreLogic component. The Functional Mock-up Interface (FMI) [1] is a free standard that defines a container and an interface to exchange dynamic models using a combination of XML files, binaries and C code zipped into a single file. It is supported by over more than hundred tools and maintained as a Modelica Association Project on GitHub [1]. Whereas a Functional Mock-up Unit is an executable that can be run on a Functional Mock-up Interface. There are two functions for which a Functional Mock-up Interface can be used as given in Fig. 3.1.

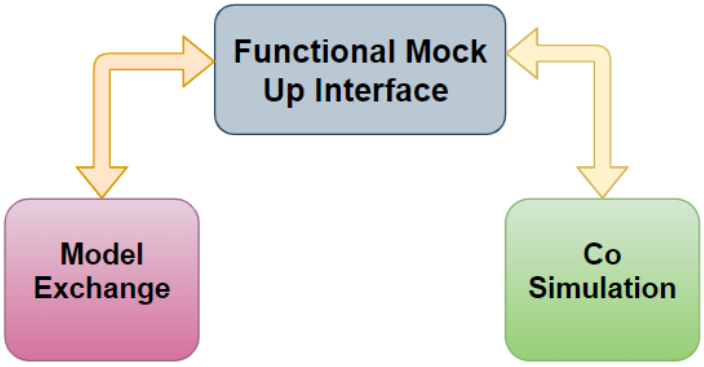

Figure 3.1: Functional Mock-up Interface consisting of two components

3.1 Functional Mock-up Interface

The Functional Mock-up Interface for model exchange defines an interface to the model of a dynamic system described by differential, algebraic and discrete time equations. The interface is provided to evaluate different equations in other simulation environments with explicit or implicit integrators, and fixed or variable step size [13]. Whereas the Functional

Mock-up Unit for Co-Simulation Interface is used to make different simulation tools work together coupled. The subsystem models that have been exported by their respective simulators with the solvers as runnable code can also be coupled [13]. The ulterior motive of a Functional Mock-up Interface for Co-Simulation is to find a solution of time dependent coupled systems consisting of subsystems that are continuous in time or are time discrete. When the coupled systems are represented as blocks, the subsystem are represented by blocks with state variables that are connected to other subsystems of the coupled problem by subsystem inputs and subsystem outputs.

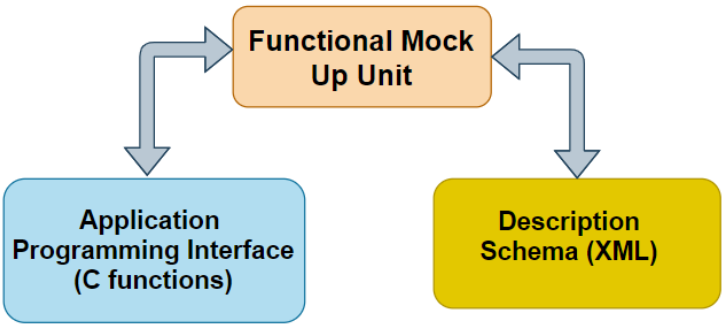

Figure 3.2: Functional Mock-up Unit is an integration of two individual parts

3.2 Functional Mock-up Unit

A Functional Mock-up Unit, as mentioned before is an executable unit run on a Functional Mock-up Interface. A Functional Mock-up Unit file consists of two parts that are enumerated in Fig. 3.2. The Functional Mock-up Interface API has all required equations or tool coupling computations are evaluated by calling standardized C language functions [13]. In this case C is used because it's the most portable programming language today and is the only programming language that is utilized in all embedded control systems. Whereas the Description Schema has an XML file contains the definition of all the variables of the FMU in a standardized way. It is possible to run the C code in an embedded system without the overhead of the variable definition [13]. Additional data in FMU specific file formats can also be added to a Functional Mock-up Unity.

Prespective supports Functional Mock-up Interface for Model Exchange where a Functional Mock-up Unit can be exported from a preferred modelling tool and the path of the Functional Mock-up Unit is added to Prespective. The solver for the Functional Mock-up Unit is present inside Prespective. When the Functional Mock-up Unit script is run the Functional Mock-up Unit starts running according to the behaviour specified by the model. The transformation results from the Functional Mock-up Unit are tied to the Unity3D

World Transformations with the help of visual descriptor files. When testing with the help of an Functional Mock-up Unit it is necessary to put all the three dimensional files of the machine or component inside the asset folder in Prespective. Subsequently we use Load the scene with Functional Mock-up Unit using the Functional Mock-up Unit visualizer tab under the PRElogic component and select the Functional Mock-up Unit. Then we instantiate an instance of the Functional Mock-up Unit and generate the game object after which inputs can then be given to the Functional Mock-up Unit [4].

Chapter 4 - Methodology

In this chapter we enumerate the individual steps that are necessary to research for our basic prototype. We then present our research findings for the individual steps and finally conclude the chapter by explaining how we implemented our prototype. The following steps are researched so that a basic prototype can be implemented after integration the individual steps:

- Find a way to extract data from SysML models inside IBM Rhapsody and represent the SysML diagrams in a browser that can be embedded in the 3D visualization environemnt.
- Find a command prompting mechanism inside IBM Rhapsody that can be controlled remotely and can be embedded inside Unity3D.
- Find a method to embed the command mechanism and the 3D visualization web pages generated from previous steps inside Unity3D using a plugin.
- Find a good alternative to set up a live-link between IBM Rhapsody and Unity3D so that data can be sent and received via a two-way interface.

After thorough research each individual step is implemented and integrated with each other to form our whole system. We research each step separately in this chapter and the results along with the interpretation are published and a conclusion is drawn. The choice made on how to test formal models and why visualization in 3D is better than in 2D is discussed in previous sections. Whereas the research and decisions made on each individual step listed above along with why a particular messaging protocol and a JavaScript Visualization library is presented in this chapters. Fig. 4.1 shows an overview of the research choices made along with a guideline on which areas to focus while conducting our project research. The figure shows our choice of MQTT for the two-way live link between IBM Rhapsody and Unity3D and Go.JS library for the visualization of Rhapsody model's XML data generated on the localhost.

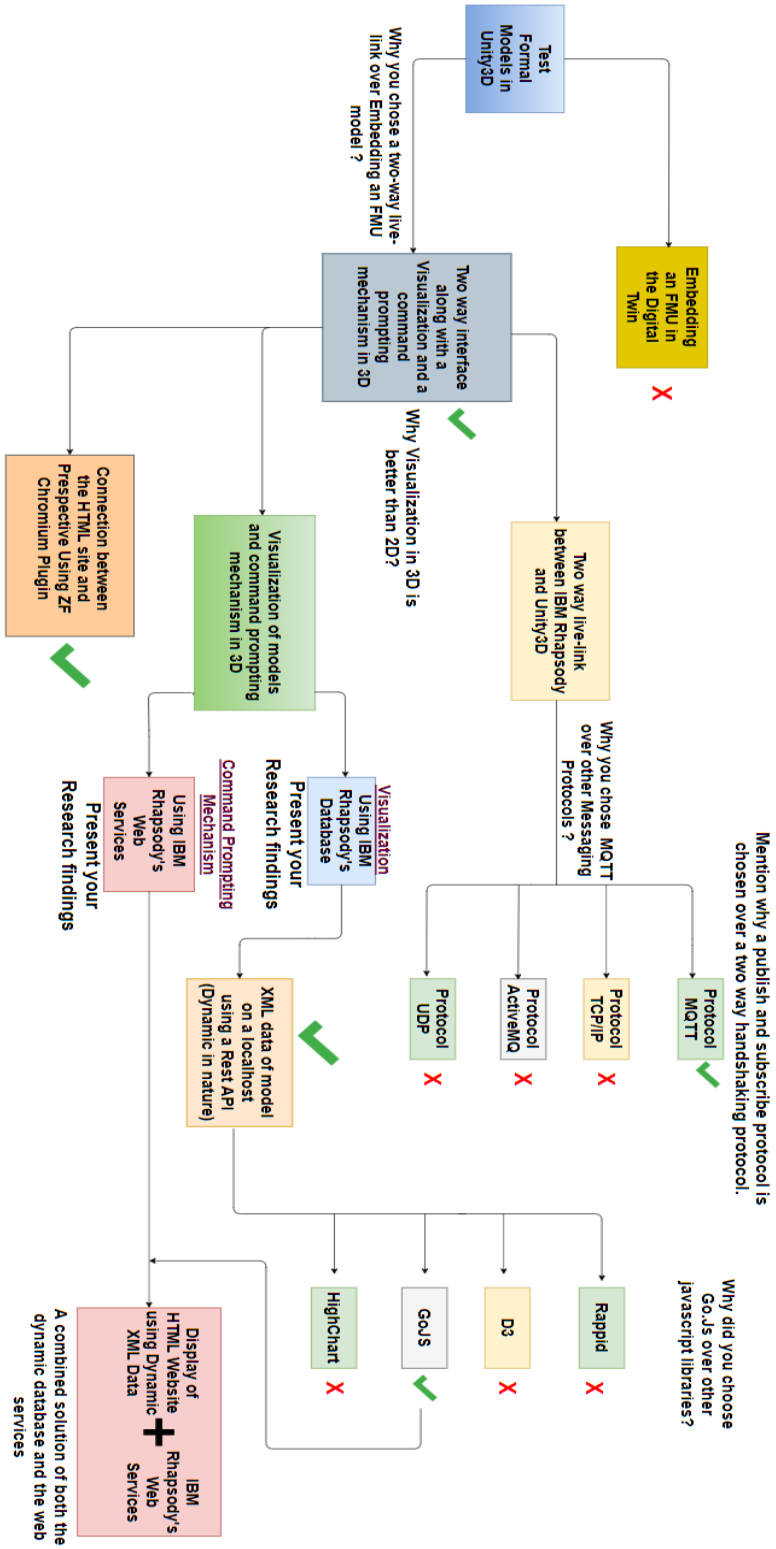

Figure 4.1: Research Plan Overview and choices made along with research areas to focus

4.1 Step 1 - Visualization of SysML Models

We research a method to visualize complex SysML diagrams inside Unity3D. This can be mainly done in two steps; firstly, we research on how to transfer the entire data of models from IBM Rhapsody to a web server and secondly, we research on how to parse the data from the web server and make visual models inside the 3D environment. An effective way to do the first step is to use IBM Rhapsody's Publishing Engine embedded inside Rhapsody. Rhapsody's Rational Publishing Engine is an eclipse based document producing and designing software. The entire process of generation of documents can be automated by exposing a Java API by Rhapsody Publishing Engine [24]. Rational Publishing Engine offers developers the option of publishing the entire data of a Rhapsody project on a localhost server created by Rhapsody in XML format. Rational Rhapsody uses a Rest service to publish the data by using a TCP (27643) port [23]. When the model data inside Rhapsody is published, a Web Server that can transfers the entire data on a localhost is generated as shown in the figure 4.2.

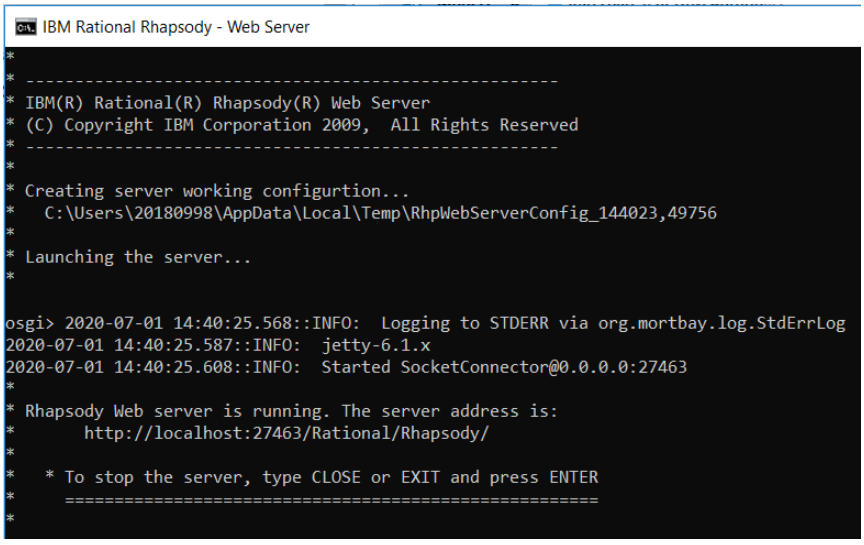

Figure 4.2: Web Server generated to publish model data from IBM Rhapsody on a localhost

The XML data that is published on the localhost is dynamic in nature. When changes are made inside IBM Rhapsody the changes are automatically reflected on the XML data [23]. This dynamic XML data can be coupled with JavaScript and HTML to make visual models in the 3D environment. The reason why we choose JavaScript and HTML is that both work efficiently with chromium web browsers [7] and hence the resulting HTML page can be integrated inside Unity3D using the ZF chromium plugin [7], which will be discussed later in the report. We use in our implementation JavaScript to parse the XML data of state machines, sub-state machines and the blocks in the system. For our research and implementation we limit to just one nesting level 4.3 for sub-state machines. The data that is parsed by JavaScript needs to be processed through a JavaScript library that can

recreate the state machines, sub-state machines and blocks visually in 3D. The HTML pages hence generated can then be integrated inside Unity3D using the ZF chromium plugin. An example snippet of the XML data generated from models in IBM Rhapsody, published the localhost is given in the figure 4.3.

```
<?rhapsody version="8.3.1" repositoryFormat="8.15.0" buildNo="9835550" ?>
▼<rhp:Projects xmlns:rhp="http://w3.ibm.com/xmlns/terms/Rhapsody" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
 ▼<rhp:Project href="http://localhost:27463/Rational/Rhapsody//TruckLab/GUID__7f61dcdd-23af-4028-bb3e-d002d06338c2">
    <rhp:iconFileName>C:\ProgramData\IBM\Rational\Rhapsody\8.3.1\Share\PredefinedPictures\Icons\RhapsodyIcons_0.gif</rhp:iconFileName>
    <rhp:id>GUID 7f61dcdd-23af-4028-bb3e-d002d06338c2</rhp:id>
    <rhp:isReadOnly>false</rhp:isReadOnly>
    <rhp:isSaveUnit>true</rhp:isSaveUnit>
    <rhp:isUnresolved>false</rhp:isUnresolved>
    <rhp:label>TruckLab</rhp:label>
    <rhp:metaclass>Project</rhp:metaclass>
    <rhp:name>TruckLab</rhp:name>
    <rhp:nestingLevel>1</rhp:nestingLevel>
    <rhp:nestingLevelAndFullPathName>1 </rhp:nestingLevelAndFullPathName>
    <rhp:requiremenTracabilityHandle>0</rhp:requiremenTracabilityHandle>
    <rhp:rmmServerID>0</rhp:rmmServerID>
    <rhp:stereotypes>SysML::.SysML (Stereotype)</rhp:stereotypes>
    <rhp:userDefinedMetaClass>SysML</rhp:userDefinedMetaClass>
   ▼<rhp:NestedElements>
```


Figure 4.3: Snippet of XML Data generated from models inside IBM Rhapsody

4.1.1 Visualization Tools

For visualization of the data we review the following state-of-the-art web technologies, third-party libraries and frameworks that can be utilized to fulfil the promise of interactive browser based custom visualization applications. These libraries use SVG [31] standard libraries for appending and manipulating SVG elements, which is supported in almost all modern browsers, smart phones and tablets. These libraries and frameworks are developed using pure JavaScript, so users get interactivity without requiring round-trips to servers and without any additional plugins [31]. We select the best option suited for our implementation. There are several visualization tools available in the market. However for our research we review the following four state-of-the-art JavaScript libraries and draw a conclusion on the best choice for our research and implementation:

HighChart

Highcharts is a pure JS charting library [31] [20], that offers a way of adding interactive web-based charts in 3D. It is designed from ground up with mobile browsers in mind, everything from multitouch zooming to touch-friendly tooltips responds great on mobile platforms [31]. Highchart can be employed to visualize engineering, scientific and experimental data seamlessly using interactive 2-D and 3-D line, spline, area, areaspline, column, bar, pie, scatter, angular gauges, arearange and many more interactive figures and features. Highcharts provide interactive plot enable/disable, zooming, panning and cross hairline support. Furthermore, developer have full control on colors, fonts, legends and axis labels.

GoJS

The second library we have is GoJS [16], another library that offers developers a chance to make interactive diagrams and visualizations in 3D. A simplified construction of diagrams can be made which include nodes, links and groups. In addition to visualization of data the library offers feature drag and drop of components and even handlers that make it a good choice for our implementation [31]. GoJS is implemented in TypeScript and can be used as a JavaScript library or built into your project from TypeScript sources. With GoJS a developer can build from custom modelling environments and domain-specific visual languages(SysML and UML) using it's powerful features. GoJS provides both a system editor and a read-only status monitor using shared code and templates. Simultaneously a developer can implement alternative visualizations of the same data in different diagrams.

D3

The third library we take a look at is D3.js [14], a JavaScript library for manipulating documents based on data and bring it to life using HTML, SVG and CSS. D3 emphasis on web standards gives developer the powerful visualization and provide a data-driven approach to Document Object Model manipulation. Built on top of jQuery D3 can be utilized to draw interactive diagrams and 2D/3D charts and plots.

Rappid

Finally, we look at Rappid [30], a commercial version to JointJS core and helps in extending functionality with interactive components and additional shapes. Rappid defines custom - and interactive - shapes using a combination of SVG and APIs that ease creation of these shapes. Using Rappid diagrams can be made interactive or not along with the option to export diagrams to JSON for easy front end and back end transfer plus storage [30]. The API of Rappid is easy to add, remove, update and connect ports to any shape. Also, the developers can resize and rotate the diagram's shapes on all sides and in any direction, including when rotated. Zoom, scroll and pan operations can be carried out on diagrams using animation transitions. Rappid offers a variety of ready to use set of built-in shapes such as rectangles, ovals, lines, ERD, State machines, Logic, ORG for most popular diagrams.

From our research we found that D3 and Highchart do not offer developers libraries that can be exclusively used to parse data and draw diagrams in SysML and UML. Therefore we conclude that GoJS and Rapid are the best options available for our implementation. Both provide an interactive way to visualize complex SysML diagrams and both require to convert the XML data from Rhapsody first into JSON, which will then be subsequently read by one of these libraries. Both libraries can be used to visualize blocks in the system. Rappid can be used to visualize state machines and sub-states in one diagram. However GoJS has the option of visualizing state machine and sub states machines in separate HTML tabs. For our implementation it is necessary to implement ease of visualization and

debugging for the developer and hence GoJS is a better choice over Rappid. In addition for better visualization GoJS has the option of animation along with state machines that show the passage of tokens when the system moves from one state to another. The overall architecture of the system along with the components of GoJS to be implemented is given in Fig. 4.4.

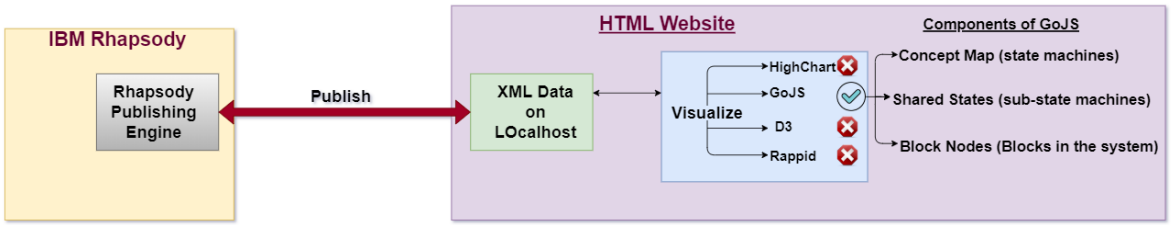

Figure 4.4: Publish and Visualize Data

4.2 Step 2 - Communication of IBM Rhapsody

The next step of our flow is to research a way to send commands to a Rhapsody executable inside a virtual 3D environment. Rhapsody's Web services is a good viable option according to our research. IBM Rhapsody's web services offers users the option of managing embedded software through the internet by setting certain components as web-enabled. This helps developers carry out maintenance process, monitoring and control of the embedded software remotely [21]. The web service acts as an interface for updating or changing the built software. A website is generated the moment web manageability is assigned to the elements in the model and the code is run. The graphical interface can be used to remotely control the device and it's performance. Developers can use their expertise and refine the appearance and capability of the device. In web management Rhapsody acts as the server and the web browser as the client. For this functionality to work the developer first needs to select which elements of the model do they want web enabled so that the code for the model is generated [21]. As the web browser can be used to invoke real-time events within the device, developers can make the web enabled browser an integral part of their development process. In addition the device resources are not overloaded with the capability to refresh continuously changed values [21]. The web service provides several web pages for the developer to navigate, manage and control the embedded software inside IBM Rhapsody as given in the Fig. 4.5 [21].

In addition, the web service provides a hierarchy of views that help in easy navigation. By selecting an aggregate in the left frame of this web service page, you can monitor and control your model in the aggregate table displayed in the right frame. Aggregate tables contain events and operations of a model model that are web enabled so that a developer can give commands remotely. To initialize and send events a developer can use the Activate

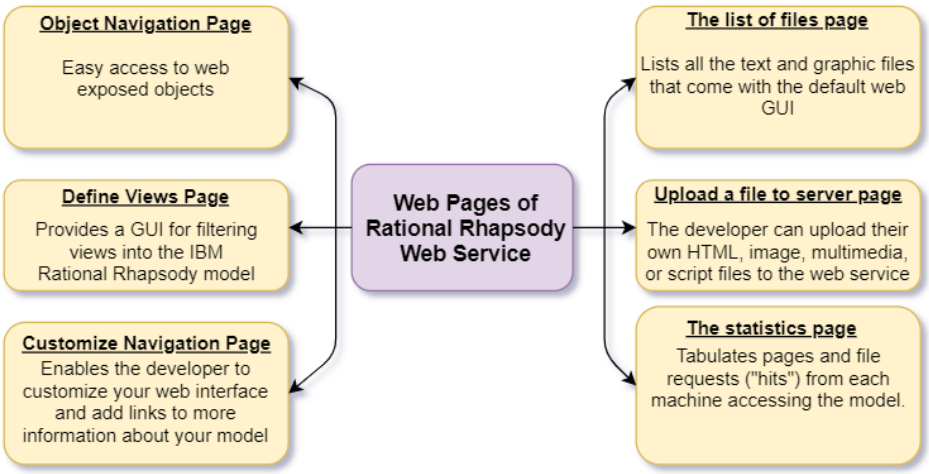

Figure 4.5: Web Pages provided by Rhapsody's Web Service to manage and control models

button besides the desired event. When the Rhapsody executable is running the events can be generated and sent via the web service and the models can be simulated [21]. The Fig. 4.6 shows an example of the right aggregate table that have operations, events and values of models inside IBM Rhapsody which can be activated, managed and controlled remotely.

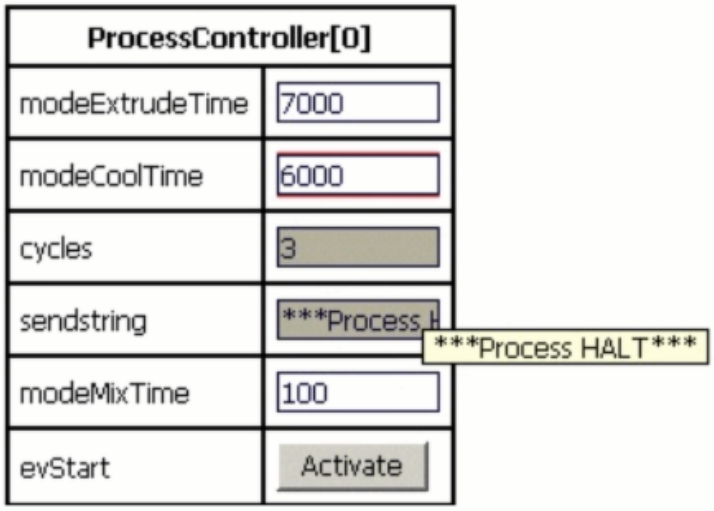

Figure 4.6: Aggregate Table used to send values and commands to Rhapsody remotely

We can similarly use the Rhapsody's web service to give web access to our own Rhapsody project so that we can send events remotely. The web service can then be accessed inside of Unity3D using the ZF chromium plugin. Hence, a user can send events and

simulate models in Rhapsody by giving commands inside a virtual 3D environment.

4.3 Step 3 - Embedding the Web Pages inside Unity3D

The third step is to embed all the web pages that are researched and implemented in the previous sections inside of Unity3D. An effective way to do so is to use the ZF browser plugin, that allows chromium based browsers like google chrome to open web pages as an object inside of Unity3D. The ZF library [7] is produced by a number of third-party open source software projects and the plugin is available on the asset store as a paid asset. The Unity Asset Store is home to a growing library of free and commercial assets created both by Unity Technologies and also members of the virtual twin community [6]. Browsers run JavaScript, specifically, ECMAScript whereas the components inside Unity runs .NET languages, in particular: C#. In addition to accessing web browsers users can embed web resources (HTML, JS, CSS, images, etc.) with your project and access them without the need for an external web server. All mouse/keyboard input events are collected in Unity3D and forwarded to the underlying browser and the developer can customize the input sent. If a project is running on Virtual 3D reality an input system is also included to get inputs from tracked controllers [7].

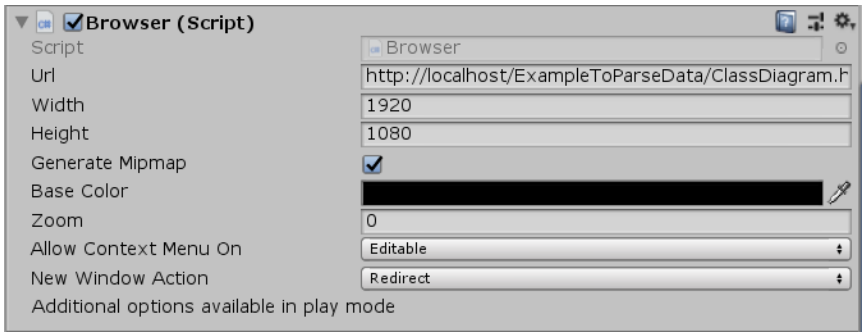

Figure 4.7: Script to input the URL, width, height and base color of the webpage

All the features of the plugin are varied and a developer can effectively use it to debug and test various parts of their project using the web browser. A developer can also access several web pages necessary for the integration of the project inside of Unity3D using this plugin. All activities such as inputs, mouse and VR controller clicks are sent asynchronously to a separate renderer process and the results are asynchronously reported back. As such, there is a minimal delay between when a command is issued and when the results are visible [7]. However for our research and implementation it serves perfectly well. Web browsers containing the state-machines, sub-state machines, blocks in a system and the IBM Rhapsody's web services can be accessed. Developers can zoom into various parts of the web browsers for better understanding of their code and also manage and send events to Rhapsody. Fig. 4.7 shows parameters such as the URL, width, height and base color that can be fitted inside the browser script to display the web page. The developer can

specify the URL along with the width and height of the pixels and the base color. The developer can also specify how much zoom they want in default.

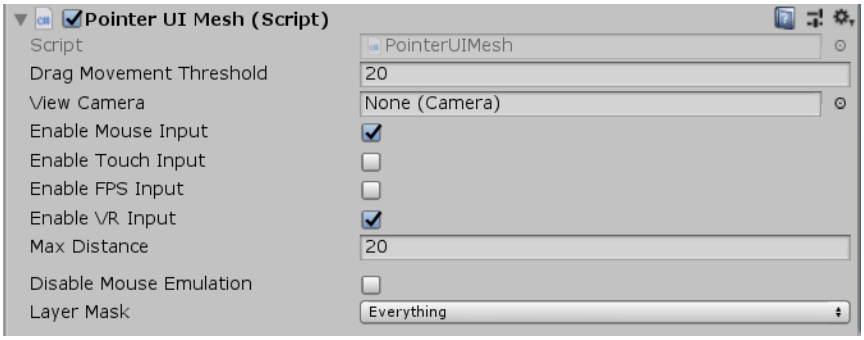

Figure 4.8: Script to enable various inputs and control the drag threshold on the web page

The User Interface mesh script shown in Fig. 4.8 can be used to enable various types of inputs and also control the drag movement threshold. The developer can enable the mouse input, touch input as well as VR input. For our implementation it's essential we have the VR input enabled so that developers and users can zoom in and zoom out of models and also send events using the Rhapsody web services. To embed the web pages inside Unity3D, the C# chromium plugin component scripts in that are added to a project. A developer can efficiently embed the state machines, sub-state machines, blocks and Rhapsody's web service inside Unity3D using this plugin.

4.4 Step 4 - Live-Link between IBM Rhapsody and Unity3D

The final step of this implementation is to research how a messaging protocol can be integrated with IBM Rhapsody and Unity3D so that a two way live-link is established between them. On Unity3D side we have Prespective plugged in that contains the Prelogic component. Prelogic helps a developer connect to a mechanical component or an assembly with control logic systems like several modelling tools. Prelogic is a predefined adapter source acting like a data connector that connect to a broker when the broker is active. Inside Prespective a developer can define network streams to send(publish) and receive(subscribe) data of various types with an active broker. The developer can select from a variety of protocols inside Prespective for establishing the connection. Hence, for our research we need to focus on IBM Rhapsody and consider four protocols for our implementation. Two of them have a publish/subscribe type architecture and the other two do not. We consider for our research four protocols namely MQTT, ActiveMQ, TCP/IP and UDP/IP. TCP/IP and UDP/IP and transport level protocol and the other two are sessions level protocol. MQTT and ActiveMQ can use a layer of TCP/IP in their implementation. Our main parameters for selecting a good protocol is that the protocol should be lightweight, easy to integrate with IBM Rhapsody and should allow scalability.

4.4.1 MQTT

The first protocol we take a look at is MQTT [33]; an internet of things connectivity protocol that is extremely lightweight and enables publish/subscribe messaging transport. It helps to use MQTT when we have remote locations where a small footprint is required and where network bandwidth is at a premium. MQTT is ideal for low scale applications as it's lightweight, utilizes low power and is small in size. This primarily explains the use of MQTT in sensors communicating to a broker with the help of satellite link, home automation and small device scenarios. In publish/subscribe architecture we have clients and a broker that the clients connect to. An MQTT broker is a server that receives all messages from the clients and then routes the messages to the appropriate destination clients [33]. An MQTT client is any device (from a micro controller up to a full-fledged server) that runs a MQTT library and connects to an MQTT broker over a network [8]. Information is organized in a hierarchy of topics. The MQTT protocol consists of two functions namely the publish and subscribe wherein the publisher publishes messages and a user subscribes to a topic, this is commonly known as the Publish/Subscribe model [26]. The user receives every message published by the publisher when it subscribes to a particular topic [32]. When a publisher has a new item of data to distribute, it sends a controlled message with the data to the connected broker. The broker then send this information to any of the clients that has subscribed to that topic. In an MQTT protocol the clients are unaware of each other as every client's interaction is with only the broker.

The broker discards messages that have a topic in which there are no current subscribers unless the message is designated as a retained message. A retained message has a retained flag that is set to true. Each client that subscribes to a topic pattern that matches the topic of the retained message receives the retained message immediately after they subscribe, the broker stores only one retained message per topic[9]. This allows new subscribers to a topic to receive the most current value rather than waiting for the next update form a publisher. MQTT offers a quality of service measurement that helps a developer define the priority of the message. The QOS values can range between 0 to 2 where for 0 the message is sent only and no acknowledgment, for 1 the message is sent as long as an acknowledgment is not received and in 2 both the sender and receiver engage in a two-level handshake [22]. The default unencrypted MQTT port is 1883 whereas the encrypted port is 8883.

4.4.2 ActiveMQ

The second protocol we look at is ActiveMQ [11]; an open project that can act as a full Java Messaging Service client. The communication is managed with features such as computer clustering and ability to use any database as a JMS persistence provider besides virtual memory, cache, and journal persistency [11]. The Java Messaging Service (JMS) is a communication middleware for distributed software components [19]. It is an elegant solution to make large software projects feasible and future-proof by a unified communication interface which is defined by the JMS API provided by Sun Microsystems [18]. A salient

feature of JMS is that applications can communicate with each other without knowing their communication partners as long as they agree on a uniform message format. Information providers publish messages to the JMS server and information consumers subscribe to certain message types at the JMS server to receive a certain subset of these messages [11]. This is known as the publish/subscribe principle.

ActiveMQ has a very high availability is efficient and easy to use however it leaves a relatively larger data footprint compared to MQTT which makes it a less viable option. For our implementation we require a protocol where the message size is small and protocol is lightweight and hence a better choice is MQTT among both. Though both have a publish/subscribe architecture MQTT offers varying lengths of quality of service that ActiveMQ does not. Hence, a developer has the freedom to choose the QOS of their messages based on the cruciality of their application and whether the application is real time or not. MQTT is the clear winner among the two when it comes to the implementation.

4.4.3 TCP/IP

The third protocol we research is TCP/IP; the most commonly used protocol suite or family containing many protocols offering computers different ways to communicate with each other. Though there are many protocols in this suite only a select few have gained widespread use. The TCP/IP protocol is the result of years of planned implementations and includes a set of standards on how computers and devices can and should communicate. The TCP/IP suite defines several conventions for the connection of various types of networks and how to route traffic through various routers and bridges. The source code for the implementation of TCP/IP is present in several public domains allowing networks to adapt to new systems. Today, essentially all network systems support the TCP/IP suite as it is the standard protocol suite on the Internet and in several companies.When you have data transferred from and to various external devices, the data is referred to in-terms of blocks. However, in case of TCP/IP the information unit is referred to as packets. The packets can have varying sizes, the sizes can sometimes be small enough so that it can be sent over a network fast enough. Each of the packets get a very small amount of time on the processor and the entire process happens so quickly that it's impossible to notice any faulty or corrupted data packet being transferred. When we look closely at the process of network communication we see several layers stacked on top of each other. Each layer in this stack supports the other stacks and in return is also supported by other stacks. The communication among the various layers of the protocol and their position with respect to each other can be seen graphically in Fig. 4.9.

When we look at the architecture at a closer level we realize that each layer is communicating independently with each other. TCP on remote machine communicates with TCP on local machine; passage of data is taking place between telnet on local machine and telnet on remote machine. IP on local machine gets information from IP on remote machine. In addition the network cards communicate among themselves with specific languages. The

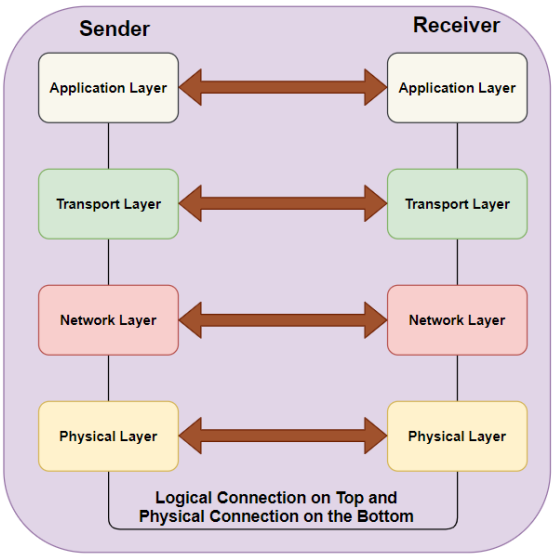

Figure 4.9: Communication between various layers of the TCP/IP Protocol

application layer uses a application header that contains the information essential to transfer the data to the right destination. When this is done the application layer calls the TCP to send information in the form of packets. A TCP packet contains a TCP header and the application data and a data packet contains the following parts shown in Fig. 4.10.

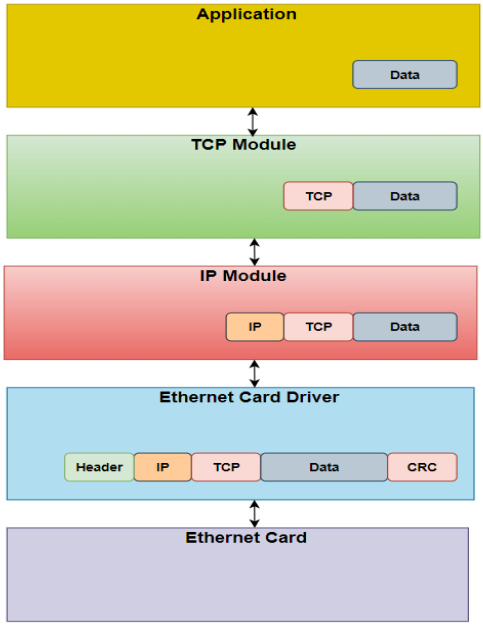

Figure 4.10: Components present inside a TCP/IP data packet

Once the packet is ready the TCP hands the packet to IP which wraps the packet up into a IP datagram which includes an IP header. Finally the IP gives the data off to the hardware driver. The communication among the various layers is conceptual and in reality the communication is between the different layers on each machine as compared to the concept that communication happens between corresponding layers of every machine. The application layer talks to the TCP layer which in turn converts the data from the application into a type of envelope. This process is called encapsulation. The data is passed on further to the IP layer. The network has a system in place to determine what data is connected to what process. The TCP also ensures that the packets are delivered with the correct contents and that they are put in the right order. There is a checksum of data that is contained within the packets that performs the job of detecting errors of the TCP envelope; this information is on the header. The checksum value should match the contents of the packets and this is the responsibility of the TCP to check. The TCP packets are resend until an acknowledgment is received withing a specific period of time. And the packet is resend if the acknowledgment is not received.

4.4.4 UDP/IP

The final protocol we look at is the User Datagram Protocol (UDP); a transport layer protocol that is defined for use with the Internet Network layer Protocol. The protocol in itself is quite an unreliable way of transmitting and receiving messages as there is no assurance that the message sent will be received or not and in addition the user cannot guarantee that a duplicate of their message is not created thus disrupting the connection. There is also no layer of security when using this protocol. However as UDP is simple in understanding and implementation it's overhead is reduced thus making in ideal for use in low resource applications. The components present inside a single UDP datagram along with their individual functions is given in Fig. 4.11. UDP is unique as compared to other communication protocols as UDP does not establish end-to-end connections among the various communicating systems. In addition UDP is minimal and quite unreliable as it does not incur connection establishments and tear-down overheads and hence there is a minimum associated end. The use and advantage of UDP entirely depends on the application for which it is being used. If the urgency of data being sent is not very high and loss of data can be overlooked like in video games then UDP can be effectively used. However, when you have a real time application UDP should be the last option as it lacks control over congestion of data and reliability.

When some of the applications use UDP the link interface's line rate is much higher than the capacity of the path which result in overcrowding of data on the stream and hence a lot of design considerations and knowledge of the application needs to be taken into consideration while designing such a system. Even though UDP is considered an unreliable protocol it does have some advantages as it's a tunelling protocol. Due to this feature tunnels help in establishing virtual link that appear to directly connect locations that are far away in the physical internet technology and hence can be used effectively to

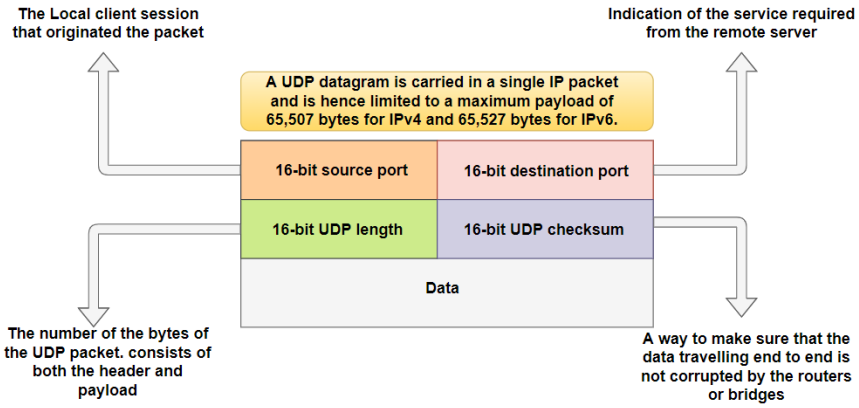

Figure 4.11: Components present inside a UDP datagram along with the functions

make virtual data networks. When it comes to protection from threats the user has to put in additional protection settings, services and protocols to avoid tampering and forgery. A connection between the sender and receiver is not essential for a UDP packet to be sent. When the data packets are large the fragmentation of IP packets is essential. However this in turn affects the reliability and efficiency of the sent data packet. The communication among the various layers of the UDP protocol and their position with respect to each other can be seen graphically in Fig. 4.12.

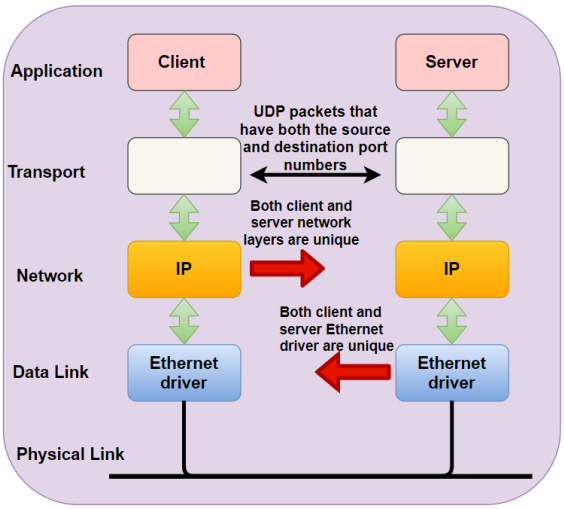

Figure 4.12: Communication between various layers of the UDP Protocol

With the help of the right port number a program listens for UDP packets and in return tells its local UDP layer to send packets matching the destination port number. It determines which client these packets come from by examining the received IP source

address. The source port number of the server is used to send any response that the server wants to send back to the client. If there are more than one IP interface on a host the execution should be such that the IP addresses of the source and destination match.

4.4.5 Comparison Study between TCP and UDP

The User Datagram Protocol and Transmission Control Protocol are transportation layer protocols that are considered the core protocols of the internet protocol suite [10]. Both the TCP and UDP protocols are approved to work on a transport layer of a network where the data is handled differently [12]. TCP uses a connection oriented way to handle the data thus providing a very reliable way of handling messaging and information where there is guarantee that the message will be delivered. In addition when there are errors in the transmission the packets will be resent over the network [17]. In contrast to TCP/IP, UDP uses a simpler transport model with a minimum of protocol technique. UDP uses datagrams to send and receive messages inside computer applications. A study and comparison between the two protocols reveals that TCP in some of the implemented mobility models outperforms the UDP in terms of thoroughput [25]. TCP is basically a connection-oriented protocol which provides end-to-end communication. The moment a connection is established between the transmitter and receiver, data can be sent. However in contrast UDP does not constitute a dedicated end-to-end communication among the transmitter and the receiver before there is real exchange of data. The data in case of UDP is sent over without the need to verify whether the data is received by the receiver [10]. Fig. 4.13 shows the segment fields of TCP and UDP that can be used to distinguish the size difference between the data packets.

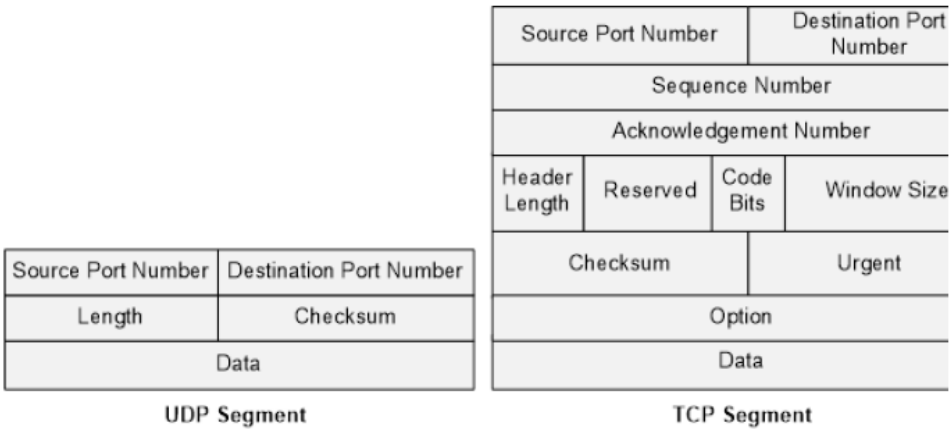

Figure 4.13: Segment Fields of TCP and UDP [10].

When it comes to basic operations and applications both the UDP and TCP communication protocols are similar. But the primary difference is in the data transmission wherein TCP offers a reliable data transfer and UDP does not as it's considered a connection less protocol. TCP uses re transmission of data and if there is some loss in data it's taken care

of by the acknowledgement message [10]. However UDP does not have re transmission, timeout or message acknowledgement. When it comes to the order in which messages are received TCP transmits messages in a particular order whereas UDP does not care much about the order in which messages are sent and received. The sequence of messages are not maintained over the transmission in UDP whereas in TCP the messages are reordered in case the packets of data reach in the wrong order. TCP sends data as segments and records them as a stream of bytes whereas in contrast UDP sends messages as datagrams in the network [10]. The comparison between TCP/IP and UDP based on resource usage, functionality, and data packets is given in table 4.1.

TCP/IP	**UDP**
(1) A track of lost data packs is kept and packets are resent	No track of lost packets is kept
(2) A sequence number of packets is kept and the packets are reordered when they arrive in the wrong order	UDP is not bothered about which order the packets arrive
(3) The additional functionalities make TCP slower	As UDP does not have extra features it is faster
(4) As the Operating system needs to keep track of ongoing communication sessions it requires more computer resources	The required computer resources are less.
(5) HTTP, HTTPS, FXP are examples of services using TCP	DNS, IP telehopny and DHCP are examples of services using UDP.

Table 4.1: Comparison between TCP and UDP [10].

For our implementation the order of the messages sent as well as the fact that messages sent should be received is important and hence TCP/IP is a more better choice as compared to UDP/IP.

We finally shortlish two protocols namely MQTT and TCP/IP. Both have great features and suite well our implementation. Both have a way to acknowledge whether a message is received or not and both take care of the order of the messages sent and received. However there is a fundamental difference in both the architectures. TCP/IP is a traditional client server architecture and MQTT is a publish/subscribe architecture. MQTT has a great advantage over TCP/IP in scalability. As clients using MQTT libraries are unaware of each other several instances of IBM Rhapsody can communicate with one instance of Unity3D and vice versa which is hard to implement in TCP/IP. Hence, our implementation can be scaled up by a large amount thus helping us test several parts of the digital twin with large models. Therefore, MQTT is a good choice for our implementation. MQTT can be build on top of TCP/IP which is an added advantage of using a sessions level MQTT protocol over a transport level TCP/IP protocol. In addition devices and applications are more

secure with MQTT as it used Transport Layer Security(TLS) encryption with user name and password protected connections along with optional certifications that requires client to provide a certificate file that matches with the server's. Fig. 4.14 has four clients; one can publish on the broker and three can subscribe from the same broker. This functionality can be reversed as well where n number of clients can publish and n number of clients can subscribe, all on the same message broker.

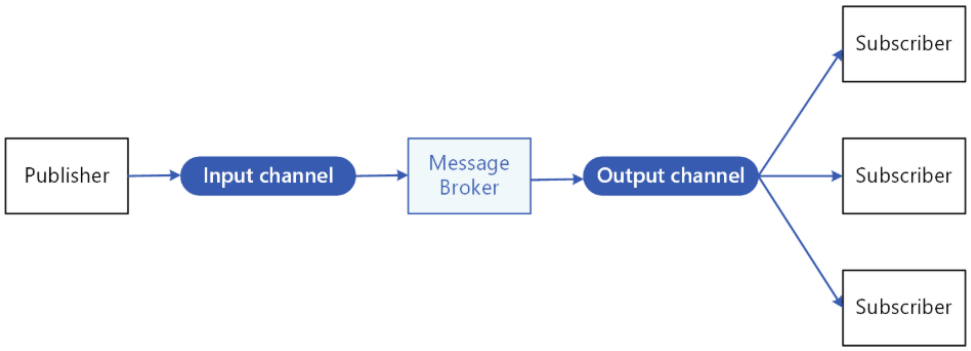

Figure 4.14: Pub/Sub Architecture where several clients communicate via one broker [28].

4.5 Implementation

In this subsection we present how the prototype was implemented. The architecture of the proposed prototype along with the logical flow of data and command is given in Fig. 4.15.

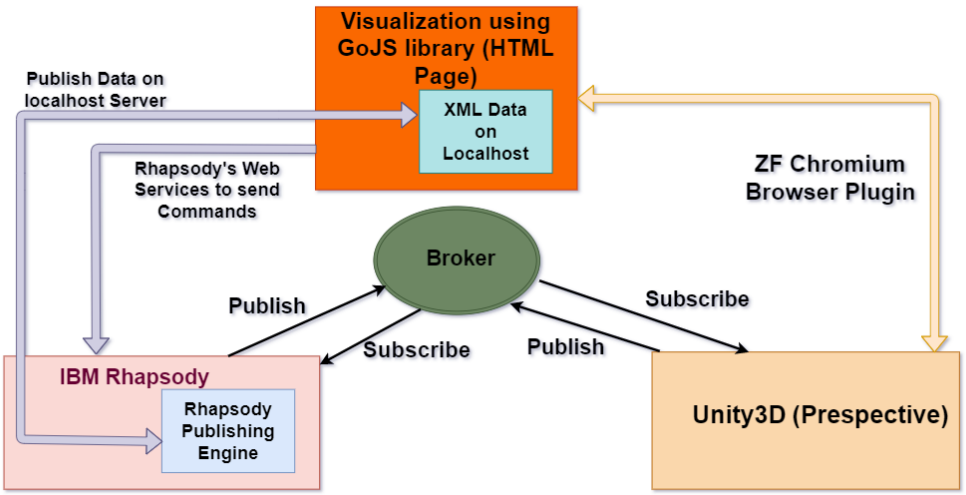

Figure 4.15: Architecture of prototype along with logical flow of data and command

First, we obtain the nodes and links from the parsed XML data generated by Rhapsody. This data is subsequently converted into JSON using JavaScript. This procedure is carried out for state machines, sub-state machines and block definition diagrams. For sub-state machines we have an extra property namely super state that needs to be filled with the parent state of the sub-states. We get the elements for our system by using get elements by tag name and get elements by ID functions in JavaScript. For the block diagram we put the names of all the blocks in a empty JSON object. The GoJS library is then used to draw all the tree types of model diagrams. The resulting web pages generated are included inside Unity3D with the *ZF* plugin using their respective Url's. Now these models can be visualized in 3D. Next we implement the web enabling option in our Rhapsody project and embed the localhost url inside Unity3D using *ZF* chromium plugin. The web page is only active when Rhapsody has an active executable running. Finally we work on the two way link. We use Eclipse Paho MQTT Open Source C code to build the necessary MQTT libraries. The libraries are then included inside Rhapsody Default Config folder. Due to this the MQTT functions defined inside these libraries can be effectively used inside Rhapsody to connect to an active broker to publish and subscribe to values. Unity3D already has the capability to connect to an active broker using the Prelogic adapter as described in previous sections.

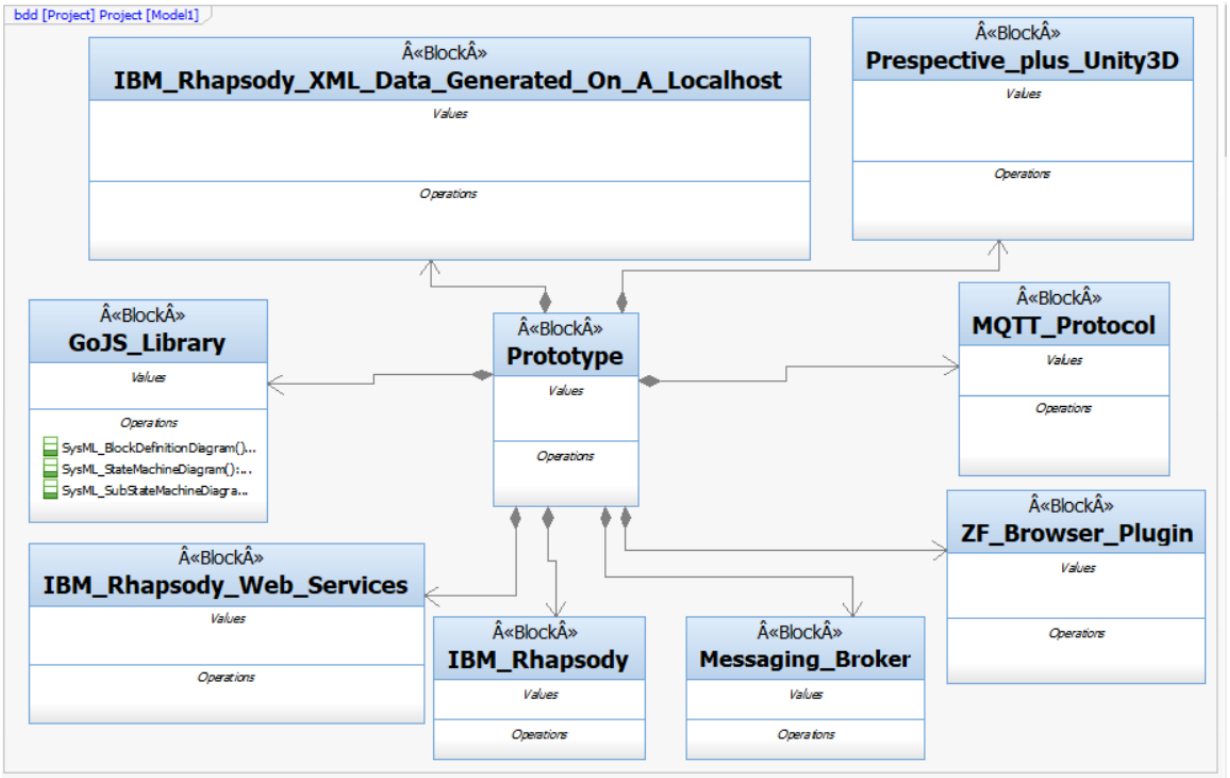

Figure 4.16: Components present inside the implemented prototype

Fig. 4.16 explains the components that are present in the prototype using a block

diagram. The Go.JS library visualizes all the SysML diagrams in our project scope. The ZF browser plugin embeds all the web browsers like the Rhapsody Web Service and the SysML diagrams inside of Unity3D+Prespective. The MQTT protocol is implemented inside IBM Rhapsody and is also a part of prespective in the PreLogic Component. Both Rhapsody and Unity3D can communicate with each other using the messaging broker. Our prototype is hence complete.

Chapter 5 - Discussion and Interpretation of Results

5.1 State and Sub-state Machines in the System

As enumerated in the project context, we test our prototype on a autonomous truck inside a virtual distribution center. The SysML state machine diagrams and block definition diagrams for this truck are created inside Rhapsody. Fig. 5.1 shows the SysML state machine diagram along with sub-states inside IBM Rhapsody that is to be tested. The testing of our prototype is carried out on this SysML model and the results with respect to the visualization of the model in Unity3D is presented.

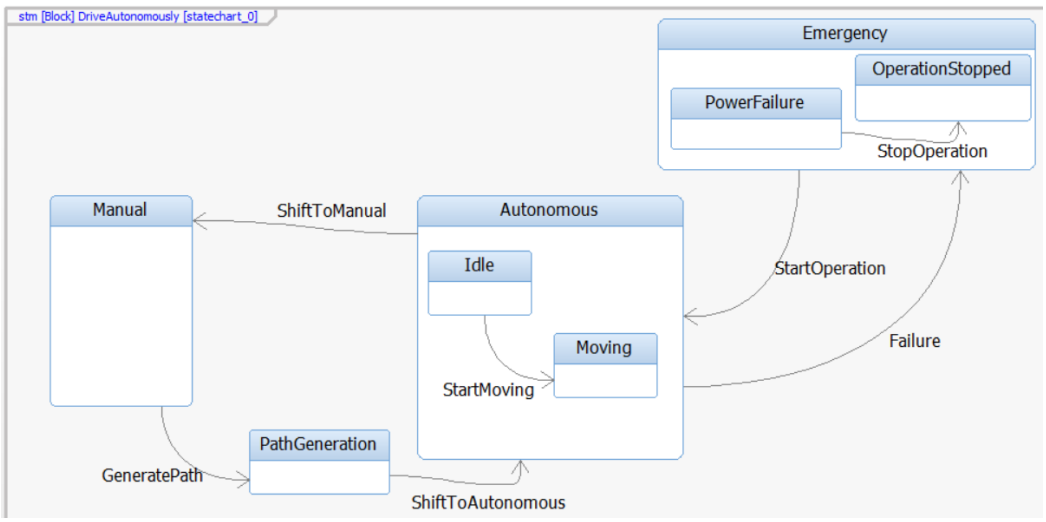

Figure 5.1: State Machine and Sub-state Machine in Rhapsody

Fig. 5.2 shows the same SysML state machine and sub-state machine diagrams integrated inside the 3D virtual environment. Hence, we can infer that the visualization of any SysML state machine diagram can be done effectively using the Rhapsody's localhost server. The models in 3D show the state and sub-state machines separately for better visualization. In addition the model created inside Unity3D changes when changes are

made inside Rhapsody validating our claim that the XML data generated is real time in nature.

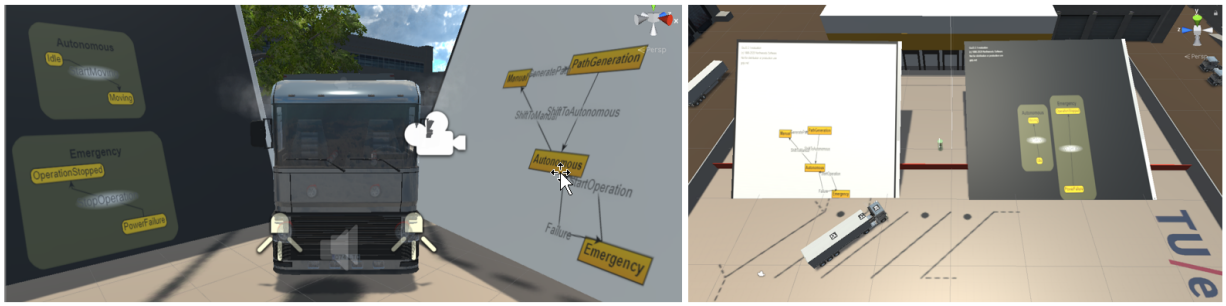

Figure 5.2: Truck Lab inside a Distribution Center with models

The same result is zoomed in Fig. 5.3 for better visualization. As can be seen the exact replica of the model in Rhapsody is created inside Unity3D for visualization. Hence, we can conclude that the web pages generated can be embedded effectively inside Unity3D

Figure 5.3: State and Sub-state Machine in Unity3D

5.2 SysML Block Definition Diagram

Fig. 5.4 shows the block definition diagram integrated inside the 3D virtual environment. Just like state machines in the previous section the data generating the blocks of the system inside Unity3D is dynamic in nature. Hence, changes made inside the block diagram in Rhapsody is visible on the web page integrated inside Unity3D. However we have a limitation wherein the implementation only shows the blocks and not the relations between

them. The structure of the XML data generated is such that there are limitations in reading the relation tags and making JavaScript objects out of it. Hence, we limit our research and implementation to only displaying the blocks of the BDD and not the relations between them.

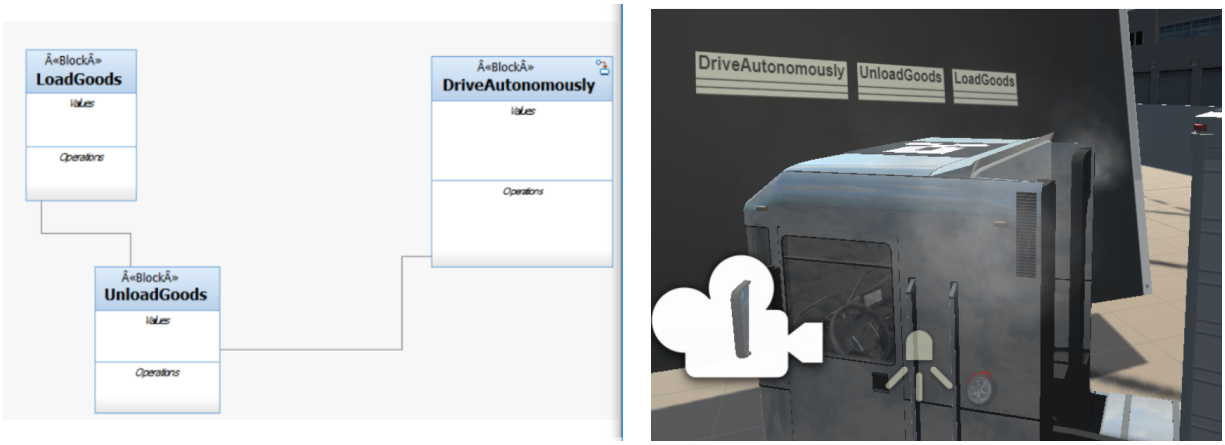

Figure 5.4: Block Definition Diagram containing the blocks of the System

5.3 Rhapsody's Web Services and Live-Link between IBM Rhapsody and Unity3D

Fig. 5.4 validates our research that the web services of Rhapsody can be embedded inside Unity3D.

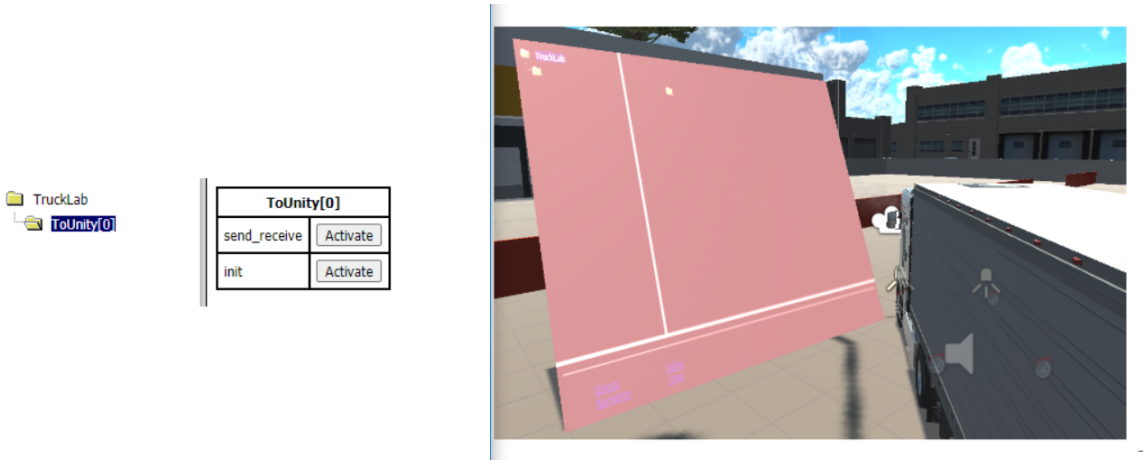

Figure 5.5: Rhapsody's Web Service embedded inside Unity3D

The web service can be used to send commands to IBM Rhapsody remotely which in turn sends data to Unity3D via a MQTT protocol as explained in the architecture of

the system diagram in the previous section. The MQTT protocol established between Rhapsody and Unity3D allows Rhapsody to publish speed and steering wheel values on the broker which in turn is subscribed by Unity3D. In turn Unity3D can publish values of the truck position on the broker that are subscribed by Rhapsody. The web service gives users the option to send commands to Rhapsody; as can be seen in the Fig. 5.5 the command init is used to create a MQTT client and connect to an active broker. When that happens the user can then press the send_receive activation to publish and subscribe to data between Rhapsody and Unity3D and hence setup the live-link. The functioning of these operations is given in Appendix D. Fig. 5.6 displays the values that are published and subscribed by Unity3D to the message broker. As can be seen the values are updated in real time thus helping a developer to carry out real time testing. The desktop input manager has an option pre-logic that can be selected to connect to an external broker.

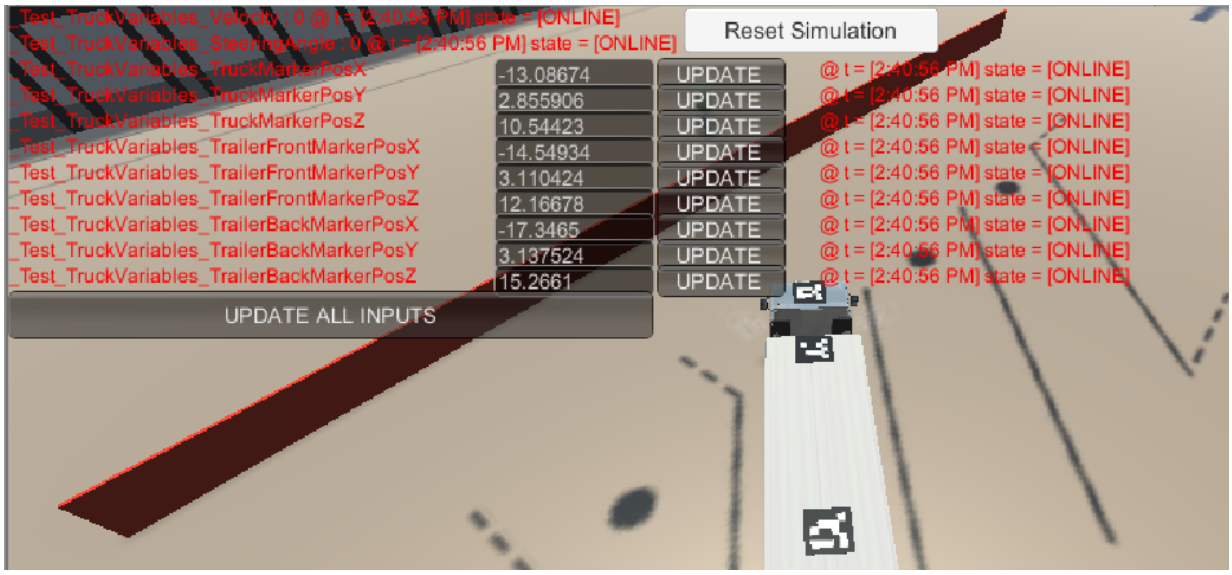

Figure 5.6: Values published and subscribed by Unity3D on the message broker

The results obtained through our detailed research and implementation validates our claim that such a system is possible to implement and test. A developer can introduce their models inside this virtual 3D environment for visualization as well as validate and test various aspect of their system sending commands and publishing and subscribing to data via a live link in real time thus reducing the time to market. Such a system can also be scaled up to larger models with several instances of Rhapsody and Unity3D communicating with each other. The results obtained till now give us a starting point to analyze such systems that help in virtual 3D testing and also open a possibility to make it more efficient. This validates another claim we made about extending such a functionality to other modelling tools. Any modelling tool that has a dynamic database can be read efficiently; the data can be parsed and the models can be remade inside Unity3D using a suitable visualization tool. In addition a MQTT Client can also be established between the modelling tool and Unity3D using suitable libraries.

Chapter 6 - Conclusions and Future Work

Based on the prototype, that can transfer data between a modelling tool and a virtual environment, we can proclaim that such a system can be created and used. By integrating IBM Rhapsody with Unity3D, we can integrate and visualize the SysML models in a 3D virtual environment. The implementation of the basic prototype and it's architecture is explained in section 4.5 and Fig. 4.15 respectively, thus answering our main research question. The prototype is tested against SysML models of a truck inside a distribution center and the results are published in chapter 5 thus validating our prototype. In section 4.4 we've enumerated how a publish/subscribe architecture for the two-way live-link between IBM Rhapsody and Unity3D can help scale up the prototype. Hence, this answers our second main research question that our prototype can be scaled up to include larger models with more amount of data. In addition, in section 4.1 the XML data geneated on the localhost by Rhapsody is dynamic in nature which helps us to implement a layer on top of it to parse the data and remake SysML diagram in the virtual environment. This method can be followed for other modelling tools like simulink and modelica provided that they have a dynamic database of their models that can be published on a localhost.

Just like IBM Rhapsody the data can be parsed for these tools as well and diagrams can be remade inside a virtual environment using a suitable JavaScript library. These modelling tools can also be integrated with a MQTT library so that they can connect to an external broker, for communication with Unity3D. This answers our third and final research question, that our research can be extended to other modelling tools like simulink and modelica. Several Rhapsody and Unity3D clients can exchanging data over one single central broker. A developer is not affected due to limitations of 2D modeling to comprehend the system's complexity. Large and complex models can be visualized, tested, debugged, validated and executed in a 3D virtual environment. As part of our future work research can be done on how to make the Rest API of IBM Rhapsody more efficient so that animations inside IBM Rhapsody can be tied up with token visualization of 3D SysML models inside Unity3D. This will help developers in visualizing the animations also in a 3D environment and make testing and debugging more efficient. In addition visualization of other behavioural SysML diagrams such as activity diagrams and sequence diagrams can

also be included in the scope for future work. Also research can be conducted on how to bind the relations that various blocks of a SysML block definition diagram have with each other inside Unity3D. The prototype that was implemented through this project opens several research areas that can be focused on to make an even more efficient prototype with larger functionalities.

Bibliography

[1] Functional mock-up interface homepage, https://fmi-standard.org/. 2, 7

[2] Ibm rhapsody webpage https://www.ibm.com/products/systems-design-rhapsody. 2

[3] Sysml homepage https://sysml.org/. 2

[4] Unit040: Prespective documentation 2020. 3, 9

[5] Unity webpage : https://unity.com/ 2020. 3

[6] Unity3d documentation https://docs.unity3d.com/2018.3/Documentation/Manual/AssetStore, author : Unity. 18

[7] Zf browser documentation https://zenfulcrum.com/docs/browser/Readme.html, author : Zf browser. 13, 18

[8] Part 3".hivemq.com. Client, Broker / Server and Connection Establishment - MQTT Essentials, 2019. 20

[9] Part 8".hivemq.com. Retained Messages - MQTT Essentials, 2019. 20

[10] Fahad Taha AL-Dhief, Naseer Sabri, NM Abdul Latiff, Nik Noordini Nik Abd Malik, Musatafa Abbas, Abbood Albader, Mazin Abed Mohammed, Rami Noori AL-Haddad, Yasir Dawood Salman, Mohd Khanapi, et al. Performance comparison between tcp and udp protocols in different simulation scenarios. *International Journal of Engineering & Technology*, 7(4.36):172–176, 2018. ix, x, 25, 26

[11] Apache. Activemq features http://activemq.apache.org/. 20, 21

[12] Purnya Awasthi and Akhilesh Kosta. Comparative study and simulation of tcp and udp traffic over hybrid network with mobile ip. *International Journal of Computer Applications*, 83(13), 2013. 25

[13] Torsten Blochwitz, Martin Otter, Martin Arnold, Constanze Bausch, Christoph Clauss, Hilding Elmqvist, Andreas Junghanns, Jakob Mauss, Manuel Monteiro,

Thomas Neidhold, et al. The functional mockup interface for tool independent exchange of simulation models. In *Proceedings of the 8th International Modelica Conference*, pages 105–114. Linköping University Press, 2011. 7, 8

[14] Michael Bostock, Vadim Ogievetsky, and Jeffrey Heer. D^3 data-driven documents. *IEEE transactions on visualization and computer graphics*, 17(12):2301–2309, 2011. 15

[15] Yanja Dajsuren and Mark van den Brand. *Automotive Systems and Software Engineering*. Springer, 2019. 1

[16] GoJS. Interactive diagrams for javascript and html. 2016. https://www.nwoods.com/products/gojs/. 15

[17] Bhargavi H Goswami. Experimental based performance testing of different tcp protocol variants in comparison of rcp+ over hybrid network scenario. *International Journal of Innovations & Advancement in Computer Science IJIACS ISSN*, pages 2347–8616, 2014. 25

[18] Kim Haase. Java message service api tutorial. *Sun Microsystems, Inc*, pages 1–278, 2002. 20

[19] Robert Henjes, Daniel Schlosser, Michael Menth, and Valentin Himmler. Throughput performance of the activemq jms server. In *Kommunikation in Verteilten Systemen (KiVS)*, pages 113–124. Springer, 2007. 20

[20] Highcharts. Høns, t.. highcharts, highstock and highmaps documentation http://www.highcharts.com/docs, 2013. 14

[21] IBM. Ibm rhapsody managing web-enabled devices https://www.ibm.com/support/knowledgecenter/SSB2MU_8.4.0/com.ibm.rhp.animation.doc/topics/rhp_t_dm_mnging_web_dvcs.html.. 16, 17

[22] IBM. Knowledge center https://www.ibm.com/support/knowledgecenter.SSGU8G12. 20

[23] IBM. Rational publishing engine user guide release 1.1. 13

[24] IBM. Rational publishing engine, Retrieved February 13, 2014. 13

[25] Santosh Kumar and Sonam Rai. Survey on transport layer protocols: Tcp & udp. *International Journal of Computer Applications*, 46(7):20–25, 2012. 25

[26] Valerie Lampkin, Weng Tat Leong, Leonardo Olivera, Sweta Rawat, Nagesh Subrahmanyam, Rong Xiang, Gerald Kallas, Neeraj Krishna, Stefan Fassmann, Martin Keen, et al. *Building smarter planet solutions with mqtt and ibm websphere mq telemetry*. IBM Redbooks, 2012. 20

[27] Atif Mahboob, Stephan Husung, Christian Weber, Andreas Liebal, Heidi Krömker, et al. Sysml behaviour models for description of virtual reality environments for early evaluation of a product. In *DS 92: Proceedings of the DESIGN 2018 15th International Design Conference*, pages 2903–2912, 2018. 5

[28] Microsoft. Azure `https://docs.microsoft.com/en-us/azure/architecture/patterns/publisher-subscriber`. ix, 27

[29] Parastoo Mohagheghi, Wasif Gilani, Alin Stefanescu, Miguel A Fernandez, Bjørn Nordmoen, and Mathias Fritzsche. Where does model-driven engineering help? experiences from three industrial cases. *Software & Systems Modeling*, 12(3):619–639, 2013. 1

[30] Rappid. Diagramming framework for advanced applications `https://www.jointjs.com`. 15

[31] Farrukh Shahzad, Tarek R Sheltami, Elhadi M Shakshuki, and Omar Shaikh. A review of latest web tools and libraries for state-of-the-art visualization. *Procedia Computer Science*, 98:100–106, 2016. 14, 15

[32] Dipa Soni and Ashwin Makwana. A survey on mqtt: a protocol of internet of things (iot). In *International Conference On Telecommunication, Power Analysis And Computing Techniques (ICTPACT-2017)*, 2017. 20

[33] Michael Yuan. Getting to know mqtt. 13th october 2019. 20

State Machine (Appendix A)

The first step of my thesis is to use the Go.js library to write my own javascript code so that it can read XML data that is produced by IBM Rhapsody. The XML data is published on a localhost server created by IBM Rhapsody. The data that is read has to be put in well defined JSON(JavaScript Object Notation) format. The advantage of this mechanism is that every state machine inside an IBM Project has the same format for the XML produced. Hence the code written can be effectively used for every state machine inside any IBM Rhapsody project.

We describe the step by step method followed to read the XML data along with the method the Go.Js library used to convert the JSON objects into a state machine.

1) The JSON format consists of two arrays of objects namely the nodeDataArray and the linkDataArray. The node data array consists of names of all the states along with an identifier assigned to each of them. Whereas the link data array consists of all the transitions between the various states along with the names of the transitions.

2) We define an empty array of nodeData and linkData; to be filled with data from the XML that is turned into objects.

3) We first get the states tag inside the XML using the 'getElementsByTagNameNS' command. If it is undefined that means there are no states in the respective XML.

4) For each of the individual state inside the states tag we check their Parent state tag; if and only if it's equal to "ROOT" we proceed to get the name of that particular state. Else we omit that state.

5) The names of all the states obtained in the meta-state are assigned to a particular object along with an identifier and then the objects are put in the nodeData Array element wise.

6) Now we form the transitions array by first getting all the states in the metamodel which have their parent state as the 'ROOT' state as done in the antecedents steps.

7) Then for each of the state we find all the outgoing edges and get their names. This is done by using the 'getElementsByTagName' and 'getElementsByTagNameNS' functions. This forms the outer loop

8) The inner loop consists of getting all the incoming edges for each of the states. This process is repeated for each outgoing edge of each state as explained in the following steps.

9) Now we have the outgoing edges for each state and also the incoming edges for each state. The j-loop loops through each outgoing edge of each state. When it reaches a particular outgoing transition of a particular state the code runs another inner loop(the k-loop and l-loop) that checks all the incoming transitions of each of the state. We then check if the outgoing transition that we've got from the outer loop is equal to the incoming transition we've got from the inner loop. If it is; we output a statement that XYZ is a transition from state A(outer loop) to state B(inner loop).

10) Then we check the transitions tag inside the XML; we check the label of the edge by using a for loop(w-loop). When the loop sees that a particular transition inside the transitions tag is equal to the edge we find out the label for that transition.

11) We finally form objects of all the transitions and start feeding it element wise to the 'linkDataArray' which will later be read by the Go.js library.

The next step is that the node and link data is read appropriately by the library:

1) The library consists of templates defined for nodes and links. The nodes have their size and shape defined along with the type and format of the text inside the nodes. The link have properties like the arrowhead, the label background, the label text which defines the font as well as the text alignment.

2) The library also defines a custom layout of the state machine along with a custom dragging tool that will help in zooming in and out of the state machine so that only certain parts of the state machine are visible.

3) The library also has a save and load method that helps in saving the model and then loading it in the library using the my diagram element.

```
<html>
<head>
  <meta charset="UTF-8">
  <title>State Chart</title>
  <meta name="description" content="A finite state machine chart with
      editable and interactive features." />
  <meta name="viewport" content="width=device-width, initial-scale=1">
  <!-- Copyright 1998-2020 by Northwoods Software Corporation. -->
```

```
<main>

  <ul id="nested-elements"></ul>
  <ul id="States"></ul>
  <ul id="Transitions"></ul>
  <ul id="incoming-edges"></ul>
  <pre id="output"></pre>
</main>
  <script src="jquery-3.4.1.js"></script>
  <script src="go.js"></script>
     <script src="require.js"></script>
<script>
 var s;
document.addEventListener('DOMContentLoaded', ()=>{

let url="Paper.xml";
fetch(url)
.then(response=>response.text())
.then(data=>{
  let parser = new DOMParser();
  let xml = parser.parseFromString(data, "application/xml");
  var identifier ={};
  //document.getElementById('output').textContent=data;
  console.log(xml);
  //buildStatesList(xml);
  //buildTransitionsList(xml);
  buildTransitionNamesList(xml);
  //buildTransitionFromToList(xml);
  BuildJsonFile(xml);
  //BuildJsonFile2(xml);

})
})

function buildTransitionFromToList(xml){
  let list = document.getElementById("States");
  let StatesTag = xml.getElementsByTagNameNS("http://w3.ibm.com/xmlns/terms/
     Rhapsody","States");
// For each of the state we first find the outgoing transition and then we
   find out for which of the states that transition is an incoming
   transition. Hence we then club up the information to display as one
   statement.
  if(StatesTag[0]!==undefined)
  {

    let ChildOfStatesTag = StatesTag[0].getElementsByTagNameNS("http://w3.
       ibm.com/xmlns/terms/Rhapsody","State");

    for(let i=0; i<ChildOfStatesTag.length; i++)
    {
```

```
let states=ChildOfStatesTag[i].getElementsByTagName('rhp:name');
let OutgoingEdges = ChildOfStatesTag[i].getElementsByTagNameNS("http:
    //w3.ibm.com/xmlns/terms/Rhapsody","OutgoingEdges");

 if(OutgoingEdges[0]!==undefined)
 {
   let OutgoingTransitionsOfAState=OutgoingEdges[0].
       getElementsByTagName('rhp:Transition');
   for (let j=0; j<OutgoingTransitionsOfAState.length; j++)
   {
     let li = document.createElement('li');
     let state = states[0].firstChild.nodeValue;
     let NameOfOutgoingEdge = OutgoingTransitionsOfAState[j].
         getElementsByTagName('rhp:name');
     let OutgoingEdgeOfState = NameOfOutgoingEdge[0].firstChild.
         nodeValue;

     let StatesTagI = xml.getElementsByTagNameNS("http://w3.ibm.com/
         xmlns/terms/Rhapsody","States");

      if(StatesTagI[0]!==undefined)
      {
        let ChildOfStatesTagI = StatesTagI[0].getElementsByTagNameNS(
            "http://w3.ibm.com/xmlns/terms/Rhapsody","State");
        for(let i=0; i<ChildOfStatesTagI.length; i++)
         {
            statesI=ChildOfStatesTagI[i].getElementsByTagName('
                rhp:name');
            let IncomingEdges = ChildOfStatesTagI[i].
                getElementsByTagNameNS("http://w3.ibm.com/xmlns/terms/
                Rhapsody","IncomingEdges");
              if(IncomingEdges[0]!==undefined)
              {
                 let IncomingTransitionsOfAState=IncomingEdges[0].
                     getElementsByTagName('rhp:Transition');
                 for (let j=0; j<IncomingTransitionsOfAState.length;
                     j++)
                  {
                     let li = document.createElement('li');
                     let stateI = statesI[0].firstChild.nodeValue;
                     let NameOfIncomingEdge =
                         IncomingTransitionsOfAState[j].
                         getElementsByTagName('rhp:name');
                     let IncomingEdgeOfState = NameOfIncomingEdge[0].
                         firstChild.nodeValue;
                     if(IncomingEdgeOfState===OutgoingEdgeOfState)
                     {
                        li.textContent= IncomingEdgeOfState + '  - is a
                            transition from   ' + state + '  to   ' +
                           stateI;
                        //list.appendChild(li);
```

```
                        }
                    }
                }
            }
        }
    }
  }
}

function buildStatesList(xml){
  let list = document.getElementById("States");
  let statesTag = xml.getElementsByTagNameNS("http://w3.ibm.com/xmlns/terms/
      Rhapsody","States");

  if(statesTag[0]!==undefined)

  {

  let ChildOfstatesTag = statesTag[0].getElementsByTagNameNS("http://w3.ibm.
      com/xmlns/terms/Rhapsody","State");

    for(let i=0; i<ChildOfstatesTag.length; i++)
    {
      States=ChildOfstatesTag[i].getElementsByTagName('rhp:name');
      let li = document.createElement('li');
      let State = States[0].firstChild.nodeValue;
      li.textContent = State;
      list.appendChild(li);
       obj = {"id" : i, "text" : State};
       console.log(obj);

        var dict = new Object()

        dict={"EngineFail": i};

    }

  }
    else
  {
      let li = document.createElement('li');
      li.textContent= 'This XML has no states';
      list.appendChild(li);
  }

    console.log(obj);

}
```

```
function buildTransitionsList(xml){
  let list = document.getElementById("States");
  let StatesTag = xml.getElementsByTagNameNS("http://w3.ibm.com/xmlns/terms/
     Rhapsody","States");

  if(StatesTag[0]!==undefined)

  {

  let ChildOfStatesTag = StatesTag[0].getElementsByTagNameNS("http://w3.ibm.
     com/xmlns/terms/Rhapsody","State");

//TO CHECK ALL THE INCOMING EDGES-------------------------------------------
   ---------
    for(let i=0; i<ChildOfStatesTag.length; i++)
   {

      states=ChildOfStatesTag[i].getElementsByTagName('rhp:name');
      let IncomingEdges = ChildOfStatesTag[i].getElementsByTagNameNS("http:
         //w3.ibm.com/xmlns/terms/Rhapsody","IncomingEdges");

      if(IncomingEdges[0]!==undefined)
      {
       let IncomingTransitionsOfAState=IncomingEdges[0].getElementsByTagName
          ('rhp:Transition');
       for (let j=0; j<IncomingTransitionsOfAState.length; j++)
         {
            let li = document.createElement('li');
            let state = states[0].firstChild.nodeValue;
            let NameOfIncomingEdge = IncomingTransitionsOfAState[j].
               getElementsByTagName('rhp:name');
            let IncomingEdgeOfState = NameOfIncomingEdge[0].firstChild.
               nodeValue;
            li.textContent= IncomingEdgeOfState + ' - is incoming edge of -
                ' + state;
            list.appendChild(li);
         }

      }
      else
         {
            let li = document.createElement('li');
            let state = states[0].firstChild.nodeValue;
            li.textContent= state + ' - has no incoming edge';
            list.appendChild(li);
         }

   }

//TO CHECK ALL TH OUTGOING EDGES--------------------------------------------
   ------
```

```
for(let i=0; i<ChildOfStatesTag.length; i++)
      {
       states=ChildOfStatesTag[i].getElementsByTagName('rhp:name');
       let OutgoingEdges = ChildOfStatesTag[i].getElementsByTagNameNS("http:
          //w3.ibm.com/xmlns/terms/Rhapsody","OutgoingEdges");

       if(OutgoingEdges[0]!==undefined)
       {
        let OutgoingTransitionsOfAState=OutgoingEdges[0].getElementsByTagName
           ('rhp:Transition');
        for (let j=0; j<OutgoingTransitionsOfAState.length; j++)
          {
              let li = document.createElement('li');
              let state = states[0].firstChild.nodeValue;
              let NameOfOutgoingEdge = OutgoingTransitionsOfAState[j].
                 getElementsByTagName('rhp:name');
              let  OutgoingEdgeOfState = NameOfOutgoingEdge[0].firstChild.
                 nodeValue;
              li.textContent=state + ' - has outgoing edge - ' +
                 OutgoingEdgeOfState ;
              list.appendChild(li);

          }

       }
     else
         {
              let li = document.createElement('li');
              let state = states[0].firstChild.nodeValue;
              li.textContent= state + ' - has no outgoing edge';
              list.appendChild(li);
         }

      }

  }

}

function buildTransitionNamesList(xml){

   let list = document.getElementById("Transitions");
   let TransitionsTag = xml.getElementsByTagNameNS("http://w3.ibm.com/xmlns/
      terms/Rhapsody","Transitions");

   if(TransitionsTag[0]!==undefined)
   {

   let ChildOfTransitionsTag = TransitionsTag[0].getElementsByTagNameNS("
      http://w3.ibm.com/xmlns/terms/Rhapsody","Transition");
```

```
  for(let i=0; i<ChildOfTransitionsTag.length; i++)
       {
            let li = document.createElement('li');
            transitions=ChildOfTransitionsTag[i].getElementsByTagName('
               rhp:name');
            labels=ChildOfTransitionsTag[i].getElementsByTagName('rhp:label')
               ;

            if(labels[0]!==undefined)
           {


             let label=labels[0].firstChild.nodeValue;
             let transition = transitions[0].firstChild.nodeValue;
             li.textContent= transition + ' - the label for this transition
                is ' + label;
             //list.appendChild(li);  -> UNCOMMENT TO SEE THE TRANSITION

           }
            else
           {
             let transition = transitions[0].firstChild.nodeValue;
             li.textContent= transition + ' - the transition has no label ';
             //list.appendChild(li);  -> UNCOMMENT TO SEE THE TRANSITION
           }

       }
   }
     else
       {
            let li = document.createElement('li');
            //let state = states[0].firstChild.nodeValue;
            li.textContent= 'This XML has no Transitions';
            //list.appendChild(li);  -> UNCOMMENT TO SEE THE TRANSITION
       }

}


// Defining global list of states along with global node and link data
   arrays so that these arrays can be used by the library.
let list = document.getElementById("States");
let nodeDataArray= [];
let linkDataArray= [];
let  o=0;

function BuildJsonFile(xml){
  // get all the states tag in the XML file.

  let statesTag = xml.getElementsByTagNameNS("http://w3.ibm.com/xmlns/terms/
     Rhapsody","States");
```

```
let q=0;

// a statement to check if there is a states tag inside the XML. If not
   the XML is not of a state machine.
if(statesTag[0]!==undefined)   {

    //Get each of the individual states under the states tag.
    let ChildOfstatesTag = statesTag[0].getElementsByTagNameNS("http://w3.
       ibm.com/xmlns/terms/Rhapsody","State");

    // For each of the individual states we check their Parent state tag;
       if and only if it's equal to "ROOT" we proceed to get the name of
       that particualar state. Else we omit that state.
    //We also fill the 'nodeDataArray' with objects so that at the end of
       the loop the 'nodeDataArray' contains all the names of the states
       along with a key or an identifier.
    for(let i=0; i<ChildOfstatesTag.length; i++)
    {

     var dict = new Object();

     let ParentStates= ChildOfstatesTag[i].getElementsByTagNameNS("http://
        w3.ibm.com/xmlns/terms/Rhapsody","ItsParent");
     let ParentState = ParentStates[0].getElementsByTagName('rhp:name');
     let name = ParentState[0].firstChild.nodeValue;
     let p='ROOT';

     if(name===p)
     {
      States=ChildOfstatesTag[i].getElementsByTagName('rhp:name');
      let li = document.createElement('li');
      let State = States[0].firstChild.nodeValue;
      li.textContent = '% ' + State + ' ' + q;
      //list.appendChild(li); -> UNCOMMENT TO SEE ALL THE STATES.
      let obj={"key":q, "text":State};
      //console.log(obj);
      nodeDataArray[q] = obj;
      q++;
      dict= {State: i};
     }

    }
    //After the for loop above all the states which have a parent root are
       put in the list.


    // 1) We first run the same loop as above to again find all the states
       which have their parent state as the 'ROOT' state.
    let m=0;
    let f=0;
```

```
for(let i=0; i<ChildOfstatesTag.length; i++)
{
 let ParentStates= ChildOfstatesTag[i].getElementsByTagNameNS("http://
    w3.ibm.com/xmlns/terms/Rhapsody","ItsParent");
 let ParentState = ParentStates[0].getElementsByTagName('rhp:name');
 let name = ParentState[0].firstChild.nodeValue;
 let p='ROOT';

 var identity = new Object();

 if(name===p)
 {
  States=ChildOfstatesTag[i].getElementsByTagName('rhp:name');
  let li = document.createElement('li');
  let State = States[0].firstChild.nodeValue;
  li.textContent = State + '  ' + f;
  //list.appendChild(li);  -> UNCOMMENT THE LINE TO SEE ALL THE
     STATES.
  identification = {State: f};
  f++;

  // 2) Then for each of the state we find all the outgoing edges and
     get their names as well.

  let OutgoingEdges = ChildOfstatesTag[i].getElementsByTagNameNS("
     http://w3.ibm.com/xmlns/terms/Rhapsody","OutgoingEdges");
  if(OutgoingEdges[0]!==undefined)
  {
   let OutgoingTransitionsOfAState=OutgoingEdges[0].
      getElementsByTagName('rhp:Transition');
   for (let j=0; j<OutgoingTransitionsOfAState.length; j++)
   {
     let li = document.createElement('li');
     let state = States[0].firstChild.nodeValue;
     let NameOfOutgoingEdge = OutgoingTransitionsOfAState[j].
        getElementsByTagName('rhp:name');
     let OutgoingEdgeOfState = NameOfOutgoingEdge[0].firstChild.
        nodeValue;

  // 3) Then we again find all the individual states inside the states
      tag. We proceed further only if the states tag is defined.(
     Again a simple check but I guess it was unnecessary). Again same
     as the above method we check the parent state of each of the
     state; whether it is 'ROOT' or not.

     let StatesTagI = xml.getElementsByTagNameNS("http://w3.ibm.com/
        xmlns/terms/Rhapsody","States");

     if(StatesTagI[0]!==undefined)
     {
       let ChildOfStatesTagI = StatesTagI[0].getElementsByTagNameNS("
```

```
            http://w3.ibm.com/xmlns/terms/Rhapsody","State");
        let tt=-1;
        for(let k=0; k<ChildOfStatesTagI.length; k++)
        {
         let statesI=ChildOfStatesTagI[k].getElementsByTagName('
             rhp:name');
         let statesIG=ChildOfStatesTagI[k].getElementsByTagName('
             rhp:ItsParent');
         let statesIGV=statesIG[0].getElementsByTagName('rhp:name');
         let statesppcc=statesIGV[0].firstChild.nodeValue;
         let pt=0;
         if(statesppcc==='ROOT')
         {
           tt++;
         }
    // 4) We then check the incoming edges of each of these individual
        states and list out the names.
           let IncomingEdges =ChildOfStatesTagI[k].getElementsByTagNameNS
              ("http://w3.ibm.com/xmlns/terms/Rhapsody","IncomingEdges");

           if(IncomingEdges[0]!==undefined)
           {
            let IncomingTransitionsOfAState=IncomingEdges[0].
               getElementsByTagName('rhp:Transition');

            for (let l=0; l<IncomingTransitionsOfAState.length; l++)
            {
             let li = document.createElement('li');
             let stateI = statesI[0].firstChild.nodeValue;
             let NameOfIncomingEdge = IncomingTransitionsOfAState[l].
                getElementsByTagName('rhp:name');
             let IncomingEdgeOfState = NameOfIncomingEdge[0].firstChild.
                nodeValue;

    // 5) Now we have the outgoing edges for each state and also the
        incoming edges for each state. The j-loop loops through each
        outgoing edge of each state. When it reaches a particular outgoing
        transition of a particular state the code runs another inner loop(
        the k-loop and l-loop) that checks all the incoming transitions of
        each of the state. We then check if the outgoing transition that we
        've got from the outer loop is equal to the incoming transition we'
        ve got from the inner loop. If it is; we output a statement that
        XYZ is a transition from state A(outer loop) to state B(inner loop)
        .

             if(IncomingEdgeOfState===OutgoingEdgeOfState)
             {
              li.textContent= IncomingEdgeOfState + '  - is a transition
                 from   ' + state + '  to   ' + stateI;
              //list.appendChild(li); -> UNCOMMENT TO SEE ALL THE
                 TRANSITIONS ALONG WITH THE FROM AND TO STATES.
```

```
            let TransitionsTag = xml.getElementsByTagNameNS("http://w3.
                ibm.com/xmlns/terms/Rhapsody","Transitions");

    // 6) Then we check the transitions tag inside the XML; we check the
        label of the edge by using a for loop(w-loop). When the loop sees
        that a particular transition inside the transitions tag is equal to
         the edge we find out the label for that transition.

            if(TransitionsTag[0]!==undefined)
            {
             let ChildOfTransitionsTag =TransitionsTag[0].
                 getElementsByTagNameNS("http://w3.ibm.com/xmlns/terms/
                 Rhapsody","Transition");

             for(let w=0; w<ChildOfTransitionsTag.length; w++)
             {
              labels=ChildOfTransitionsTag[w].getElementsByTagName('
                  rhp:label');
              transitions=ChildOfTransitionsTag[w].getElementsByTagName
                  ('rhp:name');
              let transition = transitions[0].firstChild.nodeValue;

              if(labels[0]!==undefined && transition===
                  IncomingEdgeOfState)
              {
               let label=labels[0].firstChild.nodeValue;
               TransitionName=label;

      // 7) Here we finally make an object containing 'from and to' along
          with the transition name. And then fill it to the 'linkDataArray'
          .

               let object={"from":identification["State"], "to":tt , "
                   text":TransitionName};
               linkDataArray[m] = object;
               //console.log(linkDataArray[m]);
               m++;
              }

             }

            }

           }

          }

         }

        }
```

```
        }

       }

      }

     }

    }

   }

if (window.goSamples) goSamples();  // init for these samples -- you
    don't need to call this
var $ = go.GraphObject.make;  // for conciseness in defining templates

myDiagram =$(go.Diagram, "myDiagramDiv",  // must name or refer to the
    DIV HTML element
{
  initialAutoScale: go.Diagram.Uniform,  // an initial automatic zoom-
      to-fit
  contentAlignment: go.Spot.Center,  // align document to the center of
      the viewport
  layout:
        $(go.ForceDirectedLayout,  // automatically spread nodes apart
          { maxIterations: 200, defaultSpringLength: 30,
            defaultElectricalCharge: 100 })
 });

 // define each Node's appearance
myDiagram.nodeTemplate =$(go.Node, "Auto",  // the whole node panel
{ locationSpot: go.Spot.Center },
     // define the node's outer shape, which will surround the
        TextBlock
     $(go.Shape, "Rectangle",
       { fill: $(go.Brush, "Linear", { 0: "rgb(254, 201, 0)", 1: "rgb
           (254, 162, 0)" }), stroke: "black" }),
     $(go.TextBlock,
       { font: "bold 10pt helvetica, bold arial, sans-serif", margin: 4
           },
       new go.Binding("text", "text"))
);

 // replace the default Link template in the linkTemplateMap
myDiagram.linkTemplate = $(go.Link,  // the whole link panel
     $(go.Shape,  // the link shape
       { stroke: "black" }),
     $(go.Shape,  // the arrowhead
       { toArrow: "standard", stroke: null }),
     $(go.Panel, "Auto",
       $(go.Shape,  // the label background, which becomes transparent
```

```
            around the edges
          {
            fill: $(go.Brush, "Radial", { 0: "rgb(240, 240, 240)", 0.3:
                "rgb(240, 240, 240)", 1: "rgba(240, 240, 240, 0)" }),
            stroke: null
          }),
        $(go.TextBlock,  // the label text
          {
            textAlign: "center",
            font: "10pt helvetica, arial, sans-serif",
            stroke: "#555555",
            margin: 4
          },
          new go.Binding("text", "text"))
      )
    );

    // create the model for the concept map
    myDiagram.model = new go.GraphLinksModel(nodeDataArray, linkDataArray)
        ;
}


  // A custom layout that sizes each "Super" node to be big enough to
      cover all of it member nodes
function CustomLayout() {
    go.Layout.call(this);
  }
  go.Diagram.inherit(CustomLayout, go.Layout);

  CustomLayout.prototype.doLayout = function(coll) {
    coll = this.collectParts(coll);

    var supers = new go.Set(/*go.Node*/);
    coll.each(function(p) {
      if (p instanceof go.Node && p.category === "Super") supers.add(p);
    });

    function membersOf(sup, diag) {
      var set = new go.Set(/*go.Part*/);
      var arr = sup.data._members;
      for (var i = 0; i < arr.length; i++) {
        var d = arr[i];
        set.add(diag.findNodeForData(d));
      }
      return set;
    }

    function isReady(sup, supers, diag) {
      var arr = sup.data._members;
      for (var i = 0; i < arr.length; i++) {
```

```
        var d = arr[i];
        if (d.category !== "Super") continue;
        var n = diag.findNodeForData(d);
        if (supers.has(n)) return false;
      }
      return true;
    }

    // implementations of doLayout that do not make use of a LayoutNetwork
    // need to perform their own transactions
    this.diagram.startTransaction("Custom Layout");

    while (supers.count > 0) {
      var ready = null;
      var it = supers.iterator;
      while (it.next()) {
        if (isReady(it.value, supers, this.diagram)) {
          ready = it.value;
          break;
        }
      }
      supers.remove(ready);
      var b = this.diagram.computePartsBounds(membersOf(ready, this.
          diagram));
      ready.location = b.position;
      var body = ready.findObject("BODY");
      if (body) body.desiredSize = b.size;
    }

    this.diagram.commitTransaction("Custom Layout");
  };
  // end CustomLayout


  // Define a custom DraggingTool
  function CustomDraggingTool() {
    go.DraggingTool.call(this);
  }
  go.Diagram.inherit(CustomDraggingTool, go.DraggingTool);

  CustomDraggingTool.prototype.moveParts = function(parts, offset, check)
      {
    go.DraggingTool.prototype.moveParts.call(this, parts, offset, check);
    this.diagram.layoutDiagram(true);
  };

  CustomDraggingTool.prototype.computeEffectiveCollection = function(parts
      ) {
    var coll = new go.Set(/*go.Part*/).addAll(parts);
    var tool = this;
    parts.each(function(p) {
```

```
      tool.walkSubTree(p, coll);
    });
    return go.DraggingTool.prototype.computeEffectiveCollection.call(this,
        coll);
  };

  // Find other attached nodes.
  CustomDraggingTool.prototype.walkSubTree = function(sup, coll) {
    if (sup === null) return;
    coll.add(sup);
    if (sup.category !== "Super") return;
    // recurse through this super state's members
    var model = this.diagram.model;
    var mems = sup.data._members;
    if (mems) {
      for (var i = 0; i < mems.length; i++) {
        var mdata = mems[i];
        this.walkSubTree(this.diagram.findNodeForData(mdata), coll);
      }
    }
  };
  // end CustomDraggingTool class

  // Show the diagram's model in JSON format
  function save() {
    document.getElementById("mySavedModel").value = myDiagram.model.toJson
        ();
    myDiagram.isModified = false;
  }
  function load() {
    myDiagram.model = go.Model.fromJson(document.getElementById("
        mySavedModel").value);

    // make sure all data have up-to-date "members" collections
    // this does not prevent any "cycles" of membership, which would
        result in undefined behavior
    var arr = myDiagram.model.nodeDataArray;
    for (var i = 0; i < arr.length; i++) {
      var data = arr[i];
      var supers = data.supers;
      if (supers) {
        for (var j = 0; j < supers.length; j++) {
          var sdata = myDiagram.model.findNodeDataForKey(supers[j]);
          if (sdata) {
            // update _supers to be an Array of references to node data
            if (!data._supers) {
              data._supers = [sdata];
            } else {
              data._supers.push(sdata);
            }
            // update _members to be an Array of references to node data
```

```
            if (!sdata._members) {
              sdata._members = [data];
            } else {
              sdata._members.push(data);
            }
          }
        }
      }
    }

</script>

</head>
<body>
<div id="sample">
  <div id="myDiagramDiv" style="background-color: whitesmoke; border: solid
      1px black; width: 100%; height: 700px"></div>
  <p>
    A concept map sample depicting various suggested relationships between
        different ideas.
    See also the <a href="interactiveForce.html">Interactive Force</a>
        sample that uses the exact same data
    but a different node template and an interactive <a>ForceDirectedLayout<
        /a>.
  </p>
</div>
</body>
</html>
```

Sub-State Machine (Appendix B)

Now that we have used the Go.Js library to make a state machine it is now time to use the same library to make a substate chart that goes one level below the metamodel. The code written can be used to find all the substates and their transitions inside an XML.

We describe the step by step method followed to read the XML data along with the method the GO.Js library used to convert the JSON objects into a state machine.

1) We define 'nodeDataArray' and 'linkDataArray' as global arrays so that they can be used later outside the function. Both these variables are an array of objects where the nodeDataArray consist of the super-state and the sub states along with the identifiers. The substates have a property called 'super' that helps the sub-state know what it's super state is. Whereas the link data array consists of all the transitions between the various states.

2) We first use the 'getElemetsByTagNameNS' function to find the 'states' tag' in the entire XML and we can find more states tag inside a particular state tag along with the individual sub states. We know that all these states will have the same parent state; we find the name of that parent state and put it inside individual objects with the "category":"Super". This makes a list of all superstates.

3) Next we find the names of all the individual states in the sub 'states' tag and put it inside objects with a key, location and the key value of the superstate as well. So now the 'nodeDataArray' is completely filled with objects.

4)The next step is to find all the transitions that exist between all the states. In order to do so we first get all the sub-states tags and then loop through each one of them. When we are at a particular sub-state we get all the names of the sub-states.

5)After we have found the individual states inside a particular substate we find the outgoing edges of these individual state. For each individual outgoing edge of a particular state we run again an entire loop of sub-states tag with the individual states inside them to find all the incoming transitions of a particular sub-state.

6) From here we get all the individual states again along with all the incoming transitions and their respective names. The same method of looping through each substate tag is followed.

7) We loop through the entire XML and check whether any of the incoming edges is equal to the current outgoing edge, and if it is we go forward inside the if statement and output a statement saying that 'ABC is a transition from state A to state B'.

8) Finally we fill the nodeDataArray and linkDataArray values inside another object that later is converted into a string so that it can be fed into myDiagram.model. The myDagram.model then uses the Go.JS library to display the necessary state-machine.

The next step is that the node and link data is read appropriately by the library:

1) The library consists of templates defined for nodes and links. The nodes have their size and shape defined along with the type and format of the text inside the nodes. The link haveproperties like the arrowhead, the label background, the label text which defines the font as well as the text alignment.

2) The library also defines a custom layout of the state machine along with a custom dragging-tool that will help in zooming in and out of the state machine so that only certain parts of thestate machine are visible.

3) The library also has a save and load method that helps in saving the model and then loading it in the library using the my diagram element.

```
<html>
<head>
  <meta charset="UTF-8">
  <title>State Chart</title>
  <meta name="description" content="A finite state machine chart with
      editable and interactive features." />
  <meta name="viewport" content="width=device-width, initial-scale=1">
  <!-- Copyright 1998-2020 by Northwoods Software Corporation. -->

 <!-- <script src="../release/go.js"></script>
  <!--<script src="../assets/js/goSamples.js"></script> <!-- this is only
      for the GoJS Samples framework -->
<main>
  <ul id="nested-elements"></ul>
  <ul id="States"></ul>
  <ul id="Transitions"></ul>
  <ul id="incoming-edges"></ul>
  <pre id="output"></pre>
</main>
```

```
  <script src="jquery-3.4.1.js"></script>
  <script src="go.js"></script>
  <script src="require.js"></script>
<script>
 var s;
document.addEventListener('DOMContentLoaded', ()=>{

let url="Paper2.xml";
fetch(url)
.then(response=>response.text())
.then(data=>{
  let parser = new DOMParser();
  let xml = parser.parseFromString(data, "application/xml");
  var identifier ={};
  //document.getElementById('output').textContent=data;
  //console.log(xml);
  BuildJsonFile(xml);
})
})

  // 1) We define list along with 'nodeDataArray' and 'linkDataArray' as
      global arrays so that they can be used later outside the function.
let list = document.getElementById("States");
let nodeDataArray= [];
let linkDataArray= [];

let ik=0;
function BuildJsonFile(xml){

  // 2) We define two tags where you get the 'states tag' from the entire
     XML and we also get 'states tag' inside a particular 'states tag'(if it
      exists); that is we get one substate down for an initial state. Now
     the 'nodeDataArray' and 'linkDataArray' will be filled with all the
     array of objects that are to be read by the go.js library.
  let statesTagUP = xml.getElementsByTagNameNS("http://w3.ibm.com/xmlns/
     terms/Rhapsody","States");
  let statesTag = statesTagUP[0].getElementsByTagNameNS("http://w3.ibm.com/
     xmlns/terms/Rhapsody","States");
  let q=0;
  let tbc=0;

  for(let ik=0; ik<statesTag.length; ik++)
  {
  // 3) We first find the substate 'states tag' and all the individual
     states inside it. We know that all these states will have the same
     parent state; we find the name of that parent state and put it inside
     individual objects with the "category":"Super". This makes a list of
     all superstates.

    let ChildOfstatesTagR = statesTag[ik].getElementsByTagNameNS("http://w3.
       ibm.com/xmlns/terms/Rhapsody","State");
```

```
    let ItsParent = ChildOfstatesTagR[0].getElementsByTagNameNS("http://w3.
        ibm.com/xmlns/terms/Rhapsody","ItsParent");
    let Name = ItsParent[0].getElementsByTagName("rhp:name");
    let ParentState = Name[0].firstChild.nodeValue;

  // NOTE: The key of the super state has to be negative. Hence '-1-ik'.
    let object={"key":-1-ik, "text":ParentState, "category": "Super"};
    nodeDataArray[tbc] = object;
    tbc++;
  }

  // 4) Now we find the names of all the individual states in the sub '
      states' tag and put it inside objects with a key, location and the key
      value of the superstate as well. So now the 'nodeDataArray' is
      completely filled with objects.
   let fc=0;
  for(let ik=0; ik<statesTag.length; ik++)
  {
    let ChildOfstatesTag = statesTag[ik].getElementsByTagNameNS("http://w3.
        ibm.com/xmlns/terms/Rhapsody","State");

    for(let i=0; i<ChildOfstatesTag.length; i++)
    {
      var dict = new Object();

      States=ChildOfstatesTag[i].getElementsByTagName('rhp:name');
      let li = document.createElement('li');
      let State = States[0].firstChild.nodeValue;
      li.textContent = '% ' + State + ' ' + q;
      //list.appendChild(li);  -> UNCOMMENT WHEN YOU WANT TO DISPLAY THE
          STATES.

      let b=Math.floor((Math.random() * 260) + 10*(ik+1));
      let a=Math.floor((Math.random() * 260) + 10*(ik+1));

      let obj={"key":fc+1, "loc": b + " " + a, "text":State, "supers":[-ik
          -1]};
      nodeDataArray[tbc] = obj;
      tbc++;
      dict= {State: fc+1};
      fc++;
    }

 }


// 5) The next step is to find all the transitions that exist between all
    the states. In order to do so we first get all the sub-states tags and
    then loop through each one of them. When we are at a particular sub-state
     we get all the names of the sub-states.
 let m=0;
```

```
let f=1;
   for(let ip=0; ip<statesTag.length; ip++)
 {
let ChildOfstatesTag = statesTag[ip].getElementsByTagNameNS("http://w3.ibm.
   com/xmlns/terms/Rhapsody","State");
// 5) The next step is to find all the transitions that exist between all
   the states. In order to do so we first find all the names of the
   individual states inside a particular substate tag.

   for(let g=0; g<ChildOfstatesTag.length; g++)
{
    States=ChildOfstatesTag[g].getElementsByTagName('rhp:name');
    let li = document.createElement('li');
    let State = States[0].firstChild.nodeValue;
    li.textContent = State + '  ' + f;
    //list.appendChild(li); -> UNCOMMENT WHEN YOU WANT TO DISPLAY THE
       STATES.
    identification = {State: f};
    f++;

// 6) After we have found the individual states inside a particular
   substate we find the outgoing edges of these individual states. For each
    individual outgoing edge of a particular state we run again an entire
   loop of sub-states along with the incoming transitions.

    let OutgoingEdges = ChildOfstatesTag[g].getElementsByTagNameNS("http://
       w3.ibm.com/xmlns/terms/Rhapsody","OutgoingEdges");

    if(OutgoingEdges[0]!==undefined)
    {
      let OutgoingTransitionsOfAState=OutgoingEdges[0].getElementsByTagName
         ('rhp:Transition');
      for (let j=0; j<OutgoingTransitionsOfAState.length; j++)
      {
        let li = document.createElement('li');
        let state = States[0].firstChild.nodeValue;
        let NameOfOutgoingEdge = OutgoingTransitionsOfAState[j].
           getElementsByTagName('rhp:name');
        let OutgoingEdgeOfState = NameOfOutgoingEdge[0].firstChild.
           nodeValue;


   // 7) From here we get all the individual states again along with all
      the incoming transitions and their respective names. The same method
      of looping through each substate is followed.

        let StatesTagBig = xml.getElementsByTagNameNS("http://w3.ibm.com/
           xmlns/terms/Rhapsody","States");
        let StatesTagI = StatesTagBig[0].getElementsByTagNameNS("http://w3.
           ibm.com/xmlns/terms/Rhapsody","States");
        let tt=0;
```

```
        for(let ip=0; ip< StatesTagI.length; ip++)
        {

        if(StatesTagI[0]!==undefined)
        {
          let ChildOfStatesTagI = StatesTagI[ip].getElementsByTagNameNS("
              http://w3.ibm.com/xmlns/terms/Rhapsody","State");

          for(let k=0; k<ChildOfStatesTagI.length; k++)
          {
              let statesI=ChildOfStatesTagI[k].getElementsByTagName('
                  rhp:name');
              let pt=0;
              tt++;

              let IncomingEdges =ChildOfStatesTagI[k].getElementsByTagNameNS
                  ("http://w3.ibm.com/xmlns/terms/Rhapsody","IncomingEdges");

              if(IncomingEdges[0]!==undefined)
              {
                   let IncomingTransitionsOfAState=IncomingEdges[0].
                       getElementsByTagName('rhp:Transition');

                   for (let l=0; l<IncomingTransitionsOfAState.length; l++)
                   {
                     let li = document.createElement('li');
                     let stateI = statesI[0].firstChild.nodeValue;
                     let NameOfIncomingEdge = IncomingTransitionsOfAState[l].
                         getElementsByTagName('rhp:name');
                     let IncomingEdgeOfState = NameOfIncomingEdge[0].
                         firstChild.nodeValue;

    // 8) After looping through the entire xml we check any of the incoming
        edges is equal to the current outgoing edge, and if it is we go
        forward inside the if statement and output a statement saying that '
        ABC is a transition from state A to state B'
                     if(IncomingEdgeOfState===OutgoingEdgeOfState)
                     {
                       li.textContent= IncomingEdgeOfState + '  - is a
                           transition from   ' + state + '  to   ' + stateI;
                       //list.appendChild(li);  -> UNCOMMENT WHEN YOU WANT
                           TO DISPLAY THE EDGES AND THEIR SOURCE AND
                           DESTINATION

                       let TransitionsTag = xml.getElementsByTagNameNS("http:
                           //w3.ibm.com/xmlns/terms/Rhapsody","Transitions");
                       if(TransitionsTag[0]!==undefined)
                       {
                         let ChildOfTransitionsTag =TransitionsTag[0].
                             getElementsByTagNameNS("http://w3.ibm.com/xmlns/
```

```
                        terms/Rhapsody","Transition");

                    for(let w=0; w<ChildOfTransitionsTag.length; w++)
                    {
                     labels=ChildOfTransitionsTag[w].
                         getElementsByTagName('rhp:label');
                     transitions=ChildOfTransitionsTag[w].
                         getElementsByTagName('rhp:name');
                     let transition = transitions[0].firstChild.
                         nodeValue;

                     if(labels[0]!==undefined && transition===
                         IncomingEdgeOfState)
                     {
                       let label=labels[0].firstChild.nodeValue;
                       TransitionName=label;
                       let object={"from":identification["State"], "to"
                           :tt, "text":TransitionName};
                       linkDataArray[m] = object;
                      m++;
                     }
                    }
                   }
                  }
                 }
                }
               }
              }
             }
            }
           }
          }
         }
        }
       }
      }
     }
    }
   }
  }
 }

  // 9) Here we fill the nodeDataArray and linkDataArray values inside the
       pcc object and later we convert it into a string so that it can be
      fed into myDiagram.model.

  let pcc={"class": "go.GraphLinksModel",
          "nodeDataArray": nodeDataArray,
          "linkDataArray": linkDataArray }

  console.log(linkDataArray);

  let pt=JSON.stringify(pcc);
  myDiagram.model = go.Model.fromJson(pt);

  var arr = myDiagram.model.nodeDataArray;
  for (var i = 0; i < arr.length; i++) {
      var data = arr[i];
      var supers = data.supers;
```

```
        if (supers) {
          for (var j = 0; j < supers.length; j++) {
            var sdata = myDiagram.model.findNodeDataForKey(supers[j]);
            if (sdata) {
              // update _supers to be an Array of references to node data
              if (!data._supers) {
                data._supers = [sdata];
              } else {
                data._supers.push(sdata);
              }
              // update _members to be an Array of references to node data
              if (!sdata._members) {
                sdata._members = [data];
              } else {
                sdata._members.push(data);
              }
            }
          }
        }
      }

</script>
<script>
    function init() {
      if (window.goSamples) goSamples();  // init for these samples -- you
          don't need to call this
      var $ = go.GraphObject.make;  // for conciseness in defining templates

      myDiagram =
        $(go.Diagram, "myDiagramDiv",  // must name or refer to the DIV HTML
            element
          {
            allowCopy: false,
            allowDelete: false,
            draggingTool: $(CustomDraggingTool),
            layout: $(CustomLayout),
            // enable undo & redo
            "undoManager.isEnabled": true
          });

      // define the Node template
      myDiagram.nodeTemplate =
        $(go.Node, "Auto",
          new go.Binding("location", "loc", go.Point.parse).makeTwoWay(go.
              Point.stringify),
          // define the node's outer shape, which will surround the
              TextBlock
          $(go.Shape, "RoundedRectangle",
            {
              fill: "rgb(254, 201, 0)", stroke: "black", parameter1: 20,  //
```

```
            the corner has a large radius
          portId: "", fromSpot: go.Spot.AllSides, toSpot: go.Spot.
              AllSides
        }),
      $(go.TextBlock,
        new go.Binding("text", "text").makeTwoWay())
    );

myDiagram.nodeTemplateMap.add("Super",
  $(go.Node, "Auto",
    { locationObjectName: "BODY" },
    $(go.Shape, "RoundedRectangle",
      {
        fill: "rgba(128, 128, 64, 0.5)", strokeWidth: 1.5, parameter1:
            20,
        spot1: go.Spot.TopLeft, spot2: go.Spot.BottomRight, minSize:
            new go.Size(30, 30)
      }),
    $(go.Panel, "Vertical",
      { margin: 20 },
      $(go.TextBlock,
        { font: "bold 12pt sans-serif", margin: new go.Margin(0, 0, 5,
            0) },
        new go.Binding("text")),
      $(go.Shape,
        { name: "BODY", opacity: 0 })
    )
  ));

// replace the default Link template in the linkTemplateMap
 myDiagram.linkTemplate =
  $(go.Link,  // the whole link panel
    $(go.Shape,  // the link shape
      { stroke: "black" }),
    $(go.Shape,  // the arrowhead
      { toArrow: "standard", stroke: null }),
    $(go.Panel, "Auto",
      $(go.Shape,  // the label background, which becomes transparent
          around the edges
        {
          fill: $(go.Brush, "Radial", { 0: "rgb(240, 240, 240)", 0.3:
              "rgb(240, 240, 240)", 1: "rgba(240, 240, 240, 0)" }),
          stroke: null
        }),
      $(go.TextBlock,  // the label text
        {
          textAlign: "center",
          font: "12pt helvetica, arial, sans-serif",
          stroke: "#555555",
          margin: 4
        },
```

```
            new go.Binding("text", "text"))
        )
      );

    // read in the JSON-format data from the "mySavedModel" element
    //load();
  }

  // A custom layout that sizes each "Super" node to be big enough to
     cover all of it member nodes
  function CustomLayout() {
    go.Layout.call(this);
  }
  go.Diagram.inherit(CustomLayout, go.Layout);

  CustomLayout.prototype.doLayout = function(coll) {
    coll = this.collectParts(coll);

    var supers = new go.Set(/*go.Node*/);
    coll.each(function(p) {
      if (p instanceof go.Node && p.category === "Super") supers.add(p);
    });

    function membersOf(sup, diag) {
      var set = new go.Set(/*go.Part*/);
      var arr = sup.data._members;
      for (var i = 0; i < arr.length; i++) {
        var d = arr[i];
        set.add(diag.findNodeForData(d));
      }
      return set;
    }

    function isReady(sup, supers, diag) {
      var arr = sup.data._members;
      for (var i = 0; i < arr.length; i++) {
        var d = arr[i];
        if (d.category !== "Super") continue;
        var n = diag.findNodeForData(d);
        if (supers.has(n)) return false;
      }
      return true;
    }

    // implementations of doLayout that do not make use of a LayoutNetwork
    // need to perform their own transactions
    this.diagram.startTransaction("Custom Layout");

    while (supers.count > 0) {
      var ready = null;
      var it = supers.iterator;
```

```
      while (it.next()) {
        if (isReady(it.value, supers, this.diagram)) {
          ready = it.value;
          break;
        }
      }
      supers.remove(ready);
      var b = this.diagram.computePartsBounds(membersOf(ready, this.
          diagram));
      ready.location = b.position;
      var body = ready.findObject("BODY");
      if (body) body.desiredSize = b.size;
    }

    this.diagram.commitTransaction("Custom Layout");
  };
  // end CustomLayout


  // Define a custom DraggingTool
  function CustomDraggingTool() {
    go.DraggingTool.call(this);
  }
  go.Diagram.inherit(CustomDraggingTool, go.DraggingTool);

  CustomDraggingTool.prototype.moveParts = function(parts, offset, check)
      {
    go.DraggingTool.prototype.moveParts.call(this, parts, offset, check);
    this.diagram.layoutDiagram(true);
  };

  CustomDraggingTool.prototype.computeEffectiveCollection = function(parts
      ) {
    var coll = new go.Set(/*go.Part*/).addAll(parts);
    var tool = this;
    parts.each(function(p) {
      tool.walkSubTree(p, coll);
    });
    return go.DraggingTool.prototype.computeEffectiveCollection.call(this,
        coll);
  };

  // Find other attached nodes.
  CustomDraggingTool.prototype.walkSubTree = function(sup, coll) {
    if (sup === null) return;
    coll.add(sup);
    if (sup.category !== "Super") return;
    // recurse through this super state's members
    var model = this.diagram.model;
    var mems = sup.data._members;
    if (mems) {
```

```
        for (var i = 0; i < mems.length; i++) {
          var mdata = mems[i];
          this.walkSubTree(this.diagram.findNodeForData(mdata), coll);
        }
      }
    };
    // end CustomDraggingTool class

    // Show the diagram's model in JSON format
    function save() {
      document.getElementById("mySavedModel").value = myDiagram.model.toJson
          ();
      myDiagram.isModified = false;
    }

    function load() {
      //myDiagram.model = go.Model.fromJson(document.getElementById("
          mySavedModel").value);

      // make sure all data have up-to-date "members" collections
      // this does not prevent any "cycles" of membership, which would
          result in undefined behavior
      var arr = myDiagram.model.nodeDataArray;
      for (var i = 0; i < arr.length; i++) {
        var data = arr[i];
        var supers = data.supers;
        if (supers) {
          for (var j = 0; j < supers.length; j++) {
            var sdata = myDiagram.model.findNodeDataForKey(supers[j]);
            if (sdata) {
              // update _supers to be an Array of references to node data
              if (!data._supers) {
                data._supers = [sdata];
              } else {
                data._supers.push(sdata);
              }
              // update _members to be an Array of references to node data
              if (!sdata._members) {
                sdata._members = [data];
              } else {
                sdata._members.push(data);
              }
            }
          }
        }
      }
    }
    </script>
</head>
<body onload="init()">
<div id="sample">
```

```
  <div id="myDiagramDiv" style="border: solid 1px black; width: 100%;
      height: 700px"></div>
  <p>
  This demonstrates the ability to simulate having nodes be members of
      multiple "groups".
  Regular <a>Group</a>s only support a single <a>Part.containingGroup</a>
      for nodes.
  This sample does not make use of <a>Group</a>s at all, but simulates one
      possible "grouping" relationship
  using a custom <a>Layout</a> and a custom <a>DraggingTool</a>.
  </p>
 </div>
</body>
</html>
```

Block Definition Diagram (Appendix C)

A thorough research was done on the XML file for block diagrams that are generated by IBM Rhapsody. The XML has no tags to show the various relations between the block so hence we focus on only displaying the blocks without showing the relations between them.

1) We first define an empty array of objects that is to be filled later with all the classes and identifiers as objects.

2) Next we find the model elements tag. The model elements tag has different types of elements; like class, objects, associations and generalizations. We check for each model element whether it's equal to 'rhp:Class'; if it is we put the name of the class along with an identifier into an object.

The next step is that the node and link data is read appropriately by the library:

1) The library had a function init defined that has the size and template to be followed for the block. It has both the node template and the link template for the block and the various relations between the blocks. The library has different types of arrows for different relations.

```
<!DOCTYPE html>
<html>

<head>
  <meta charset="UTF-8">
  <title>UML Class Nodes</title>
  <meta name="description" content="UML Class-like nodes showing two
      collapsible lists of items." />
  <meta name="viewport" content="width=device-width, initial-scale=1">
  <!-- Copyright 1998-2020 by Northwoods Software Corporation. -->
<script src="go.js"></script>
<main>

  <ul id="Classes"></ul>
```

```
  <ul id="Objects"></ul>
  <ul id="Association"></ul>
  <ul id="Generalization"></ul>
  <ul id="Class"></ul>
  <pre id="output"></pre>


</main>
<script>
document.addEventListener('DOMContentLoaded', ()=>{
let url="OMD.xml";
fetch(url)
.then(response=>response.text())
.then(data=>{
  let parser = new DOMParser();
  let xml = parser.parseFromString(data, "application/xml");
  //document.getElementById('output').textContent=data;
  //console.log(xml);

  buildClassList(xml);
  buildObjectList(xml);
  buildAssociationsList(xml);
  buildGeneralizationList(xml);
  init();
})
})

// Function to find all the classes in the block diagram
var nodedata=[];
// 1) We first define an empty array of objects that is to be filled later
   with all the classes and identifiers as objects.
function buildClassList(xml){
   let list = document.getElementById("Classes");
   let ClassesTag = xml.getElementsByTagNameNS("http://w3.ibm.com/xmlns/
      terms/Rhapsody","ContainedElements");
   if(ClassesTag[0]!==undefined)
  {
// 2) Next we find the model elements tag. The model elements tag has
   different types of elements; like class, objects, associations and
   generalizations. We check for each model element whether it's equal to '
   rhp:Class'; if it is we put the name of the class along with an
   identifier into an object.
   let ModelElements = ClassesTag[0].getElementsByTagNameNS("http://w3.ibm.
      com/xmlns/terms/Rhapsody","ModelElement");

   for(let i=0; i<ModelElements.length; i++)
       {
        if(ModelElements[i].getAttribute("xsi:type")==='rhp:Class')
            {
                let li = document.createElement('li');
                let Class= ModelElements[i].getElementsByTagNameNS("http://
```

```
                    w3.ibm.com/xmlns/terms/Rhapsody","name");
                let NameOfClass = Class[0].firstChild.nodeValue;
                li.textContent= NameOfClass + '.....................
                    ' + 'is a Class in the Block Diagram';
                nodedata[i]={key:i, name:NameOfClass};
                //list.appendChild(li); -> This is to display all the
                    classes in the class diagram.
            }

        }

    }
     else
     {
                let li = document.createElement('li');
                // let Class= ModelElements[i].getElementsByTagNameNS("http:
                    //w3.ibm.com/xmlns/terms/Rhapsody","name");
                //let NameOfClass = Class[0].firstChild.nodeValue;
                //li.textContent= 'No Class in the Block Diagram'; -> To
                    display that there is no class in the class diagram.
                list.appendChild(li);
     }

}

// Function to find all the Objects(Directed Composition Relations) in the
    block diagram
function buildObjectList(xml){
    let list = document.getElementById("Objects");
    let ObjectsTag = xml.getElementsByTagNameNS("http://w3.ibm.com/xmlns/
       terms/Rhapsody","ContainedElements");
    if(ObjectsTag[0]!==undefined)
   {
    let ModelElements = ObjectsTag[0].getElementsByTagNameNS("http://w3.ibm.
       com/xmlns/terms/Rhapsody","ModelElement");

    for(let i=0; i<ModelElements.length; i++)
        {
         if(ModelElements[i].getAttribute("xsi:type")==='rhp:Object')
             {
                let li = document.createElement('li');
                let Object= ModelElements[i].getElementsByTagNameNS("http://
                    w3.ibm.com/xmlns/terms/Rhapsody","name");
                let NameOfObject = Object[0].firstChild.nodeValue;
                li.textContent= NameOfObject+ '.....................
                    ' + 'is an Object in the Block Diagram';
                //list.appendChild(li); -> This is to display all the
                    objects in the class diagram.
             }

        }
```

```
    }

}

// Function to find all the Associations(Aggregation and Association
    Relations) in the block diagram
function buildAssociationsList(xml){
    let list = document.getElementById("Association");
    let AssociationsTag = xml.getElementsByTagNameNS("http://w3.ibm.com/xmlns
        /terms/Rhapsody","ContainedElements");

      if(AssociationsTag[0]!==undefined)
    {
      let ModelElements = AssociationsTag[0].getElementsByTagNameNS("http://
          w3.ibm.com/xmlns/terms/Rhapsody","ModelElement");

    for(let i=0; i<ModelElements.length; i++)
        {
         if(ModelElements[i].getAttribute("xsi:type")==='rhp:AssociationEnd')
             {
                 let li = document.createElement('li');
                 let Association= ModelElements[i].getElementsByTagNameNS("
                     http://w3.ibm.com/xmlns/terms/Rhapsody","name");
                 let NameOfAssociation = Association[0].firstChild.nodeValue;
                 li.textContent= NameOfAssociation+ '
                     ..........................' + 'is an Association End
                     in the Block Diagram';
                 //list.appendChild(li); -> This is to display all the
                     associations in the class diagram.
             }

        }

     }

}

// Function to find all the Generalizations(Generalizations Relations) in
    the block diagram
function buildGeneralizationList(xml){
    let list = document.getElementById("Generalization");
    let GeneralizationsTag = xml.getElementsByTagNameNS("http://w3.ibm.com/
        xmlns/terms/Rhapsody","ContainedElements");

      if(GeneralizationsTag[0]!==undefined)
    {
      let ModelElements = GeneralizationsTag[0].getElementsByTagNameNS("http:
          //w3.ibm.com/xmlns/terms/Rhapsody","ModelElement");

    for(let i=0; i<ModelElements.length; i++)
```

```
        {
         if(ModelElements[i].getAttribute("xsi:type")==='rhp:Generalization')
             {
                 let li = document.createElement('li');
                 let Generalizations= ModelElements[i].getElementsByTagNameNS
                    ("http://w3.ibm.com/xmlns/terms/Rhapsody","name");
                 let NameOfGeneralization = Generalizations[0].firstChild.
                    nodeValue;
                 li.textContent= NameOfGeneralization+ '
                    ..........' + 'is an Generalization in
                     the Block Diagram';
                 //list.appendChild(li);  ->   This is to display all the
                    generalization in the class diagram.
             }

        }

    }

}

// The funcion used to define all the node and link templates.
function init() {
      if (window.goSamples) goSamples();  // init for these samples -- you
         don't need to call this
      var $ = go.GraphObject.make;

      myDiagram =
        $(go.Diagram, "myDiagramDiv",
          {
            "undoManager.isEnabled": true,
            layout: $(go.TreeLayout,
              { // this only lays out in trees nodes connected by "
                  generalization" links
                angle: 90,
                path: go.TreeLayout.PathSource,  // links go from child to
                    parent
                setsPortSpot: false,  // keep Spot.AllSides for link
                    connection spot
                setsChildPortSpot: false,  // keep Spot.AllSides
                // nodes not connected by "generalization" links are laid
                    out horizontally
                arrangement: go.TreeLayout.ArrangementHorizontal
              })
          });

      // show visibility or access as a single character at the beginning of
          each property or method
      function convertVisibility(v) {
        switch (v) {
          case "public": return "+";
```

```
      case "private": return "-";
      case "protected": return "#";
      case "package": return "~";
      default: return v;
    }
  }

  // the item template for properties
  var propertyTemplate =
    $(go.Panel, "Horizontal",
      // property visibility/access
      $(go.TextBlock,
        { isMultiline: false, editable: false, width: 12 },
        new go.Binding("text", "visibility", convertVisibility)),
      // property name, underlined if scope=="class" to indicate static
          property
      $(go.TextBlock,
        { isMultiline: false, editable: true },
        new go.Binding("text", "name").makeTwoWay(),
        new go.Binding("isUnderline", "scope", function (s) { return s
            [0] === 'c' })),
      // property type, if known
      $(go.TextBlock, "",
        new go.Binding("text", "type", function (t) { return (t ? ": " :
            ""); })),
      $(go.TextBlock,
        { isMultiline: false, editable: true },
        new go.Binding("text", "type").makeTwoWay()),
      // property default value, if any
      $(go.TextBlock,
        { isMultiline: false, editable: false },
        new go.Binding("text", "default", function (s) { return s ? " =
            " + s : ""; }))
    );

  // the item template for methods
  var methodTemplate =
    $(go.Panel, "Horizontal",
      // method visibility/access
      $(go.TextBlock,
        { isMultiline: false, editable: false, width: 12 },
        new go.Binding("text", "visibility", convertVisibility)),
      // method name, underlined if scope=="class" to indicate static
          method
      $(go.TextBlock,
        { isMultiline: false, editable: true },
        new go.Binding("text", "name").makeTwoWay(),
        new go.Binding("isUnderline", "scope", function (s) { return s
            [0] === 'c' })),
      // method parameters
      $(go.TextBlock, "()",
```

```
        // this does not permit adding/editing/removing of parameters
            via inplace edits
        new go.Binding("text", "parameters", function (parr) {
          var s = "(";
          for (var i = 0; i < parr.length; i++) {
            var param = parr[i];
            if (i > 0) s += ", ";
            s += param.name + ": " + param.type;
          }
          return s + ")";
        })),
      // method return type, if any
      $(go.TextBlock, "",
        new go.Binding("text", "type", function (t) { return (t ? ": " :
            ""); })),
      $(go.TextBlock,
        { isMultiline: false, editable: true },
        new go.Binding("text", "type").makeTwoWay())
    );

  // this simple template does not have any buttons to permit adding or
  // removing properties or methods, but it could!
  myDiagram.nodeTemplate =
    $(go.Node, "Auto",
      {
        locationSpot: go.Spot.Center,
        fromSpot: go.Spot.AllSides,
        toSpot: go.Spot.AllSides
      },
      $(go.Shape, { fill: "lightyellow" }),
      $(go.Panel, "Table",
        { defaultRowSeparatorStroke: "black" },
        // header
        $(go.TextBlock,
          {
            row: 0, columnSpan: 2, margin: 3, alignment: go.Spot.Center,
            font: "bold 12pt sans-serif",
            isMultiline: false, editable: true
          },
          new go.Binding("text", "name").makeTwoWay()),
        // properties
        $(go.TextBlock, "Properties",
          { row: 1, font: "italic 10pt sans-serif" },
          new go.Binding("visible", "visible", function (v) { return !v;
              }).ofObject("PROPERTIES")),
        $(go.Panel, "Vertical", { name: "PROPERTIES" },
          new go.Binding("itemArray", "properties"),
          {
            row: 1, margin: 3, stretch: go.GraphObject.Fill,
            defaultAlignment: go.Spot.Left, background: "lightyellow",
            itemTemplate: propertyTemplate
```

```
            }
          ),
          $("PanelExpanderButton", "PROPERTIES",
            { row: 1, column: 1, alignment: go.Spot.TopRight, visible:
                false },
            new go.Binding("visible", "properties", function (arr) {
                return arr.length > 0; })),
          // methods
          $(go.TextBlock, "Methods",
            { row: 2, font: "italic 10pt sans-serif" },
            new go.Binding("visible", "visible", function (v) { return !v;
                }).ofObject("METHODS")),
          $(go.Panel, "Vertical", { name: "METHODS" },
            new go.Binding("itemArray", "methods"),
            {
              row: 2, margin: 3, stretch: go.GraphObject.Fill,
              defaultAlignment: go.Spot.Left, background: "lightyellow",
              itemTemplate: methodTemplate
            }
          ),
          $("PanelExpanderButton", "METHODS",
            { row: 2, column: 1, alignment: go.Spot.TopRight, visible:
                false },
            new go.Binding("visible", "methods", function (arr) { return
                arr.length > 0; }))
        )
      );

    function convertIsTreeLink(r) {
      return r === "generalization";
    }

    function convertFromArrow(r) {
      switch (r) {
        case "generalization": return "";
        default: return "";
      }
    }

    function convertToArrow(r) {
      switch (r) {
        case "generalization": return "Triangle";
        case "aggregation": return "StretchedDiamond";
        default: return "";
      }
    }

    myDiagram.linkTemplate =
      $(go.Link,
        { routing: go.Link.Orthogonal },
        new go.Binding("isLayoutPositioned", "relationship",
```

```
            convertIsTreeLink),
          $(go.Shape),
          $(go.Shape, { scale: 1.3, fill: "white" },
            new go.Binding("fromArrow", "relationship", convertFromArrow)),
          $(go.Shape, { scale: 1.3, fill: "white" },
            new go.Binding("toArrow", "relationship", convertToArrow))
        );

      // setup a few example class nodes and relationships
      var Nodedata = [
        {
          key: 1,
          name: "BankAccount"
        },
        {
          key: 11,
          name: "Person"
        },
        {
          key: 12,
          name: "Student"
        },
        {
          key: 13,
          name: "Professor"
        },
        {
          key: 14,
          name: "Course"
        }
      ];
      var linkdata = [
        { from: 12, to: 11, relationship: "generalization" },
        { from: 13, to: 11, relationship: "generalization" },
        { from: 14, to: 13, relationship: "aggregation" }
      ];
      myDiagram.model = $(go.GraphLinksModel,{
          copiesArrays: true,
          copiesArrayObjects: true,
          nodeDataArray: nodedata,
          linkDataArray: linkdata
        });
    }

</script>
</head>

<div id="sample">
<div id="myDiagramDiv" style="border: solid 1px black; width:100%;
    height:700px"></div>
<p>A list of all the classes inside the class diagram are given.</p>
```

```
</div>

</html>
```

Live-Link between IBM Rhapsody and Unity3D (Appendix D)

D.1 IBM Rhapsody

The ToUnity block created inside IBM Rhapsody contains value properties and operations defined to publish and subscribe to values with Unity3D via a live link.

The MQTT client libraries that are built and used inside of IBM Rhapsody can be run on two modes of operation namely synchronous and asynchronous. The major difference between the two modes is the use of the setCallback function. When the MQTTClient setCallbacks() function is used we get an asynchronous client and when it's not the client is synchronous.

The synchronous client runs on a single thread where messages are published using the MQTTClient publish() and MQTTClient publishMessage() functions. For subscription the functions MQTTClient receive() or MQTTClient yield() are used. These subscription functions perform the role of processing acknowledgements and make sure that the network connection of the server are alive at all times. When the user wants to know whether the message has been successfully delivered the MQTTClient waitForCompletion() can be used for a QOS of 1 and 2.

The asynchronous client can run on multiple threads at one time. The main function is used to call the publishing and subscribing functions. This is similar to the synchronous mode; however the networking connection and handshaking are processed and maintained in the background. The callback function is used to notify the status of the received message which is registered with the library by the call. However it's not possible to call from multiple threads with synchronization. For this we can use the MQTTAsync library but this is out of the scope of this project. For our implementation we use the asynchronous client using the MQTTClient.h library.

The initialization function creates a MQTT client using the MQTTClient.h library and connects to a local broker As can be seen the setCallback function is also used. In addition

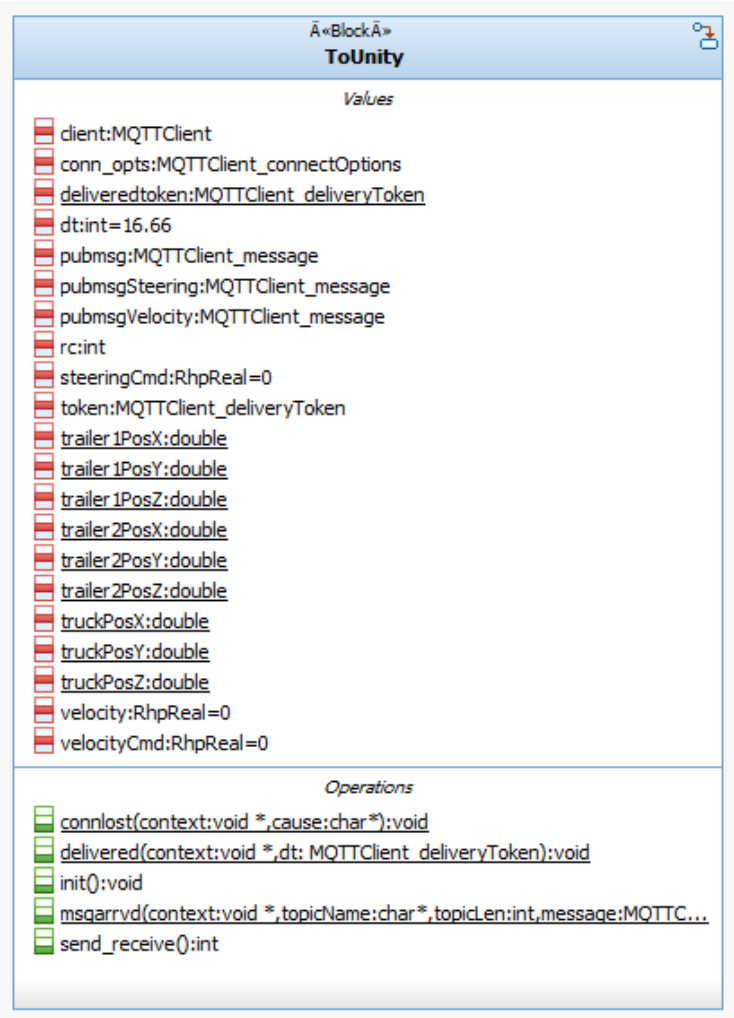

Figure D.1: To Unity Block

the initialization function subscribes to truck position topics on an active broker so that these values are received from Unity3D.

The send and receive function is used to publish values of steering wheel angle and speed on the broker. The values are first converted to strings using the gcvt() function. The operation waits for the publication to finish and then terminates for the next iteration.

Operation : init in ToUnity *

General | Description | Implementation | Arguments | Relations | Tags | Properties

void init()

```
#define ADDRESS                             "tcp://localhost:1883"
#define CLIENTID                            "ExampleClientPub"

#define TopicSubscribeTruckPosX             "TruckVariables/TruckMarkerPosX"
#define TopicSubscribeTruckPosY             "TruckVariables/TruckMarkerPosY"
#define TopicSubscribeTruckPosZ             "TruckVariables/TruckMarkerPosZ"
#define TopicSubscribeTrailerFrontPosX      "TruckVariables/TrailerFrontMarkerPosX"
#define TopicSubscribeTrailerFrontPosY      "TruckVariables/TrailerFrontMarkerPosY"
#define TopicSubscribeTrailerFrontPosZ      "TruckVariables/TrailerFrontMarkerPosZ"
#define TopicSubscribeTrailerBackPosX       "TruckVariables/TrailerBackMarkerPosX"
#define TopicSubscribeTrailerBackPosY       "TruckVariables/TrailerBackMarkerPosY"
#define TopicSubscribeTrailerBackPosZ       "TruckVariables/TrailerBackMarkerPosZ"

#define QOS                                 1

 conn_opts = MQTTClient_connectOptions_initializer;
 pubmsg = MQTTClient_message_initializer;
 pubmsgVelocity = MQTTClient_message_initializer;
 pubmsgSteering = MQTTClient_message_initializer;


 MQTTClient_create(&client, ADDRESS, CLIENTID, MQTTCLIENT_PERSISTENCE_NONE, NULL);
 conn_opts.keepAliveInterval = 20;
 conn_opts.cleansession = 1;
MQTTClient_setCallbacks(client, NULL, connlost, msgarrvd, delivered);
 if ((rc = MQTTClient_connect(client, &conn_opts)) != MQTTCLIENT_SUCCESS)
  {
      printf("Failed to connect, return code %d\n", rc);
      exit(EXIT_FAILURE);
  }
  MQTTClient_subscribe(client, TopicSubscribeTruckPosX, QOS);
  MQTTClient_subscribe(client, TopicSubscribeTruckPosY, QOS);
  MQTTClient_subscribe(client, TopicSubscribeTruckPosZ, QOS);
  MQTTClient_subscribe(client, TopicSubscribeTrailerFrontPosX, QOS);
  MQTTClient_subscribe(client, TopicSubscribeTrailerFrontPosY, QOS);
  MQTTClient_subscribe(client, TopicSubscribeTrailerFrontPosZ, QOS);
  MQTTClient_subscribe(client, TopicSubscribeTrailerBackPosX, QOS);
  MQTTClient_subscribe(client, TopicSubscribeTrailerBackPosY, QOS);
  MQTTClient_subscribe(client, TopicSubscribeTrailerBackPosZ, QOS);
```

Figure D.2: Initialization Function

Operation : send_receive in ToUnity *

General | Description | Implementation | Arguments | Relations | Tags | Properties

int send_receive()

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "MQTTClient.h"

#define CLIENTID            "ExampleClientPub"

#define TOPICVELOCITY       "TruckVariables/Velocity"
#define TOPICSTEERING       "TruckVariables/SteeringAngle"

#define PAYLOAD     "Hello World!"

#define QOS         1

#define TIMEOUT     10000L

char VelocityString[10];
char SteeringWheelString[10];

gcvt(velocityCmd, 6, VelocityString);
gcvt(steeringCmd, 6, SteeringWheelString);

pubmsgVelocity.payload = VelocityString;
pubmsgVelocity.payloadlen = strlen(VelocityString);
pubmsgVelocity.qos = QOS;
pubmsgVelocity.retained = 0;
pubmsgSteering.payload = SteeringWheelString;
pubmsgSteering.payloadlen = strlen(SteeringWheelString);
pubmsgSteering.qos = QOS;
pubmsgSteering.retained = 0;

deliveredtoken = 0;

MQTTClient_publishMessage(client, TOPICVELOCITY, &pubmsgVelocity, &token);
MQTTClient_publishMessage(client, TOPICSTEERING, &pubmsgSteering, &token);

printf("Waiting for publication of %s  " "on topic %s for client with ClientID: %s\n", VelocityString, TOPICVELOCITY, CLIENTID);
printf("Waiting for publication of %s  " "on topic %s for client with ClientID: %s\n", SteeringWheelString, TOPICSTEERING, CLIENTID);

return rc;
```

Figure D.3: Send and Receive Function

D.2 Unity3D

Developers can select the PreLogic option from the desktop input manager. When this is selected Unity3D connects to an active broker and publishes and subscribes to several values as given in the code. The output values are values received or subscribed by Unity3D whereas the input values are values published by Unity3D on the broker.

```
using System;
using System.Collections;
using System.Collections.Generic;
using u040.prespective.prelogic;
using u040.prespective.prelogic.component;
using u040.prespective.prelogic.signal;
using UnityEngine;

public class TruckBehaviour : PreLogicComponent
{

    public float Velocity;
    public float SteeringAngle;

    public float TruckMarkerPosX;
    public float TruckMarkerPosY;
    public float TruckMarkerPosZ;

    public float TrailerFrontMarkerPosX;
    public float TrailerFrontMarkerPosY;
    public float TrailerFrontMarkerPosZ;

    public float TrailerBackMarkerPosX;
    public float TrailerBackMarkerPosY;
    public float TrailerBackMarkerPosZ;


    public override List<SignalDefinition> SignalDefinitions
    {
        get
        {
            return new List<SignalDefinition>
            {
                //OUTPUT (Look at with respect to IBM Rhapsody)

                #region << OUTPUTS >>
                new SignalDefinition("Velocity", PLCSignalDirection.OUTPUT,
                    SupportedSignalType.REAL32, _onValueChange:OnOutputChange
                    , _baseValue:0f),
                new SignalDefinition("SteeringAngle", PLCSignalDirection.
                    OUTPUT, SupportedSignalType.REAL32, _onValueChange:
                    OnOutputChange, _baseValue:0f),
                 #endregion
```

```
                //INPUTS (Look at with respect to IBM Rhapsody)

                 #region << INPUTS >>
                new SignalDefinition("TruckMarkerPosX", PLCSignalDirection.
                    INPUT, SupportedSignalType.REAL32,  _baseValue:0f),
                new SignalDefinition("TruckMarkerPosY", PLCSignalDirection.
                    INPUT, SupportedSignalType.REAL32, _baseValue:0f),
                new SignalDefinition("TruckMarkerPosZ", PLCSignalDirection.
                    INPUT, SupportedSignalType.REAL32, _baseValue:0f),

                new SignalDefinition("TrailerFrontMarkerPosX",
                    PLCSignalDirection.INPUT, SupportedSignalType.REAL32,
                    _baseValue:0f),
                new SignalDefinition("TrailerFrontMarkerPosY",
                    PLCSignalDirection.INPUT, SupportedSignalType.REAL32,
                    _baseValue:0f),
                new SignalDefinition("TrailerFrontMarkerPosZ",
                    PLCSignalDirection.INPUT, SupportedSignalType.REAL32,
                    _baseValue:0f),

                new SignalDefinition("TrailerBackMarkerPosX",
                    PLCSignalDirection.INPUT, SupportedSignalType.REAL32,
                    _baseValue:0f),
                new SignalDefinition("TrailerBackMarkerPosY",
                    PLCSignalDirection.INPUT, SupportedSignalType.REAL32,
                    _baseValue:0f),
                new SignalDefinition("TrailerBackMarkerPosZ",
                    PLCSignalDirection.INPUT, SupportedSignalType.REAL32,
                    _baseValue:0f),
                #endregion
            };

    }

}


    private void OnOutputChange(SignalInstance _inst, object _newValue,
        DateTime _newChangeTime, object _oldValue, DateTime _oldChangeTime)
    {
        Debug.Log(_inst.definition.defaultSignalAddress + " " + _newValue.
            ToString());

        if(_inst.definition.defaultSignalAddress=="Velocity")
        {
            Velocity = (float) _newValue;
        }

        if (_inst.definition.defaultSignalAddress == "SteeringAngle")
        {
```

```
            SteeringAngle = (float)_newValue;
        }
    }

}
```

When the input source is set as PreLogic in the Desktop Input Manager the following code is run by Unity3D. The published ans subscribed values are bind-ed with the actual truck velocity, steering angle and positions of the truck.

```
if (inputSource == InputSource.PreLogic)
                {
                    vertical = TruckControls.Velocity;
                    horizontal = TruckControls.SteeringAngle;

                    vehicleController.input.Horizontal = horizontal;
                    vehicleController.input.Vertical = vertical;

                    // Truck Aruco Marker XYZ Positions
                    TruckControls.TruckMarkerPosX = truck_marker_pos.x;
                    TruckControls.WriteValue("TruckMarkerPosX",
                        truck_marker_pos.x);

                    TruckControls.TruckMarkerPosY = truck_marker_pos.y;
                    TruckControls.WriteValue("TruckMarkerPosY",
                        truck_marker_pos.y);

                    TruckControls.TruckMarkerPosZ = truck_marker_pos.z;
                    TruckControls.WriteValue("TruckMarkerPosZ",
                        truck_marker_pos.z);



                     // Front Trailer Aruco Markers XYZ positions
                     TruckControls.TrailerFrontMarkerPosX=
                         trailer_front_marker_pos.x;
                     TruckControls.WriteValue("TrailerFrontMarkerPosX",
                         trailer_front_marker_pos.x);

                     TruckControls.TrailerFrontMarkerPosY =
                         trailer_front_marker_pos.y;
                     TruckControls.WriteValue("TrailerFrontMarkerPosY",
                         trailer_front_marker_pos.y);

                     TruckControls.TrailerFrontMarkerPosZ =
                         trailer_front_marker_pos.z;
                     TruckControls.WriteValue("TrailerFrontMarkerPosZ",
                         trailer_front_marker_pos.z);
```

```
//Front Aruco Markers XYZ positions
TruckControls.TrailerBackMarkerPosX =
    trailer_back_marker_pos.x;
TruckControls.WriteValue("TrailerBackMarkerPosX",
    trailer_back_marker_pos.x);

TruckControls.TrailerBackMarkerPosY =
    trailer_back_marker_pos.y;
TruckControls.WriteValue("TrailerBackMarkerPosY",
    trailer_back_marker_pos.y);

TruckControls.TrailerBackMarkerPosZ =
    trailer_back_marker_pos.z;
TruckControls.WriteValue("TrailerBackMarkerPosZ",
    trailer_back_marker_pos.z); }
```