IoT Modelling for Digital Twins

A Model-Driven Engineering Approach to Architecting Digital Twin IoT Systems

Description

The CAPS Modeling Framework was created to support the engineering of **Situational Aware Cyber-Physical Systems (SiA-CPS)**. SiA-CPS are systems consisting of a set of IoT devices such as sensors, cameras, RFID, and other monitoring tools used to continuously observe given indoor and outdoor spaces. The data gathered during this process is then transformed into actionable insights.

SiA-CPS systems require monitoring in both time and space, meaning they consist of:

- **Software components** (drivers and code running on the sensors)
- **Hardware components** (sensors, cameras, RFID, etc.)
- **Environmental interactions** (how software and hardware components interact with their surroundings)

Designing such systems poses challenges in ensuring consistency between managing software, hardware, and environmental views. To address this, the CAPS Modeling Framework provides a **multi-view approach** based on the IEEE/ISO/IEC 42010 standard, ensuring structured and coherent system modeling.

For more information, visit: **https://caps.disim.univaq.it/**

CupCarbon is a comprehensive simulator designed for modeling, visualizing, and analyzing **smart city** and **Internet of Things (IoT)** wireless sensor networks (WSNs). It enables users to create and simulate networks directly on geographic maps, allowing realistic placement of nodes and environmental conditions. With support for static and mobile nodes, energy consumption modeling, routing, and communication protocols, CupCarbon provides a powerful environment for testing IoT applications before physical deployment.

Researchers, educators, and developers use CupCarbon to experiment with network behaviors, evaluate communication strategies, and validate system architectures in simulated urban or rural environments. Its scripting language (**Senscript**) allows users to program node behaviors, making it highly customizable for a wide range of IoT and WSN projects.

Learn more and download CupCarbon at: **https://cupcarbon.com/**

Parallel Discrete Event System Specification (PDEVS) is an extension of the DEVS (Discrete Event System Specification) formalism, providing a structured approach to modeling and simulating complex discrete event systems. PDEVS enhances traditional DEVS by introducing **parallel event handling**, allowing multiple simultaneous events to be processed in a single simulation step. This improves efficiency and makes it suitable for simulating large-scale, interconnected systems.

PDEVS models a system as a set of components (atomic and coupled models) that interact through well-defined input, output, and state transition functions. Each component responds to events over simulated time, enabling hierarchical, modular, and reusable system representations.

PDEVS is widely used in fields such as **IoT, smart cities, cyber-physical systems, logistics, and distributed systems** to simulate dynamic, event-driven behaviors before implementation. It supports both theoretical

analysis and practical software simulation frameworks like PyDEVS, MS4 Me, and DEVSJAVA.

To learn more, visit: **https://en.wikipedia.org/wiki/Discrete_Event_System_Specification**

https://msdl.uantwerpen.be/documentation/PythonPDEVS/

Installation

Installation

CAPS Installation Guide

Prerequisites

Before proceeding with the installation, ensure that your system meets the following requirements:

- Windows operating system (64-bit)(Tried on Windows expecting same will work for other systems as well)
- Internet connection
- Sufficient disk space (at least 5GB free)

Step 1: Download and Install Java 8

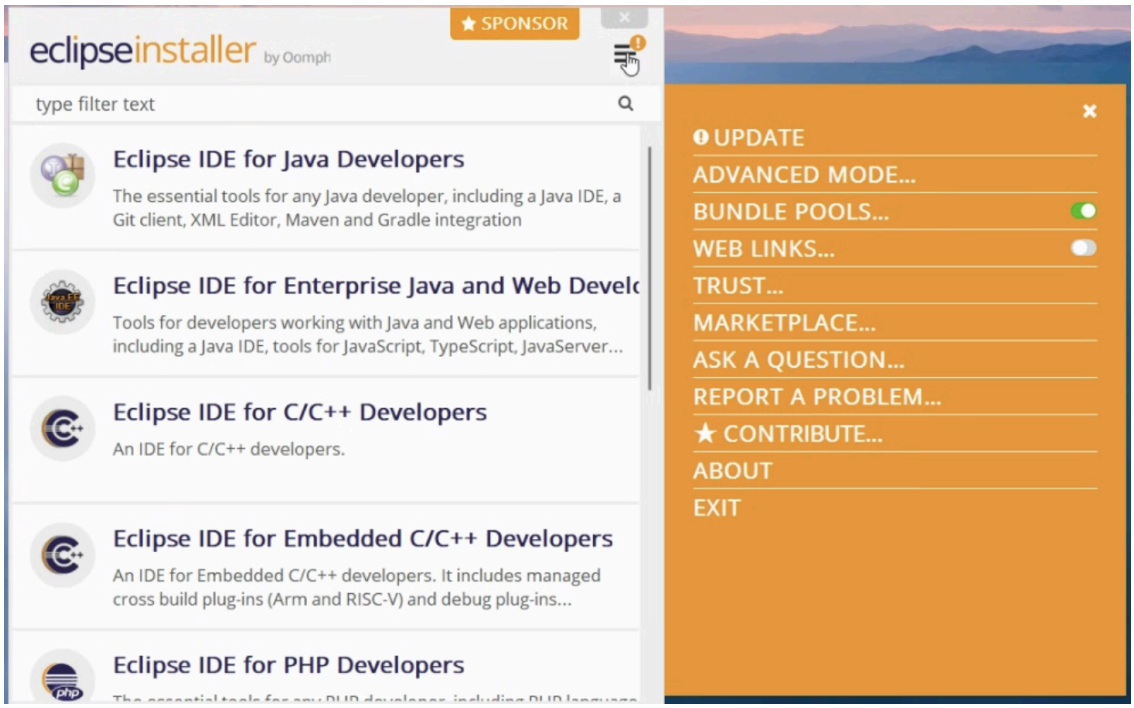
1. Visit the Java SE 8 Downloads page.
2. Accept the license agreement and download the Windows x64 JRE.
3. Run the installer and follow the on-screen instructions.
4. Verify the installation by opening the Command Prompt (`Win + R`, type `cmd`, and press Enter) and running:

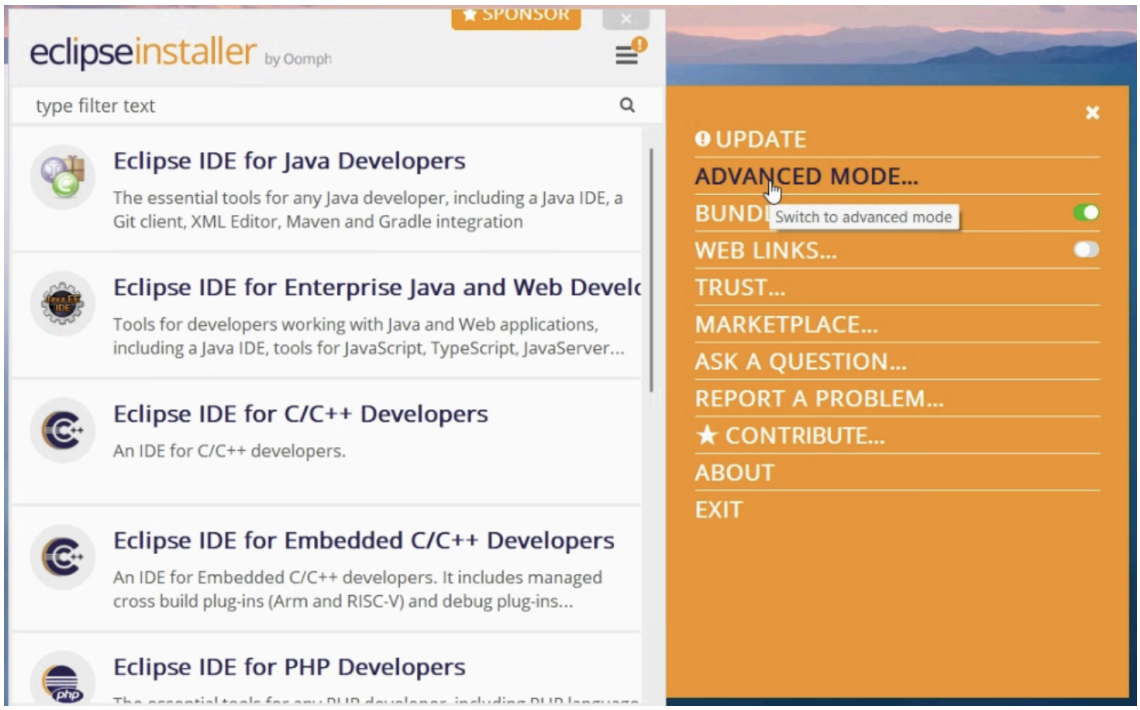
   ```
   java -version
   ```

   Ensure it displays Java 1.8.

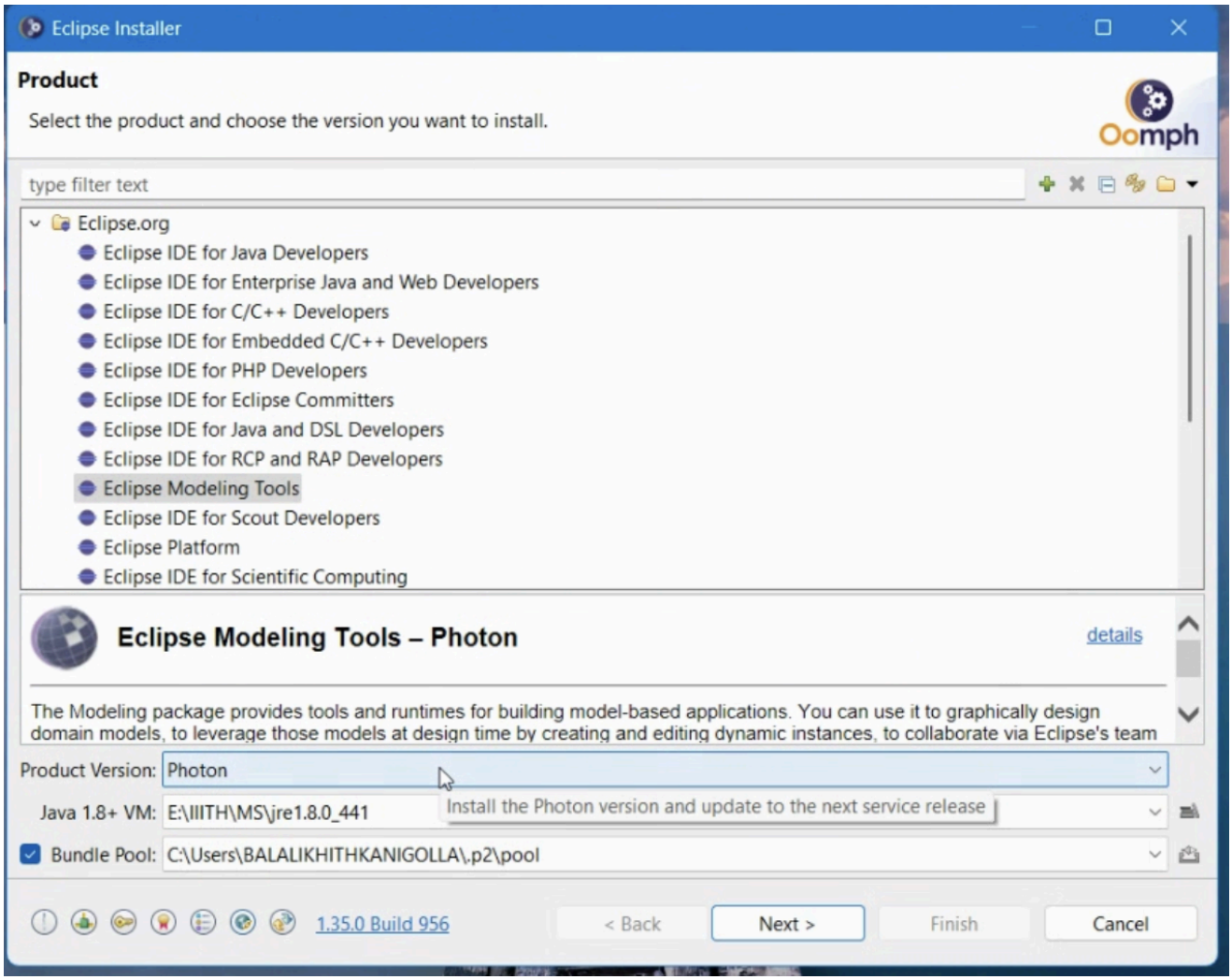
Step 2: Download and Install Eclipse Modeling Tools

1. Visit the Eclipse download page.
2. Download the Eclipse Installer for Windows (64-bit).
3. Run the installer and select **Advanced Mode**.





4. Choose **Eclipse Modeling Tools** as the package.
5. Select the Product version as Photon and proceed with the installation and select the Java 1.8



6. Once installed, launch Eclipse.

Step 3: Install EMF

1. Open Eclipse and go to `Help` → `Install New Software`.

2. In the **Work with:** field, select **All Available Sites**.
3. Search for **EMF - Eclipse Modeling Framework**.
4. Check the box for **Eclipse Modeling Framework SDK** and proceed with the installation.
5. Restart Eclipse when prompted.

Step 4: Install GMF Tooling

1. Open Eclipse and navigate to `Help` → `Install New Software`.
2. Click on `Add...` and enter the following:
   - Name: **GMF Tooling**
   - Location: `https://download.eclipse.org/epsilon/updates/gmf-tooling/`
3. Select **GMF Tooling** and click `Next`.
4. Complete the installation and restart Eclipse.

Step 5: Install Epsilon

1. Open Eclipse and go to `Help` → `Install New Software`.
2. Click `Add...` and enter:
   - Name: **Epsilon**
   - Location: `https://download.eclipse.org/epsilon/updates/`
3. Select all the Epsilon components needed and proceed with the installation.
4. Restart Eclipse when prompted.

Final Verification

After installation, verify that:

- Java 8 is installed (`java -version` In Command Prompt)
- Eclipse Modeling Tools launches without errors
- EMF, GMF Tooling, and Epsilon are installed under `Help` → `About Eclipse` → `Installation Details`

This completes the setup process. You are now ready to use Eclipse for modeling and development!

Installation

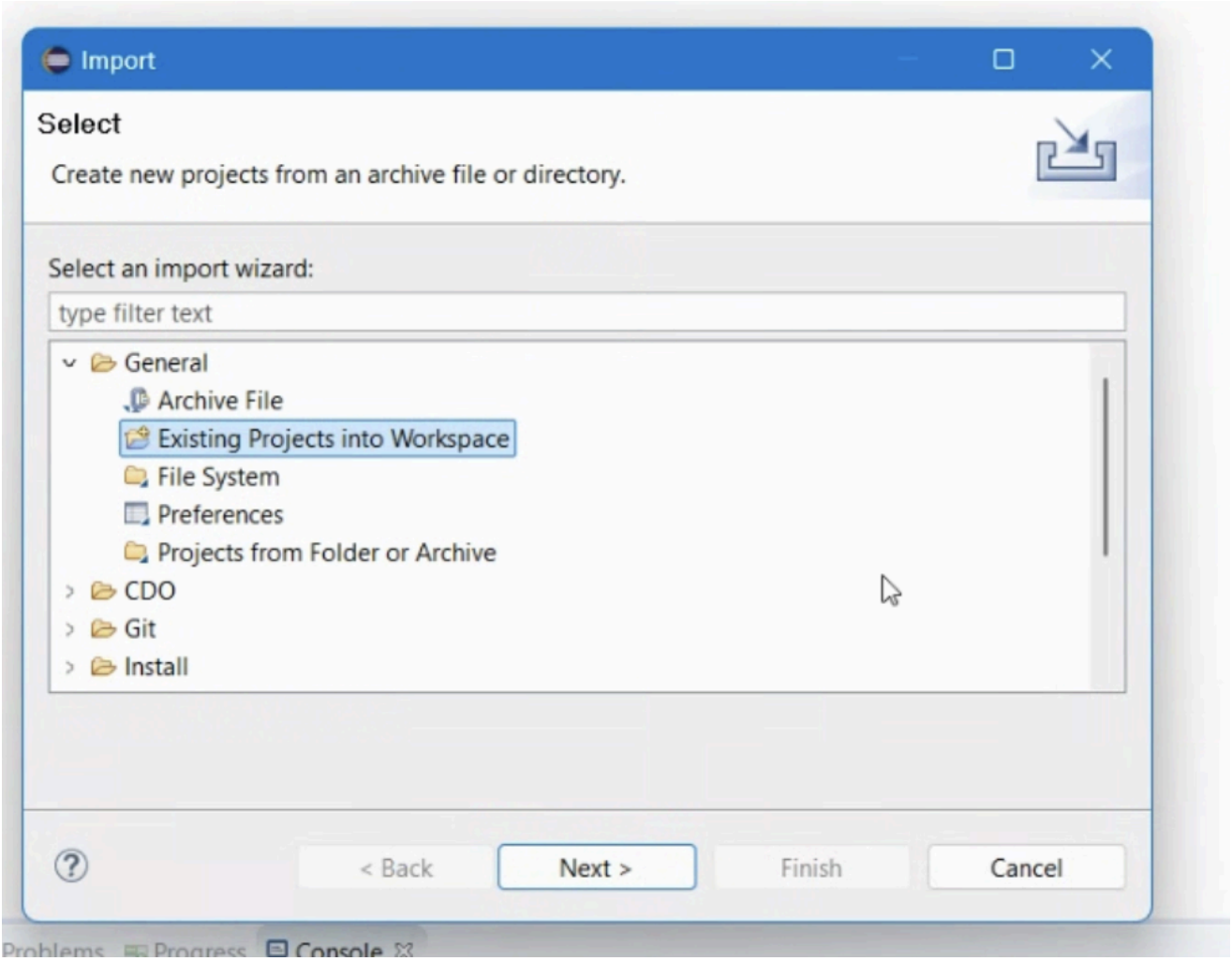
Eclipse CAPS Integration Guide

Step 1: Download the CAPS Project Files

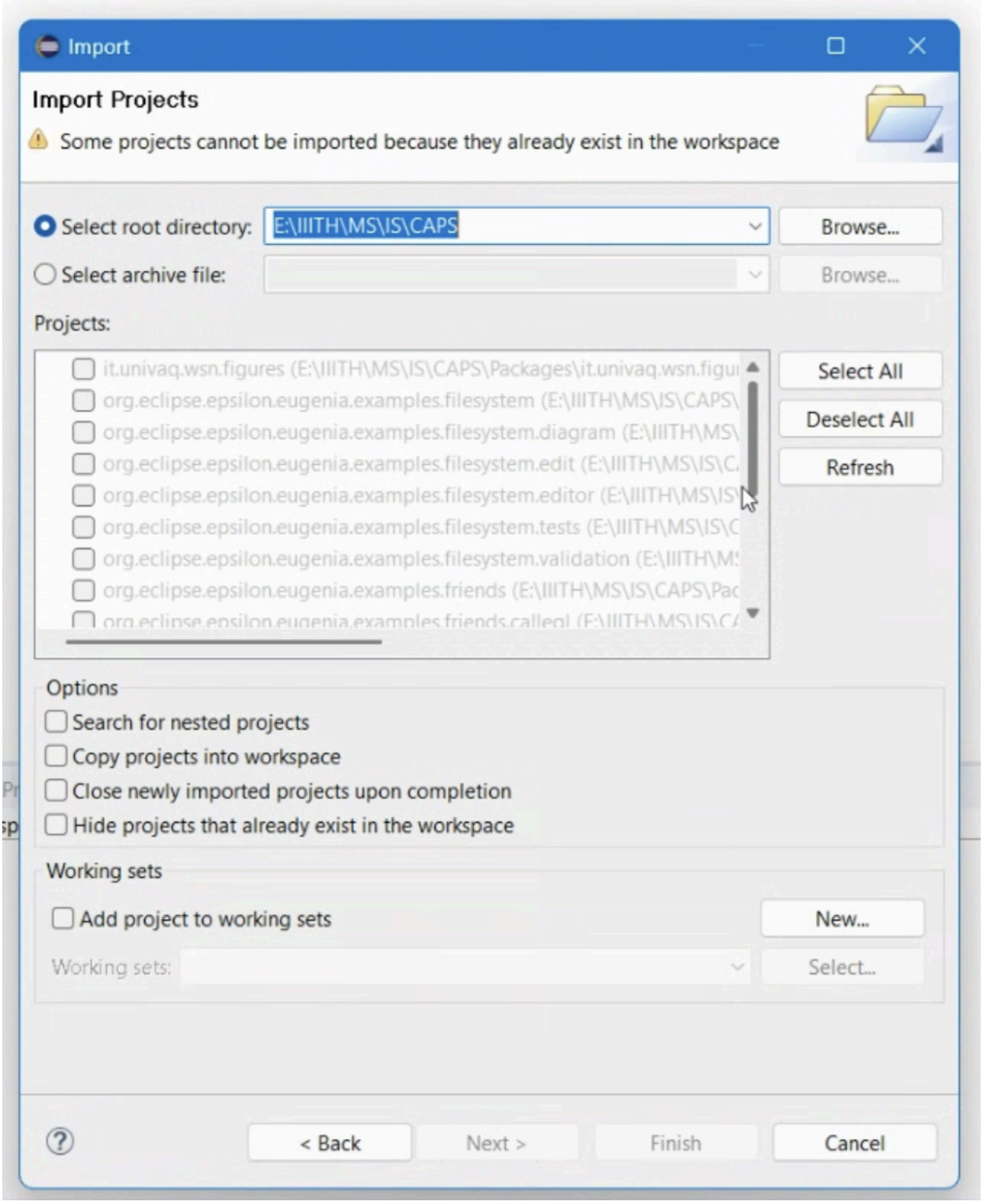
1. Obtain the latest CAPS project archive (`CAPS.7z`) from the official repository or provided link: Download CAPS.7z
2. Save the file to a preferred location (e.g., `C:\CAPS`).
3. Extract the contents of `CAPS.7z` using a tool like WinRAR, 7-Zip, or Windows Explorer.
4. Ensure the extracted folder contains a `Packages` directory with the required Eclipse projects.

Step 2: Import Projects into Eclipse

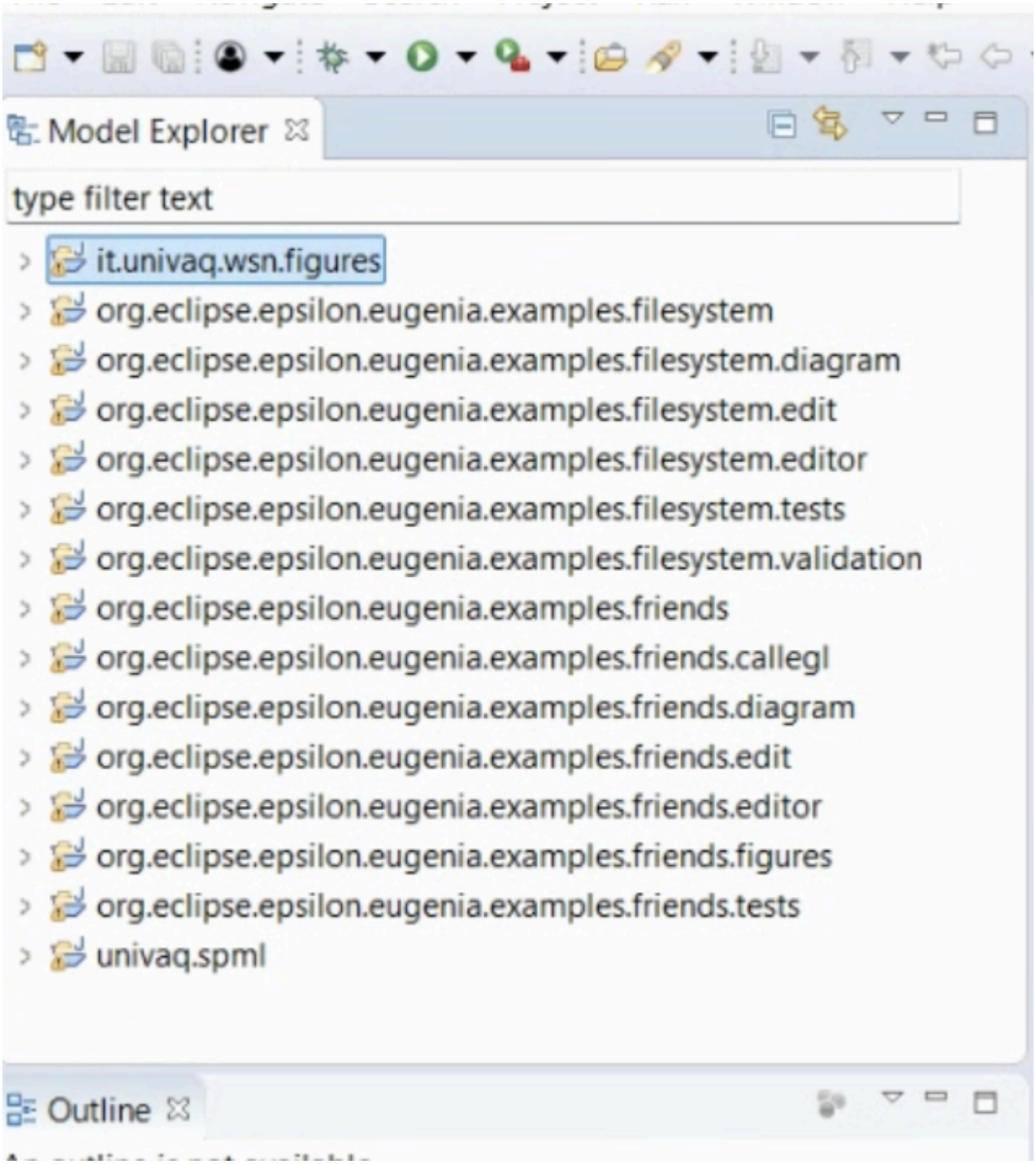
1. Open Eclipse.
2. Navigate to `File` → `Import`.
3. Select `General` → `Existing Projects into Workspace` and click `Next`



4. Click `Browse` and select the extracted `Packages` folder.



5. Ensure all projects are selected in the list.
6. Click `Finish` to import the projects into your workspace.

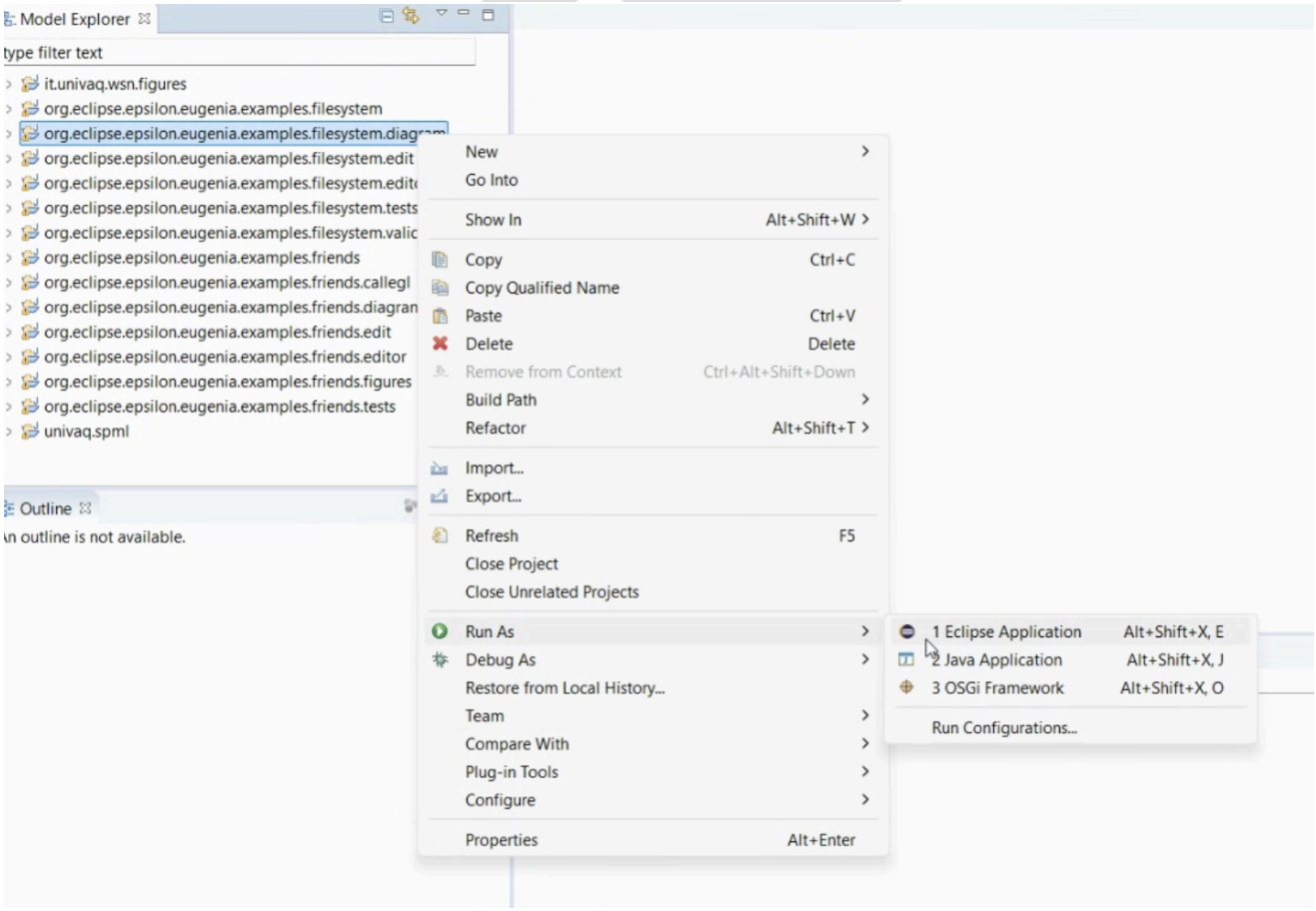


Step 3: Configure the Workspace

Run the Diagram Editor

1. In the `Project Explorer`, locate the project `org.eclipse.epsilon.eugenia.examples.friends.diagram`.

2. Right-click on the project and select `Run As` → `Eclipse Application`.



3. A new Eclipse instance will launch, allowing you to work with the modeling environment.

At this point, the CAPS framework should be successfully imported and ready for use in Eclipse.

Installation

CupCarbon Installation Guide

Step 1: Download CupCarbon

- Go to: https://cupcarbon.com/download_klines.php
- Click the download link for your platform (Windows, Mac, or Linux)

Step 2: Unzip the file

- On Windows: Right-click the downloaded ZIP file → “Extract All”
- On Mac: Double-click the ZIP file to unzip it automatically

Step 3: Run `cupcarbon.jar`

- Open the unzipped folder
- Double-click `cupcarbon.jar`
  If it doesn’t open:
- Open a terminal/command prompt
- Navigate to the folder with `cupcarbon.jar`
- Run:

```
java -jar cupcarbon.jar
```

(You need Java installed; if not, download it from https://www.oracle.com/java/technologies/javase-downloads.html

PythonPDEVS Installation Guide

1. **Prerequisites**
   - Make sure you have **Python 3.x** installed.
   - You may need **pip** (Python package manager).
2. **Clone the repository**

   Open your terminal or command prompt and run:

   ```
   git clone https://github.com/capocchi/PythonPDEVS.git
   ```

3. **Navigate to the project directory**

   ```
   cd PythonPDEVS
   ```

4. **Install PythonPDEVS**

   Run the following command to install it using `setup.py`:

   ```
   python setup.py install
   ```

   Alternatively, if you prefer installing in **development mode** (helpful if you plan to modify the source):

   ```
   python setup.py develop
   ```

SAML

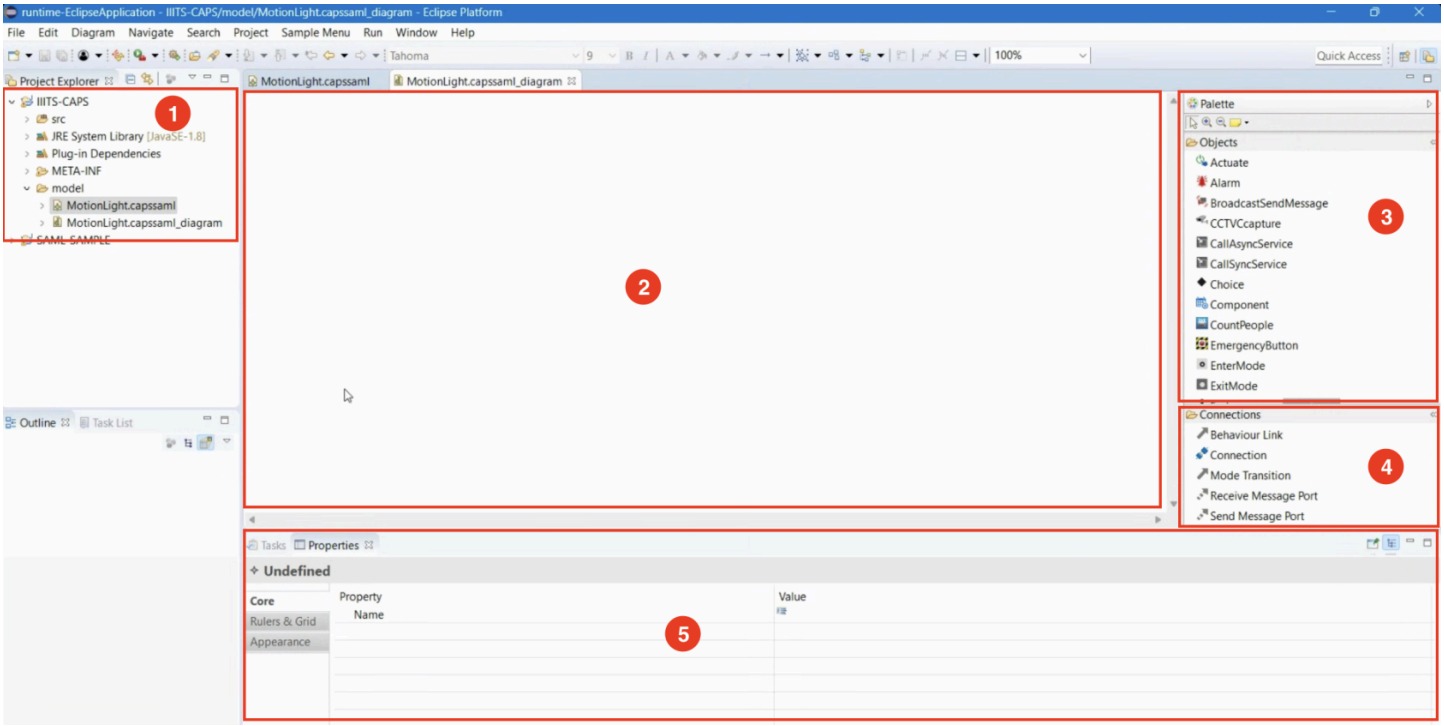
SAML

SAML (Software Architecture Modeling Language) Guide

Overview

SAML (Software Architecture Modeling Language) is used to model and simulate software architectures, particularly in IoT-based scenarios. It allows users to define components, interactions, and behaviors in a structured manner. This guide provides a step-by-step approach to using SAML for designing a temperature sensor-based automation system.

CAPS Environment in Eclipse



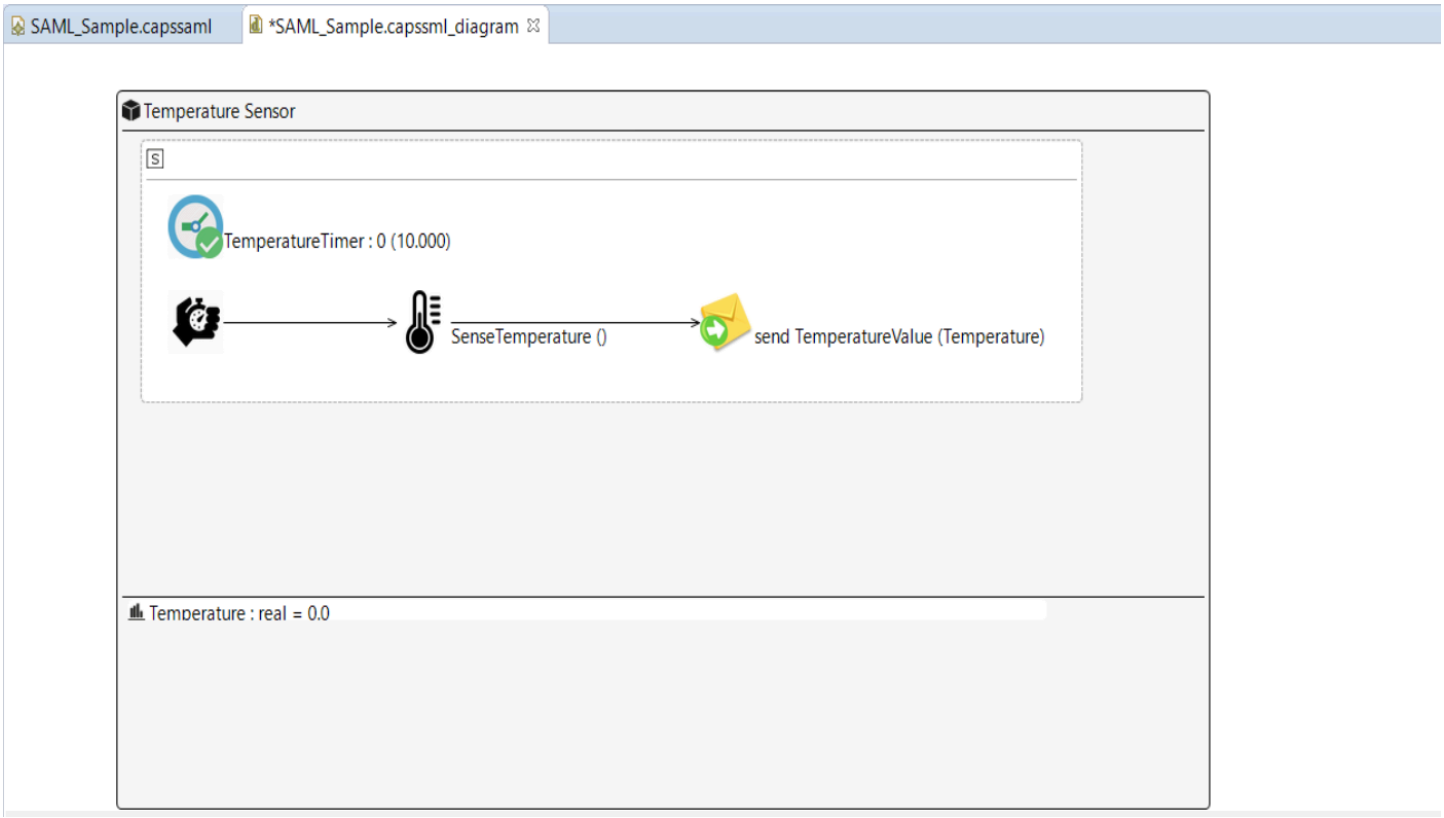
1. Folder Viewer (on the left)
2. Diagram Canvas (in the center)
3. Pallate(on the right)
4. Connections(on the right)
5. Properties Panel(at the bottom)

Folder Viewer

This displays the file structure. It contains your SAML project, and there is a folder called ‘model’ where you can create your SAML models.

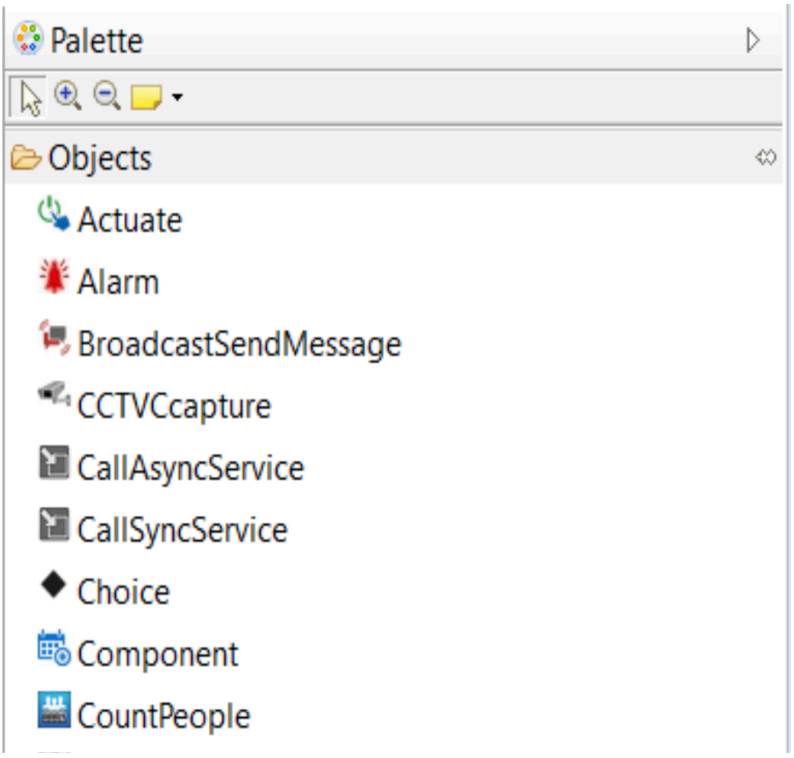
Diagram Canvas

Items can be taken from the palette and connections to model our SAML.



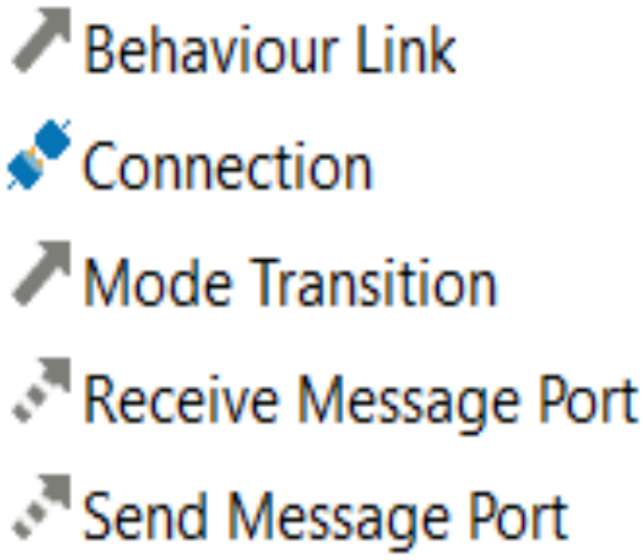
Pallate

Pallate contains all the components required for the architecture, such as sensors, timers etc



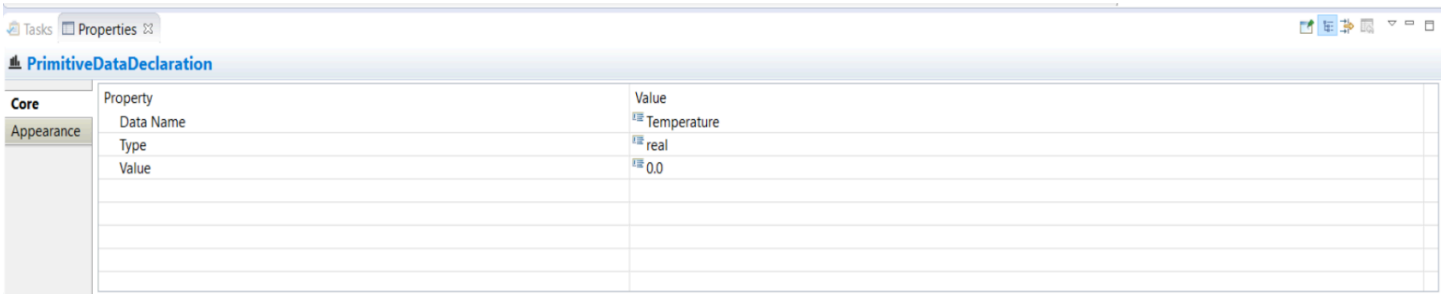
Connections

Types of connections between the components can be selected from here.



Properties

Configure the properties of the components used for modelling,



SAML

SAML for Motion Based Lighting System

Project Description

This project implements an automated motion detection and response system using a sensor-controller-actuator architecture. A **motion sensor** continuously monitors for movement within its detection range. Upon detecting motion, the sensor transmits a signal to a **controller** (or router) responsible for processing the incoming data. The controller analyzes the signal and, based on predefined logic, sends a command to a **light actuator** to turn the light on. This system enables responsive, energy-efficient lighting by automating illumination based on occupancy or movement detection, making it ideal for smart home or smart building applications.

Motion Sensor

1. Detects Motion every second
2. Sends the data to the controller/router

Controller/Router

1. Waits for the message to receive
2. Look for the available nodes/lights
3. Sends the message

Sensor/Light

1. Receives the message.
2. checks if it's 1 or 0.
3. Turn on/off the light based on the condition.

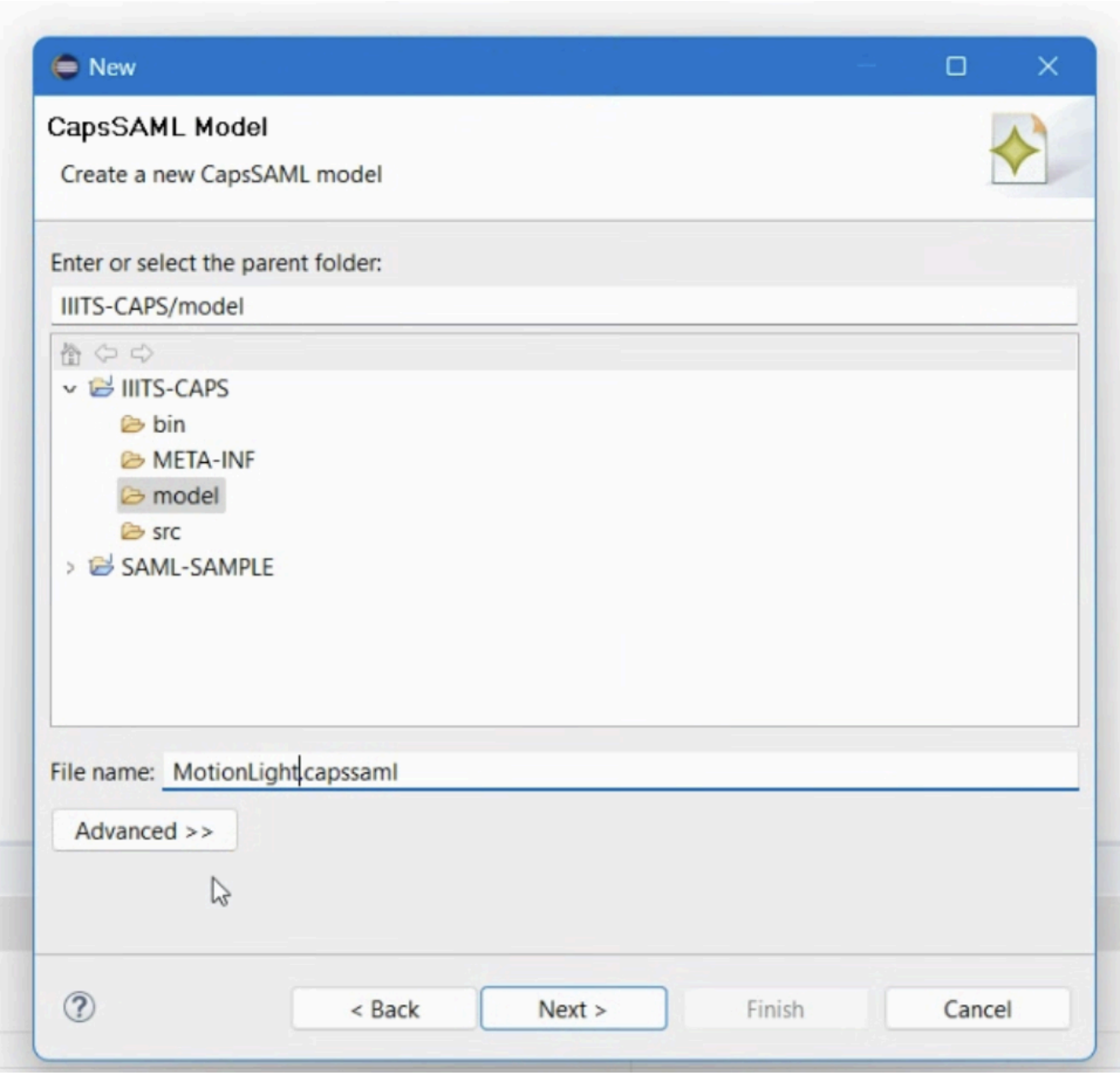
Steps to Create a SAML Project

1. Create a New SAML Project

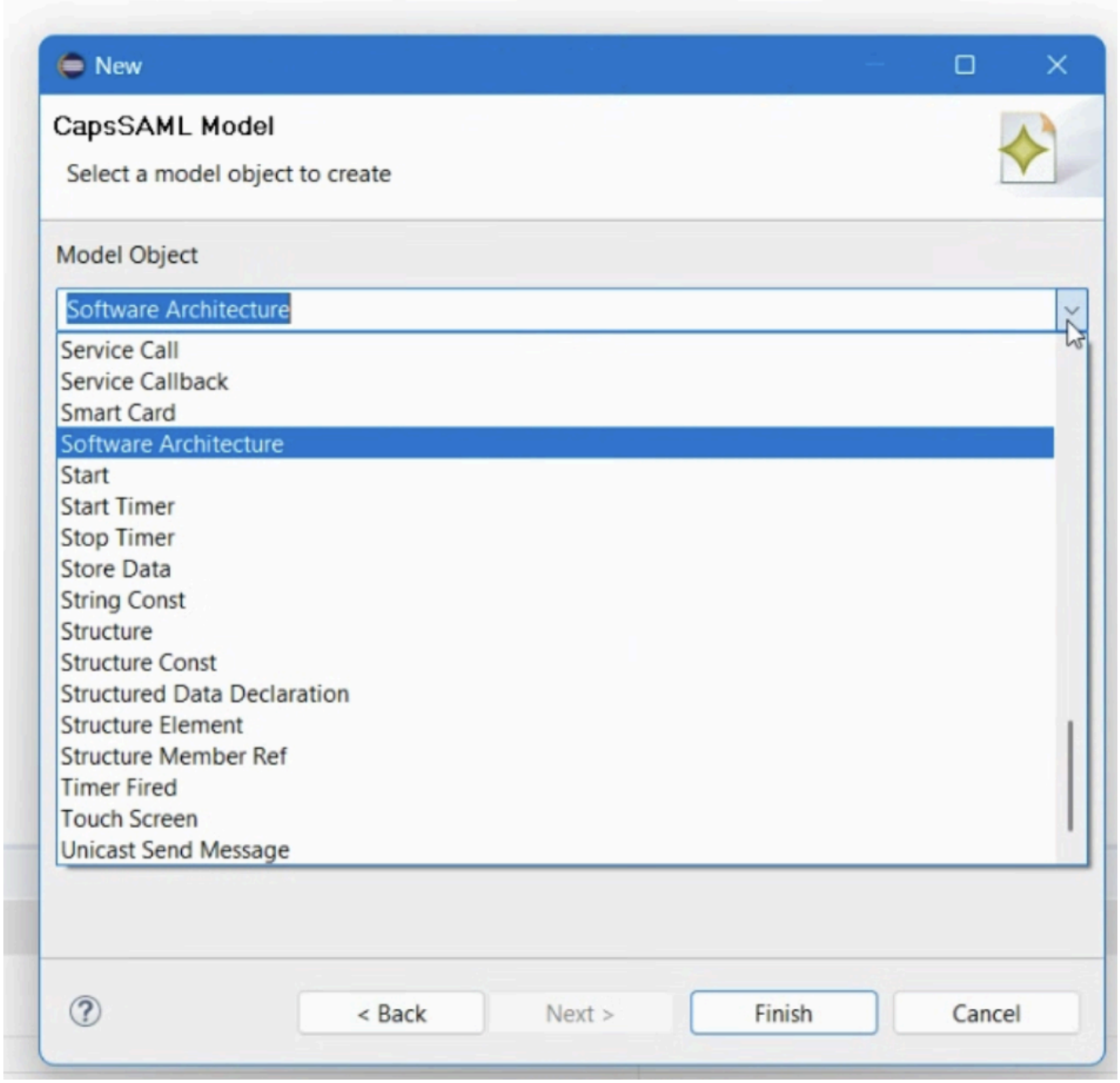
1. Open Eclipse and go to `File` → `New` → `Project`.
2. Select `EMF` → `Empty EMF Project`.
3. Name the project and select a location (keeping the default workspace location is recommended).
4. Click `Finish`.

2. Create the SAML Model File

1. In the **Project Explorer**, expand the newly created project.
2. Right-click on the `model` folder, select `New` → `Other`.
3. Search for `CAPSModel` and select `CAPSSaml`.
4. Name the file, ensuring it ends with `.capssaml`



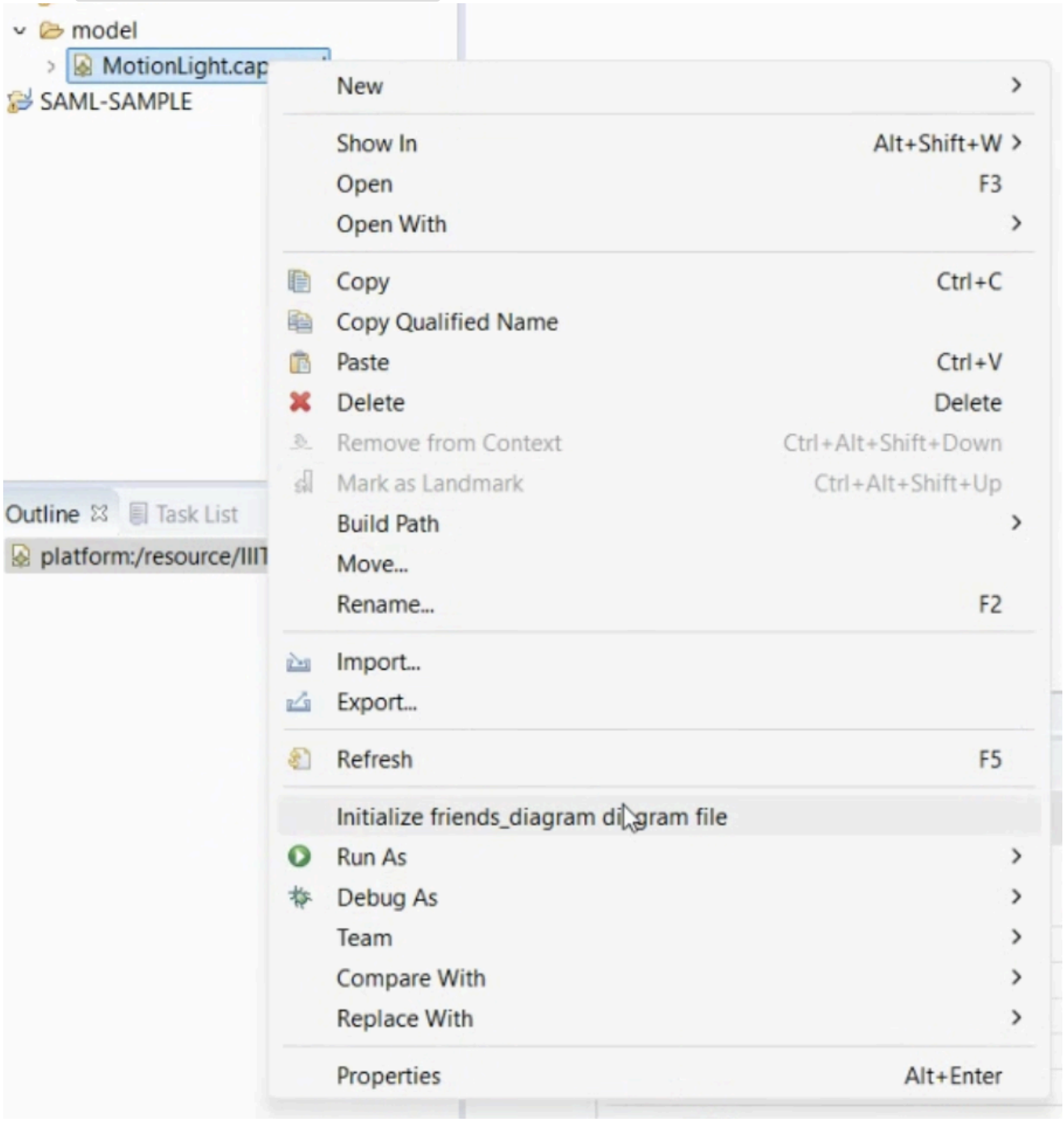
5. Click `Next`, select `Software Architecture` from the model object list, and click `Finish`.



3. Initialize the Diagram

1. In the **Project Explorer**, right-click on the new `.capssaml` file.

2. Select `Initialize friends diagram file`.

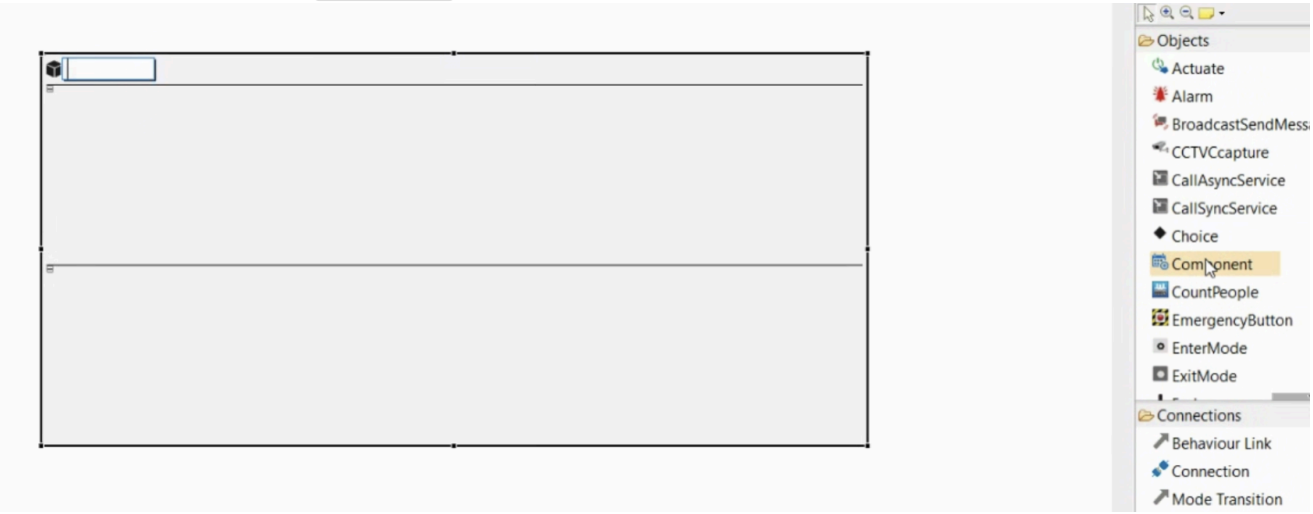


3. Keep the default name and location, and click `Finish`.
4. The diagram editor will open with an empty canvas.

4. Creating Components

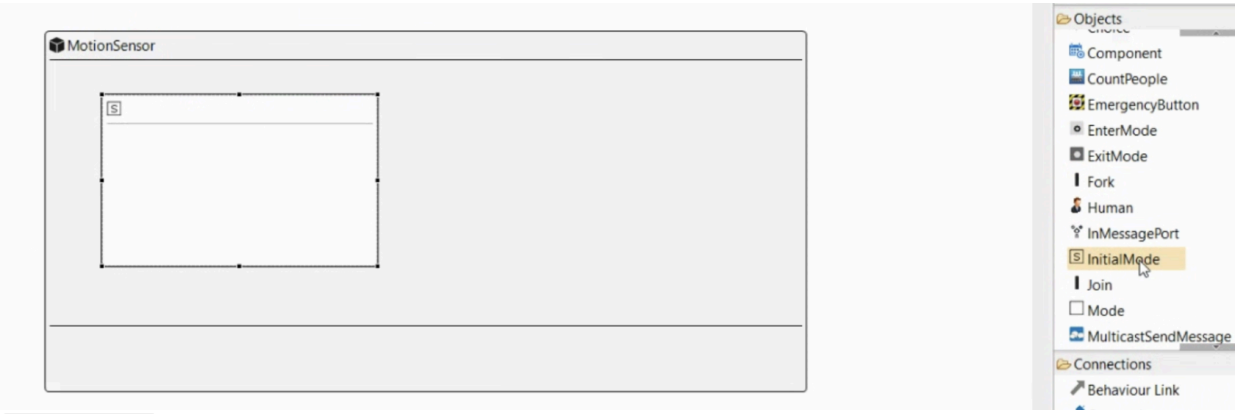
1. Motion Sensor:

- From the palette, select `Component` and place it on the canvas.

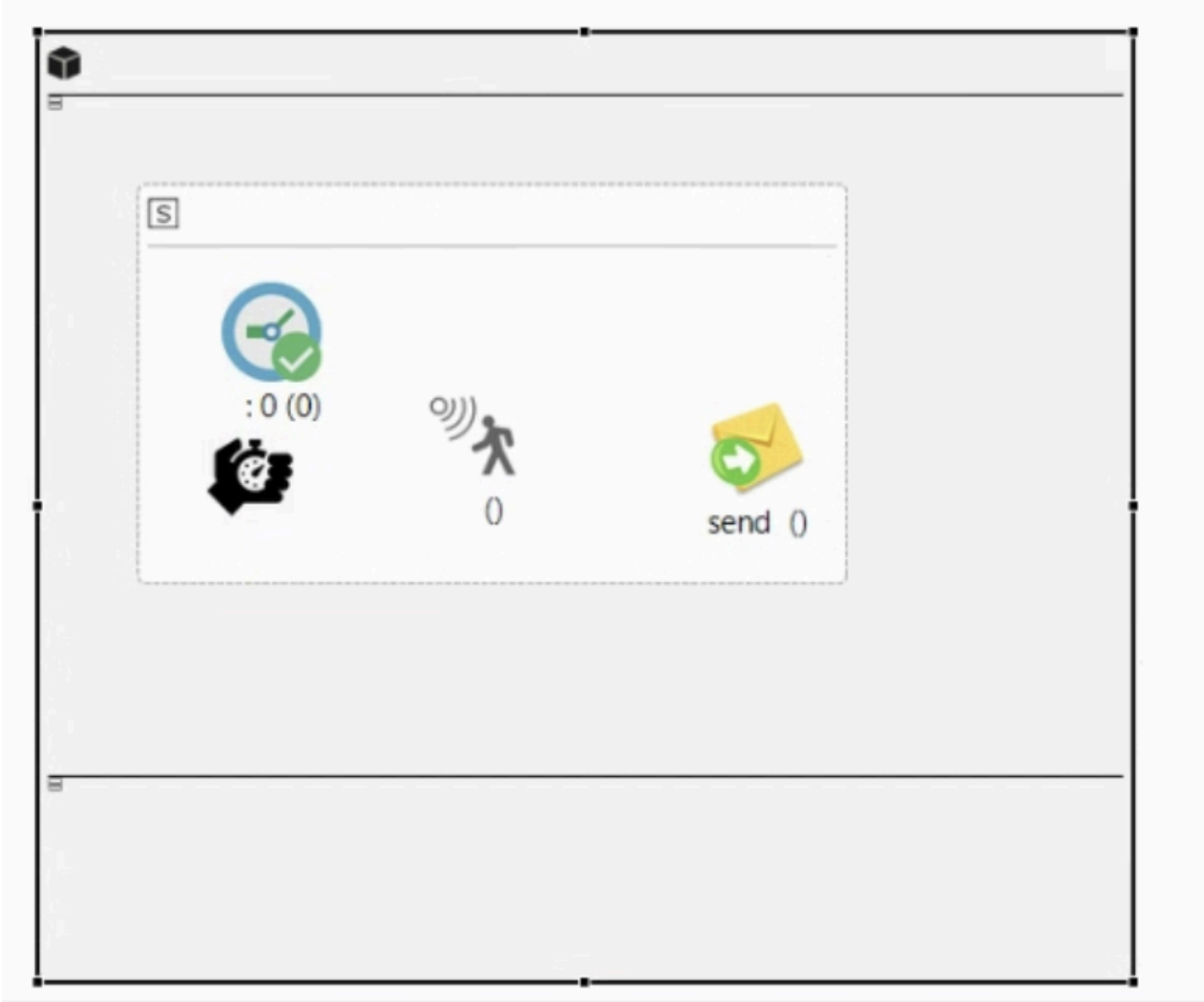


- In the **Properties** view, name it `MotionSensor`.

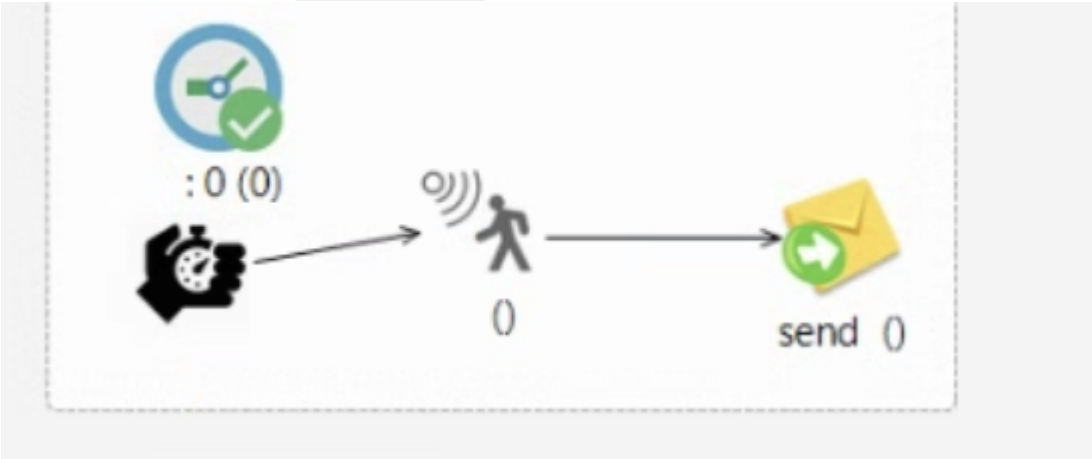
- Inside the component, add:
  - `Initial Mode` (to define its starting behavior).



  - `StartTimer` (to trigger periodic temperature readings).
  - `SenseOccupancy` (to simulate temperature sensing).
  - `TimerFired` (to handle the timer expiration event).
  - `UnicastSendMessage` (to send the motion data).



- Connect these elements using `Behavior Link`.

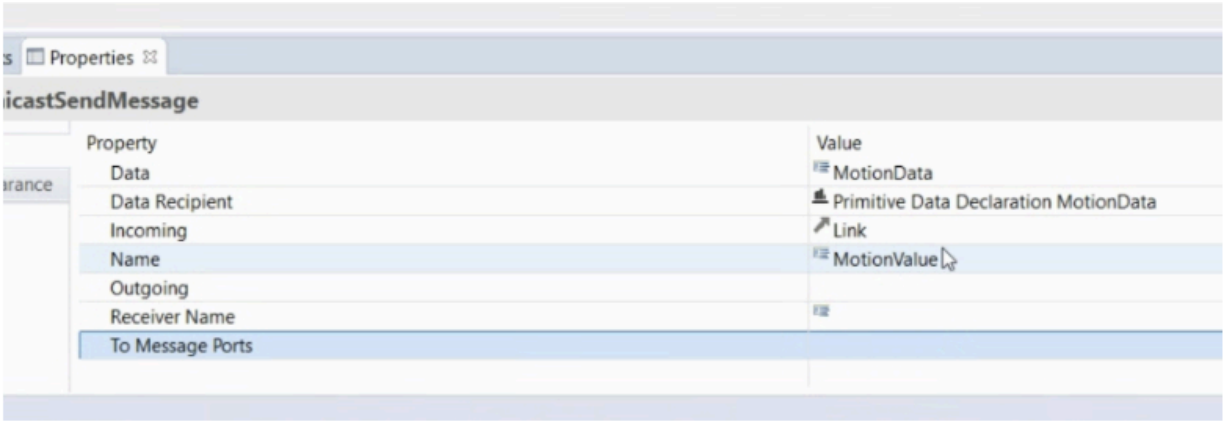


- Set **StartTimer Properties**:
  - `Cyclic` → `True`
  - `Name` → `MotionTimer`
  - `Period` → `1000` (milliseconds, i.e., every 1 second)

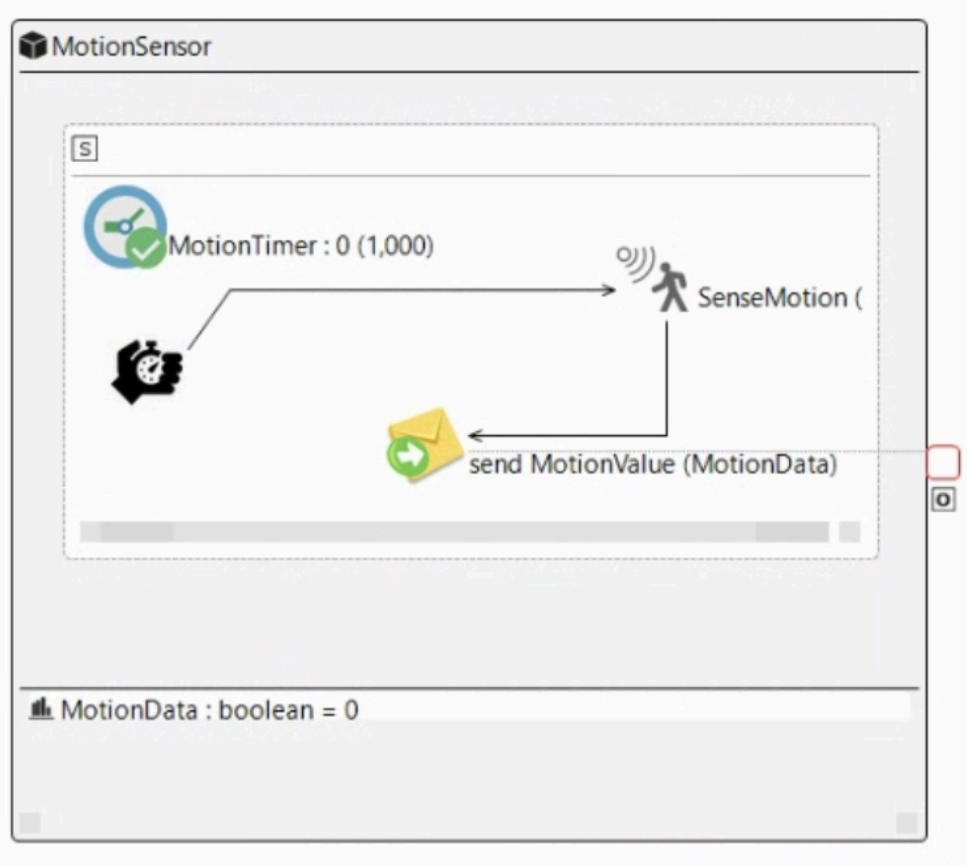
- Define `Primitive Data Declaration`:
  - `Name` → `MotionData`
  - `Type` → `Boolean`
  - `Value` → `0`
-

- Select the `SenseOccupancy` item and, in the properties view, set the following parameters.
  - Data Recipient →Double click and select the Motion Data
  - Name →SenseMotion

- Select the message item and set the following parameters:
  - Data →MotionData;
  - Data Recipient →Select the Primitive Data Declaration MotionData variable;
  - Name →MotionValue

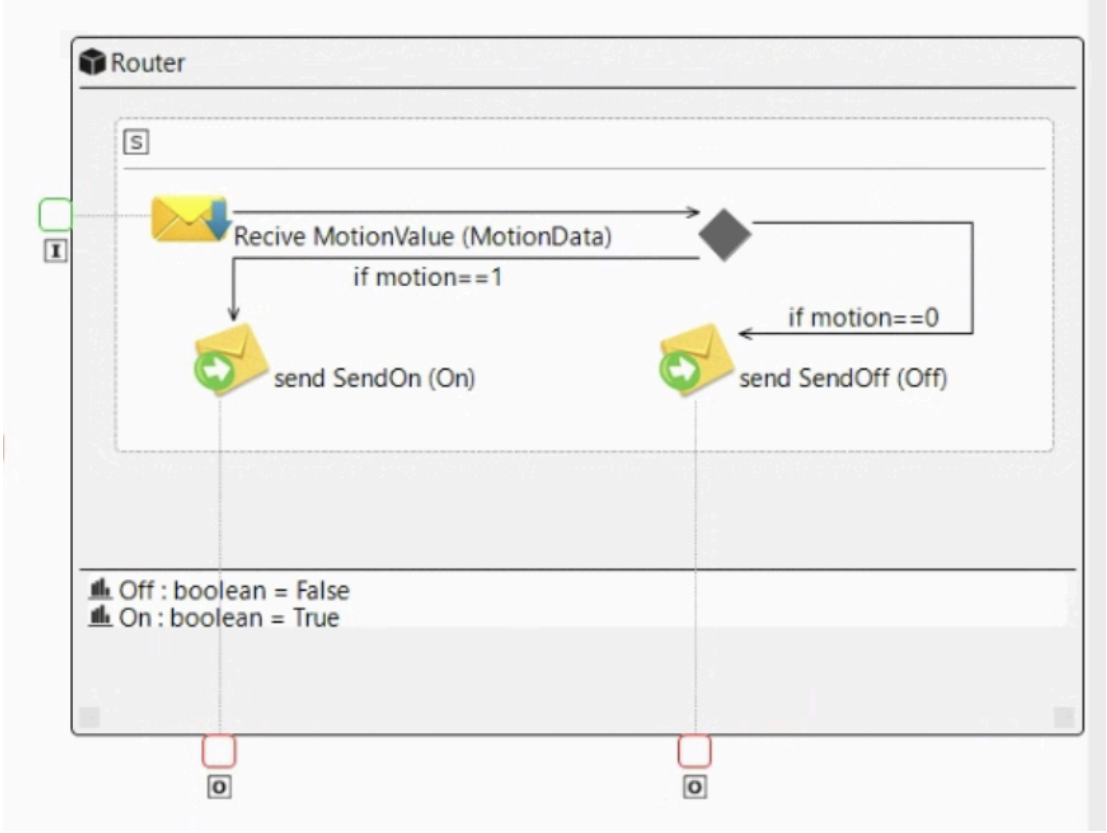


Now add one OutMessagePort and connect the send MotionValue to the outport using the Send Message Port. You will see something similar to this at this moment:



2. Controller/Router

- Create a `Component` named `Controller`.
- Inside, add:
  - `Initial Mode`
  - `ReceiveMessage` (to get motion data from the sensor)
  - `Choice` (to implement decision-making logic)
  - Two `UnicastSendMessage` instances (to send `On` or `Off` commands)
- Define `Primitive Data Declaration`:
  - `Name` → `Off`, `Type` → `Boolean`, `Value` → `False`
  - `Name` → `On`, `Type` → `Boolean`, `Value` → `True`
- Select the message item and set the following parameters for one:
  - Data →Off;
  - Data Recipient →Select the Primitive Data Declaration Off
  - Name →SendOff
- Select the message item and set the following parameters for second:
  - Data →On;
  - Data Recipient →Select the Primitive Data Declaration On
  - Name →SendOn
- Connect elements with `Behavior Link`:
  - `ReceiveMessage` → `Choice`
  - `Choice` → `SendOnMessage` (if motion==1)
  - `Choice` → `SendOffMessage` (if motion==0)
- Add the InMessagePort, connect to the Receive MotionValue using the ReceiveMessage Port, and 2 OutMessagePorts, and connect them to the UniCastSendMessage



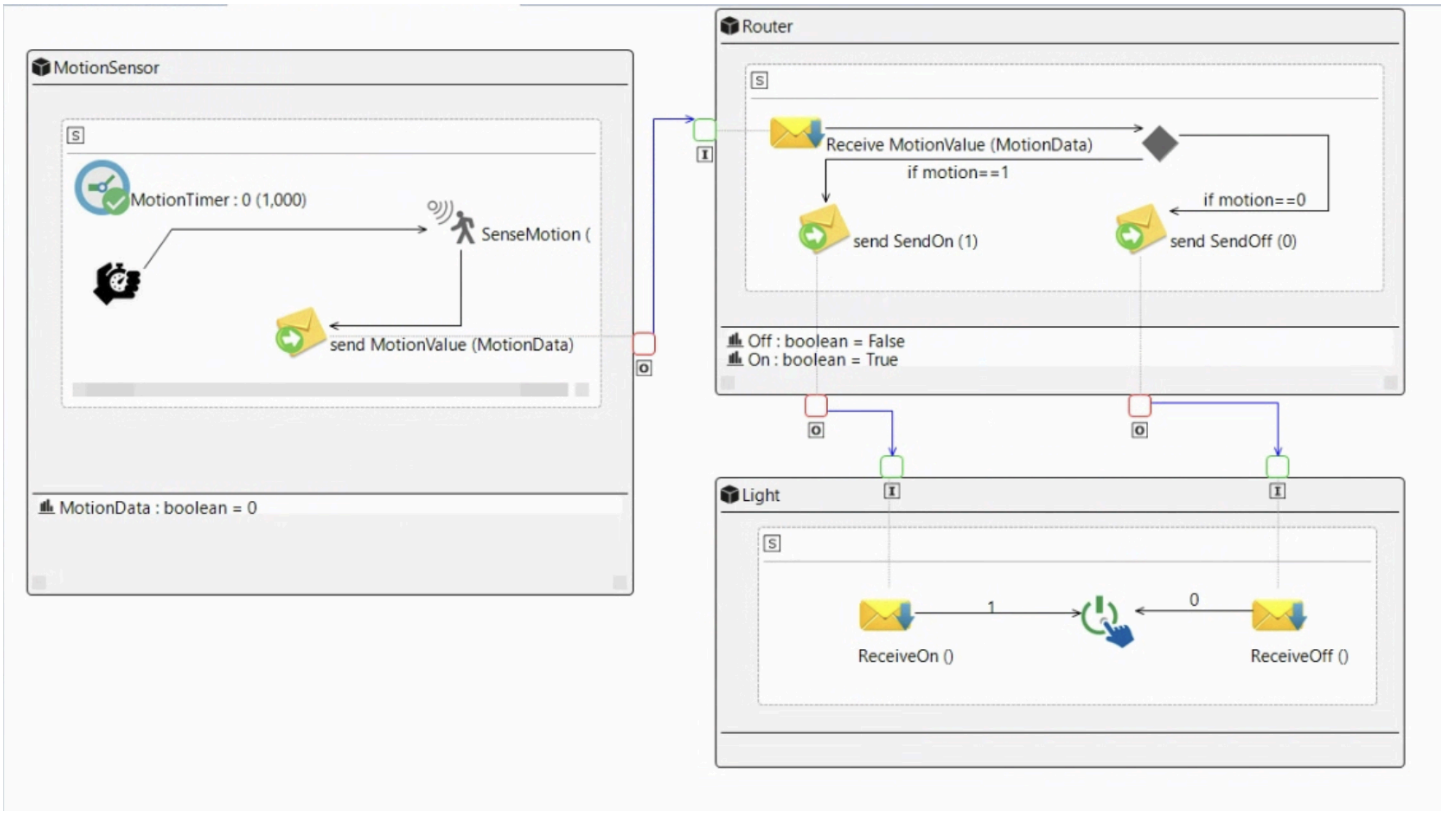
3.Sensor/Light

- Create a `Component` named `Light`.
- ◦ Inside, add:
  - ◦ `Initial Mode`
  - ◦ Two `ReceiveMessage` instances (one for `On`, one for `Off`)
  - ◦ `Actuate` (to control the window mechanism)
- ◦ Connect elements with `Behavior Link`:
  - ◦ `ReceiveOpenMessage` → `Actuate` (if 1 = True)
  - ◦ `ReceiveCloseMessage` → `Actuate` (if 0 = False)
- Now add two InMessagePorts and connect them to the two ReciveMessage using the ReceiveMessagePort

4. Final Connections

- Use `OutMessagePort` and `InMessagePort` to link using the Connection
  - MotionSensor -> Router
  - Router(SendOn) to Light(ReceiveOn)
  - Router(SendOff) to Light(ReceiveOff)

Final Diagram:



Save it and now, open the File in VSCode or in any code editor to see the XML file generated.

SAML

SAML for Temperature based window control

Project Description

We will model a simple scenario: a room equipped with a temperature sensor, an actuator connected to a window, and a server for data storage. The system functions as follows:

- The sensor detects room temperature periodically.
- If the temperature exceeds a threshold, the actuator opens the window.
- If the temperature is too low, the actuator closes the window.
- The temperature data is stored on a remote server.

Temperature Sensor

- Starts a timer to sense temperature every 10 seconds.
- Reads temperature and sends it to the server.

Server

- Receives temperature data from the sensor.
- Stores it in a database.

Controller

- Receives temperature values.
- Decides whether to open or close the window based on thresholds (25°C and 18°C).
- Sends appropriate command (`Open` or `Close`) to the actuator.

Window Actuator

- Receives commands from the controller.
- Opens/closes the window accordingly.

Steps to Create a SAML Project

1. Create a New SAML Project

1. Open Eclipse and go to `File` → `New` → `Project`.
2. Select `EMF` → `Empty EMF Project`.
3. Name the project and select a location (keeping the default workspace location is recommended).

4. Click `Finish`.

2. Create the SAML Model File

1. In the **Project Explorer**, expand the newly created project.
2. Right-click on the `model` folder, select `New` → `Other`.
3. Search for `CAPSModel` and select `CAPSSaml`.
4. Name the file, ensuring it ends with `.capssaml`.
5. Click `Next`, select `Software Architecture` from the model object list, and click `Finish`.

3. Initialize the Diagram

1. In the **Project Explorer**, right-click on the new `.capssaml` file.
2. Select `Initialize friends diagram file`.
3. Keep the default name and location and click `Finish`.
4. The diagram editor will open with an empty canvas.

4. Creating Components

1. **Temperature Sensor**:
   - From the palette, select `Component` and place it on the canvas.
   - In the **Properties** view, name it `TemperatureSensor`.
   - Inside the component, add:
     - `Initial Mode` (to define its starting behavior).
     - `StartTimer` (to trigger periodic temperature readings).
     - `SenseTemperature` (to simulate temperature sensing).
     - `TimerFired` (to handle the timer expiration event).
     - `UnicastSendMessage` (to send the temperature data).
   - Connect these elements using `Behavior Link`.
   - Set **StartTimer Properties**:
     - `Cyclic` → `True`
     - `Name` → `TemperatureTimer`
     - `Period` → `10000` (milliseconds, i.e., every 10 seconds)
   - Define `Primitive Data Declaration`:
     - `Name` → `Temperature`
     - `Type` → `Real`
     - `Value` → `0.0`
2. **Server**:
   - Create a new `Component`, name it `Server`.
   - Inside, add:
     - `Initial Mode`
     - `ReceiveMessage` (to get temperature data from the sensor)
     - `Server` (to process data)
     - `StoreData` (to save the temperature values)
   - Connect `ReceiveMessage` → `Server` → `StoreData` with `Behavior Link`.
3. **Controller**:
   - Create a `Component` named `Controller`.

- Inside, add:
  - `Initial Mode`
  - `ReceiveMessage` (to get temperature data from the sensor)
  - `Choice` (to implement decision-making logic)
  - Two `UnicastSendMessage` instances (to send `Open` or `Close` commands)
- Define `Primitive Data Declaration`:
  - `Name` → `Close`, `Type` → `Boolean`, `Value` → `False`
  - `Name` → `Open`, `Type` → `Boolean`, `Value` → `True`
- Connect elements with `Behavior Link`:
  - `ReceiveMessage` → `Choice`
  - `Choice` → `SendOpenMessage` (if temperature > 25°C)
  - `Choice` → `SendCloseMessage` (if temperature < 18°C)

4. **Window Actuator**:
   - Create a `Component` named `WindowActuator`.
   - Inside, add:
     - `Initial Mode`
     - Two `ReceiveMessage` instances (one for `Open`, one for `Close`)
     - `Actuate` (to control the window mechanism)
   - Connect elements with `Behavior Link`:
     - `ReceiveOpenMessage` → `Actuate` (if Open = True)
     - `ReceiveCloseMessage` → `Actuate` (if Close = False)

5. Connecting Components

- Use `OutMessagePort` and `InMessagePort` to link:
  - `TemperatureSensor` → `Server`
  - `Server` → `Controller`
  - `Controller` → `WindowActuator`

HWML

Hardware Modeling Language (HWML) Guide

Introduction to HWML

The Hardware Modeling Language (HWML) is a framework designed to model the hardware components of a system. It allows users to define deployment nodes, microcontrollers, processors, memory, energy sources, and various sensors. This guide provides a detailed step-by-step process for modeling hardware using HWML.

Step 1: Creating a New HWML Model

1. Open Eclipse and navigate to `Project Explorer`.
2. Right-click on the `model` folder.
3. Select `New` → `Other`.
4. In the dialog box, search for **CAPShwml Model**.
5. Click `Next`, then provide a name for the model (ensure it ends with `.capshwml`).
6. Click `Next` again and choose **Node Specification**.
7. Click `Finish` to create the HWML model file.

Step 2: Initializing the HWML Diagram

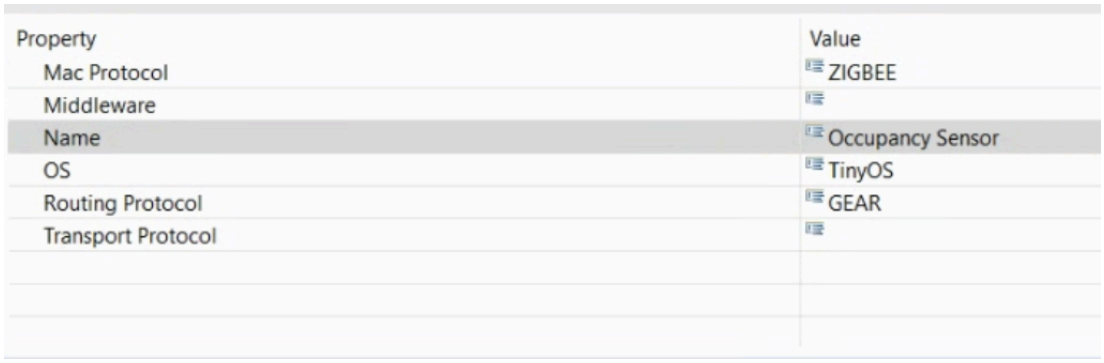
1. Locate the newly created `.capshwml` file in the `Project Explorer`.
2. Right-click the file and select `Initialize Filesystem Diagram`.
3. Enter a name for the diagram or keep the default name.
4. Click `Finish` to generate the diagram workspace.

Defining Hardware Components

Step 3: Creating a Deployment Node

1. Open the `Palette` view on the right-hand side of Eclipse.
2. Select the **Node** element and place it onto the canvas.
3. In the `Properties` view, set the following attributes:
   - **Mac Protocol**: ZIGBEE (standard communication protocol)
   - **Name**: Occupancy Sensor
   - **OS**: TinyOS (embedded operating system for wireless devices)

- **Routing Protocol**: GEAR (Geographical Energy Aware Routing)



Property	Value
Mac Protocol	ZIGBEE
Middleware	
Name	Occupancy Sensor
OS	TinyOS
Routing Protocol	GEAR
Transport Protocol	

Step 4: Adding a Microcontroller

1. From the `Palette`, select **Microcontroller**.
2. Place it inside the previously created Node.
3. This represents the microcontroller managing the sensor and data processing.

Step 5: Adding a Processor

1. Select **Processor** from the `Palette`.
2. Place it inside the **Microcontroller**.
3. Set the following attributes in the `Properties` view:
   - **CPI (Clocks Per Instruction)**: 1.0
   - **Frequency**: 120 MHz
   - **Name**: Atmel Atmega328

Step 6: Adding Volatile Memory (RAM)

1. Select **Volatile Memory** from the `Palette`.
2. Place it inside the **Microcontroller**.
3. Set the following properties:
   - **Name**: RAM
   - **Size**: 2 KB

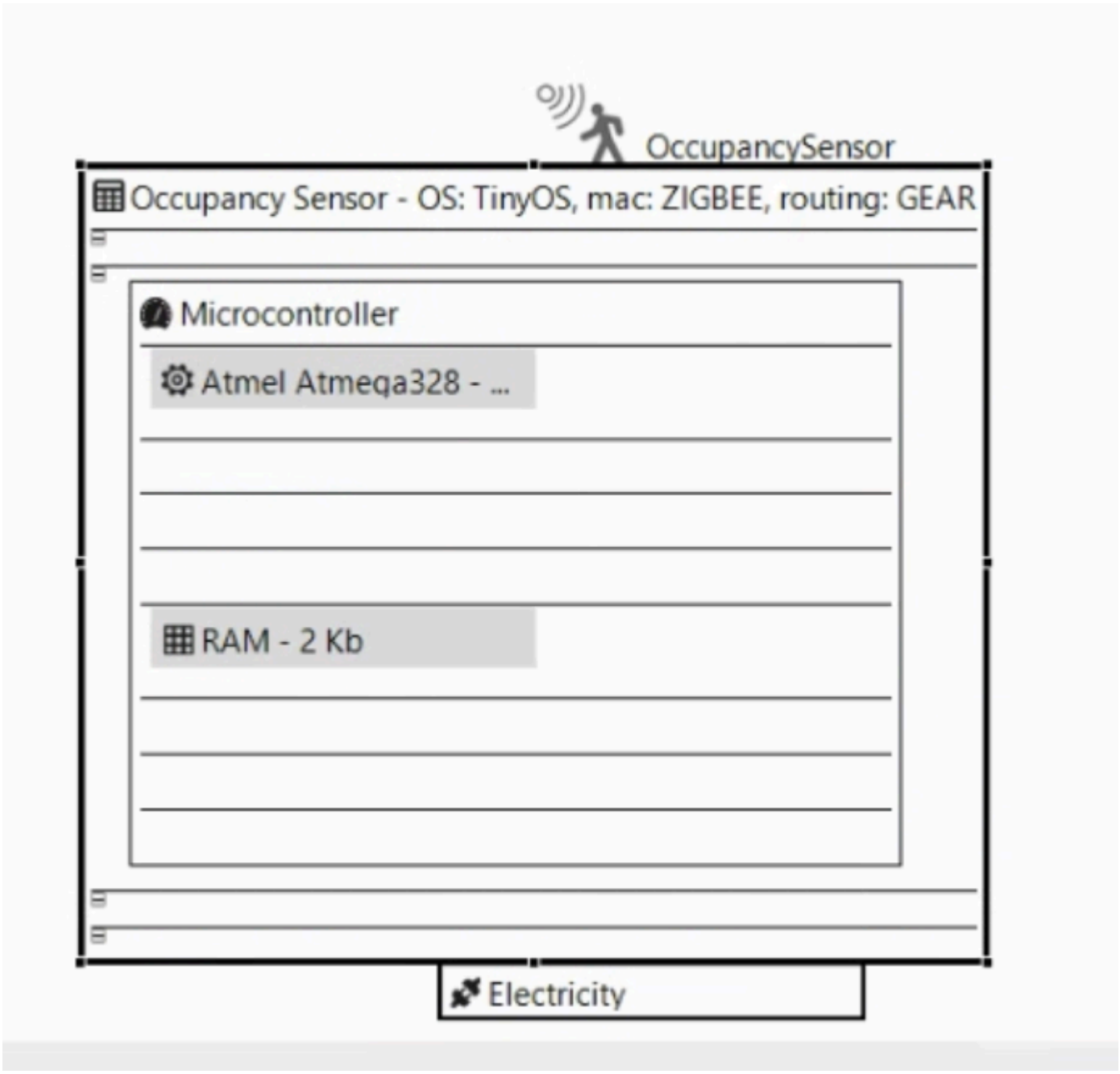
Step 7: Adding an Energy Source

1. Select **Continuous Energy Source** from the `Palette`.
2. Place it inside the **Node**.
3. Set the **Name** to `Electricity` (indicating a continuous power source).

Step 8: Adding a Sensor

1. Select **Occupancy Sensor** from the `Palette`.
2. Place it inside the **Node** to model the actual occupancy-sensing component.

Final Diagram:



Cup Carbon

Essentials

This covers a few of the essentials needed. For complete information, please refer to the **CupCarbon® User Guide Version U-One 5.1.**

CupCarbon Environment

To execute CupCarbon (jar file), use the command window and go to the directory where the jar file is located.

Then execute the following command:

```
java -jar CupCarbon.jar
```

In the case of the existence of a proxy, use the following command:

```
java -jar CupCarbon.jar proxy_host_name proxy_number_of_port
```

As shown in the Figure, the CupCarbon Graphical User Interface (GUI) is composed of the following five main

parts:

1. The map (in the center)
2. The menu bar (on the top)
3. The Toolbar (below the menu)
4. The parameter menu (on the left)
5. The state bar (at the bottom)
6. The console (in version 5, the console is separated from the main interface)

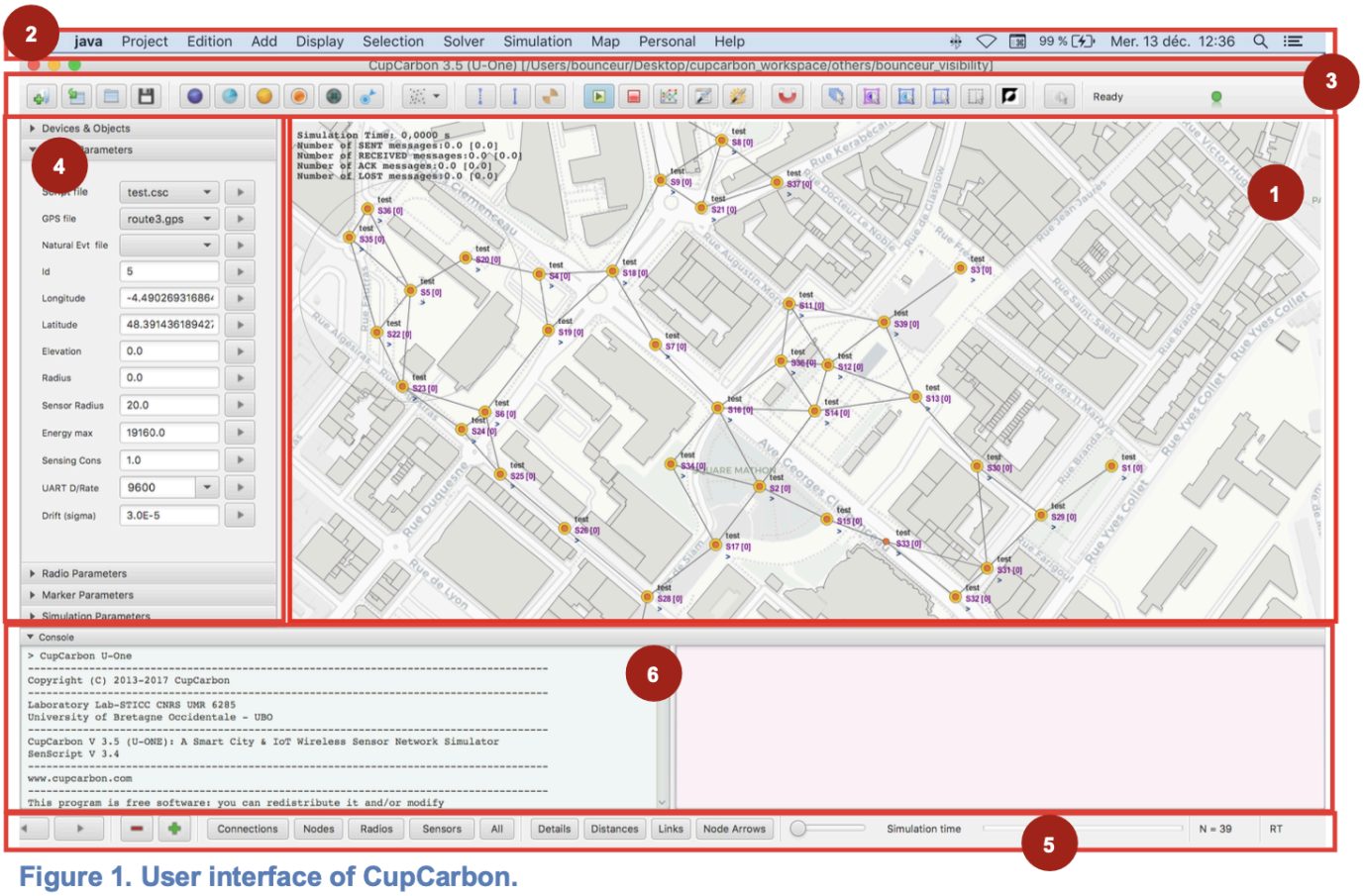
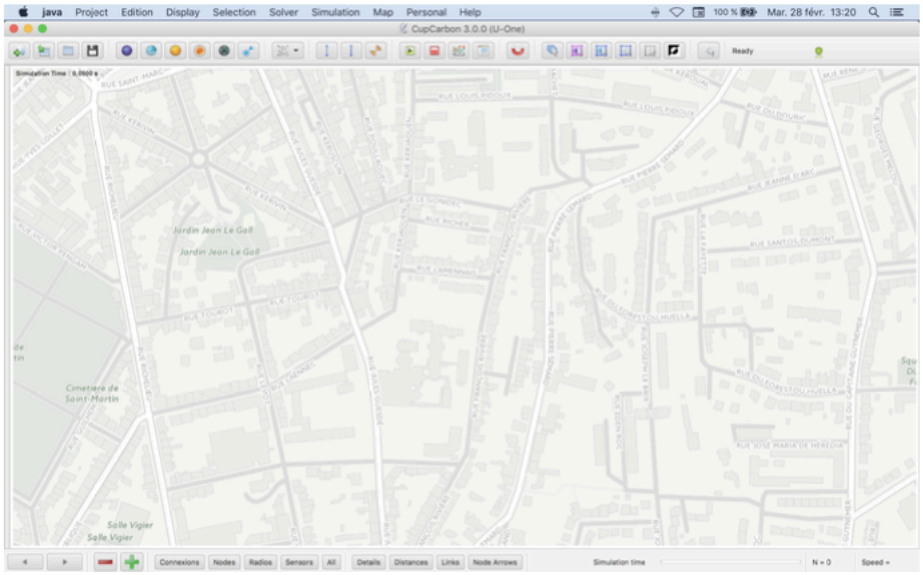


Figure 1. User interface of CupCarbon.

The Map

- The map is the main object of the simulator CupCarbon. It is the part where the network and the objects of the project can be designed. The map can be changed according to the preference of the user or the way the information must be presented.



The simulation time is displayed on the top left part of the map. During simulation, this time is displayed in red color, and an additional red rectangle is drawn around the map to detect the simulation process (cf. Figure 3 (a)). In this part, other information about the messages is also displayed, including the number of sent, received, ACK, and lost messages. This part can be hidden and displayed using the ALT+D keys.

(a) (b)

The Toolbar

The toolbar of CupCarbon is used to access the main actions of CupCarbon.

It is composed of 7 parts that are:

Project part

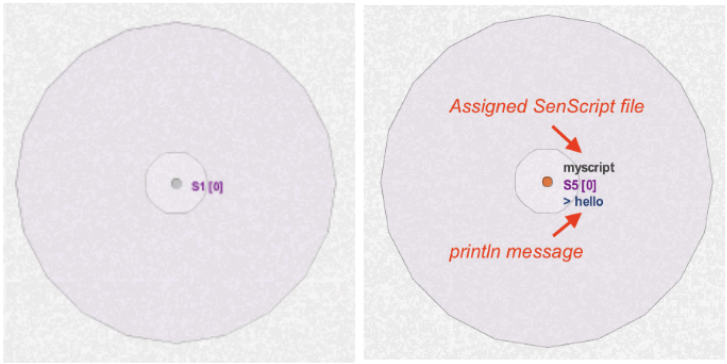
It allows creating a new project, opening the last project, opening a project, and saving a project.

Add object part

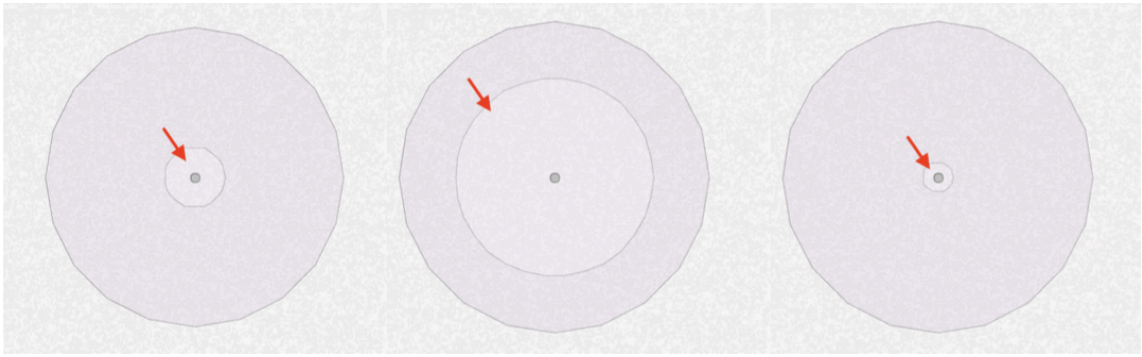
It allows adding objects (Sensor node, Directional Sensor node, Base station, Gas, Weather, Mobile, Marker, random sensor nodes) on the map.

Add Sensor Nodes: A sensor node is an object that can detect any digital event (motion event like mobiles), send and receive data. It can also be mobile. The visible parameters of a sensor node are: the radio range, the radio of the sensor unit, and the name. A sensor node has many parameters; it can contain many radio modules, a battery, and a sensing unit.

In the center of the sensor node, we find the name S followed by its ID. For example, if its ID is equal to 4, then its name will be S4. In the right part of the name, we find a number situated between brackets, which is equal to [0] by default. This number represents the MY address of this sensor node. If a SenScript is assigned to it, it will be displayed in a gray color above its name. The *print* messages will be displayed in blue below their names.



A sensor node contains a sensing unit represented by a transparent white circle. The area's radius can be changed using the buttons ‘(’ for increasing the radius and '(‘ for decreasing the radius.



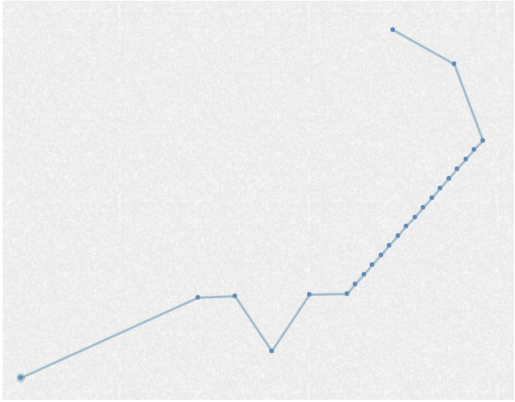
Add Mobiles: used to simulate mobiles. Markers are also used to create routes followed by mobiles. Each mobile must have its route. They are also used to generate digital events.

Route not assigned (gray center)

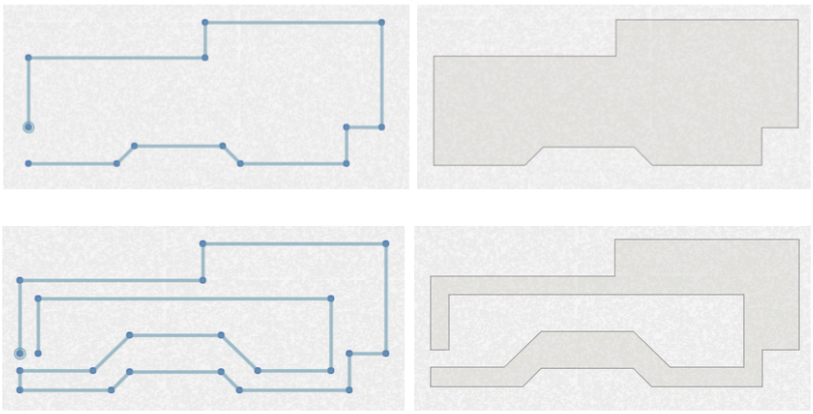
Route assigned (orange center)

A mobile depends on its route (trajectory). This last one is created using markers, as it is explained in the next section, bellow, one needs just to select the route which will be followed by the mobile in the list of created routes from the field GPS file in the Device Parameters view

Add Markers: used mainly to generate routes for mobiles (or mobile sensors). They are also used to generate sensor nodes, create new buildings, and indicate the area of generating buildings or random sensor nodes.

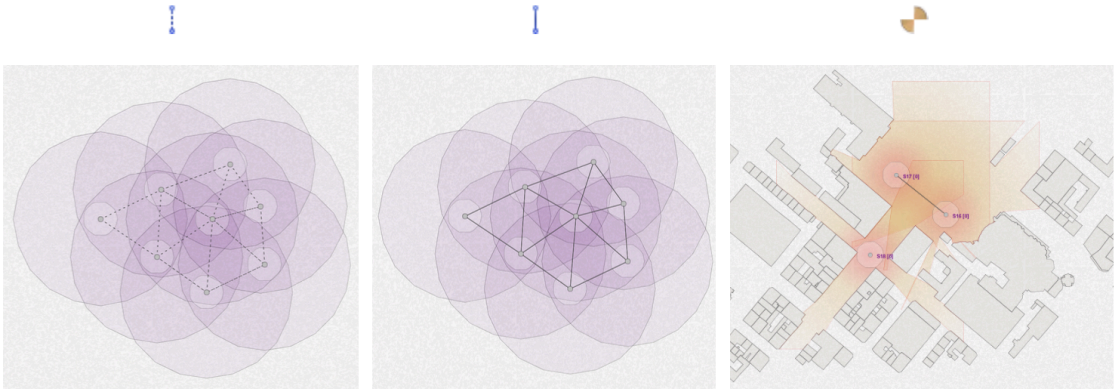


It is possible to generate routes either by drawing the route manually using the markers or by creating just two markers and clicking on the button Route From Markers in the Marker Parameters view. Then, each created route can be saved and added to the list of routes of the project. This procedure is explained above in the Marker Parameters view section. A list of buildings can be added in the area delimited by two markers. It is possible to draw a building by drawing the form using markers and then by typing on the key ':'.



Connections part

It allows for drawing normal or radio propagation-based connections between sensor nodes. It also calculates the visibility of the radio of the sensor node by considering the buildings of a city.



Simulation part

It allows running the simulation, stopping the simulation, drawing the energy consumption function, open the SenScript window and to open the Natural Event generator.

Run Simulation: to start the simulation

Stop Simulation: to stop the simulation

Energy Consumption: to display the graph of the energy consumption for the selected sensor nodes once the simulation is finished. You must first check the box *Results* before running the simulation. Two kinds of graphs are possible. The first one shows the state of the battery for the simulation time, and the other one shows the consumption of a sensor for the simulation time.

Magnetism part 

It allows to add objects in an (invisible) grid. It is recommanded to use the map (Mean gray cell background).

Selection part

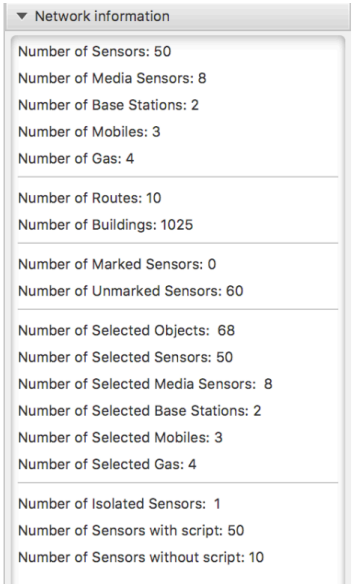
It allows to select all, select all sensor nodes, select all markers, select sensor nodes/markers, deselect all, and to invert selection

Credit: Content taken from CupCarbon® User Guide Version U-One 5.1

The parameter panel

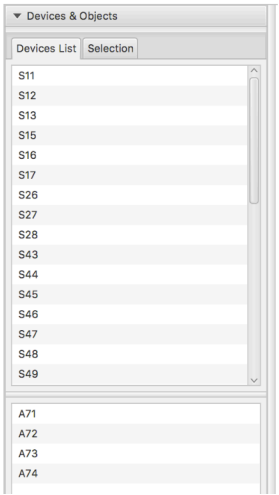
Network information panel

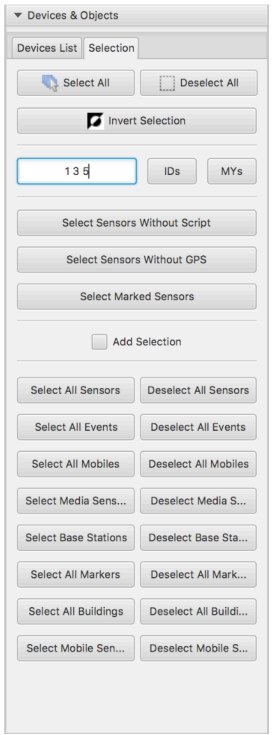
This panel shows some information about the network, like the number of sensors, the number of marked sensors, the number of isolated sensors, etc.



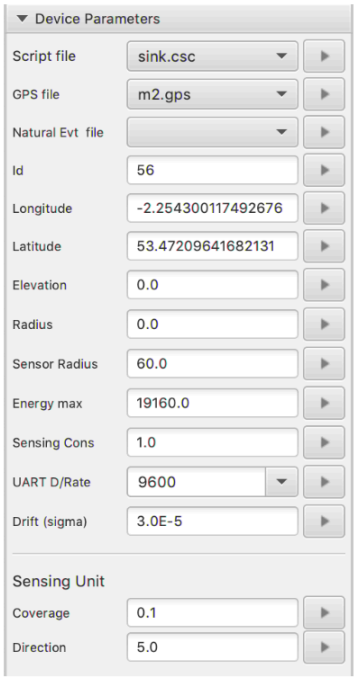
Devices & Objects [Device List] [Selection] panel

This panel has two tabs. The first one is Device List, which allows you to select an object on the map by its name. The second one is the Selection tab that allows you to select/deselect objects by their type. It is also possible to select objects by their MY addresses or their IDs. These addresses/IDs can be entered as a list of numbers in the corresponding text field.





Device Parameters panel



This panel allows you to modify the parameters of the selected objects, like:

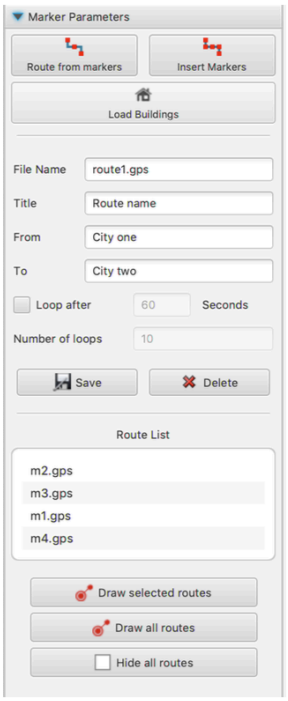
1. Script file: to assign the SenScript file
2. GPS File: to assign the route file
3. Natural Event File: to assign the natural event file generated from the Natural Event generator
4. Id: to assign an ID
5. Longitude: to assign a longitude
6. Latitude: to assign a latitude
7. Elevation: to assign an elevation
8. Radius: to assign a radius for the sensor node (this is not the radio radius)
9. Sensor Radius: to assign a radius for the sensing unit
10. Energy max: the initial energy of the battery
11. Sensing consumption: the sensing consumption in units (it is not considered in this version of CupCarbon)
12. UART Datarate: the UART datarate, which represents the necessary time to send data (bytes) to the buffer of the radio module.
13. Drift (sigma): the clock drift.
14. The coverage of a sensing unit (case of directional sensor node)
15. The direction (rotation) of a sensing unit (case of directional sensor node)

Any modification is considered only if it is followed by a click on the apply button with the right gray arrow situated in the right part of the corresponding field.

Radio Parameters panel

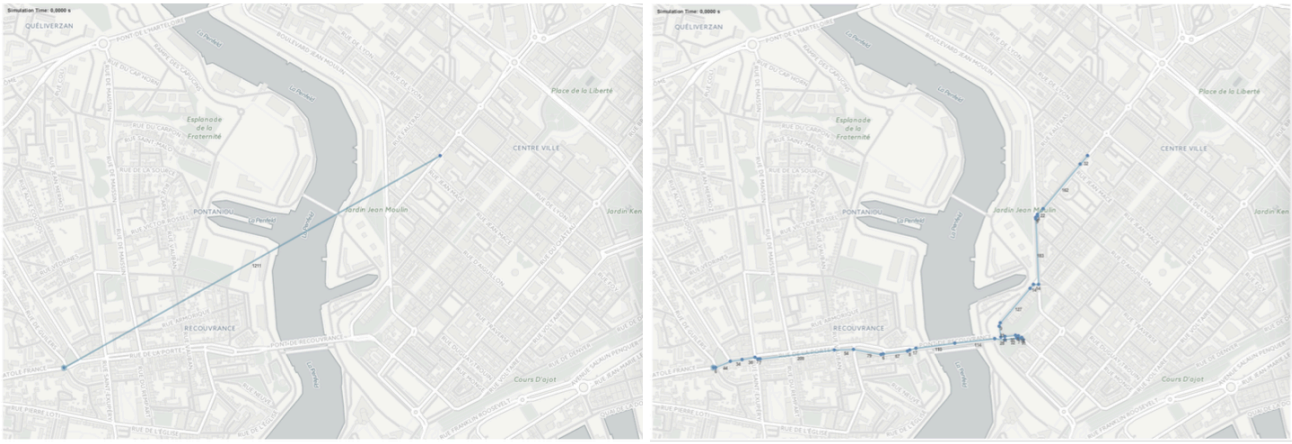
This panel allows for modifying the parameters of the radio module of the selected sensor nodes. From this, we mainly use Radius: the radius range of the selected radio module. In the propagation mode (when clicking on the icon of the state bar) is activated than this radius is calculated automatically depending on the signal propagation and the environment.

Marker Parameters panel



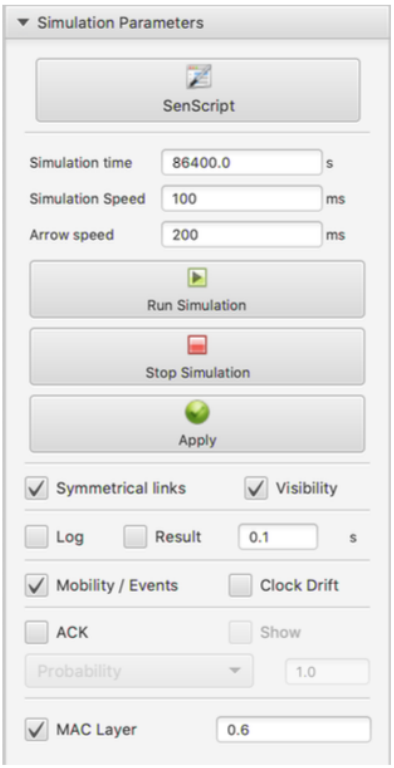
This panel allows for working with the markers as follows:

1. Route from markers: This button allows you to generate a route situated between two points of the map that are determined by two markers. If there are more than 2 markers, only the last two will be considered. This allows for drawing new routes as a continuation of the existing one.



2. Insert Markers: allows to insertion of markers after the selected ones (cf. Section Markers). The same result can be obtained by pressing the key ‘u’.

Simulation Parameters and SenScript Panel



This panel is used for the simulation. It contains the following buttons and options:

SenScript: to open the SenScript window

Simulation time: the duration of the simulation

Simulation Speed: is the speed of the simulation. The objective of this button is to be able to follow and to visualize the simulation at human speed. It is useful for debugging.

Arrow speed: the same as the Speed function, where the delay in this option is related to the sending/receiving message. It allows for to visualization of the messages.

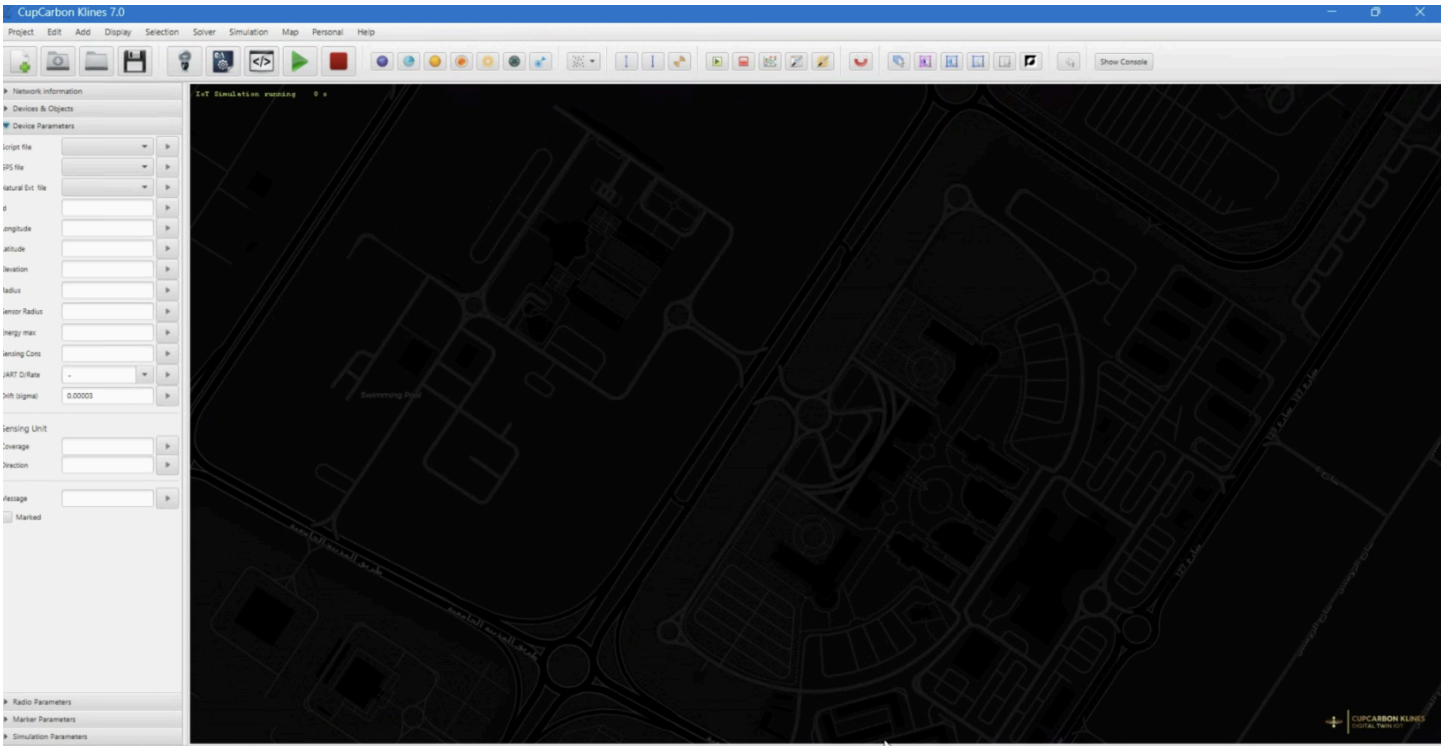
Run simulation: start the simulation (can also be done by pressing on Entree)

Stop Simulation: stop the simulation

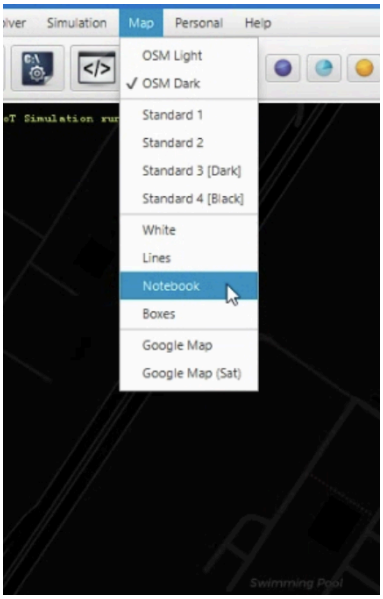
Cup Carbon

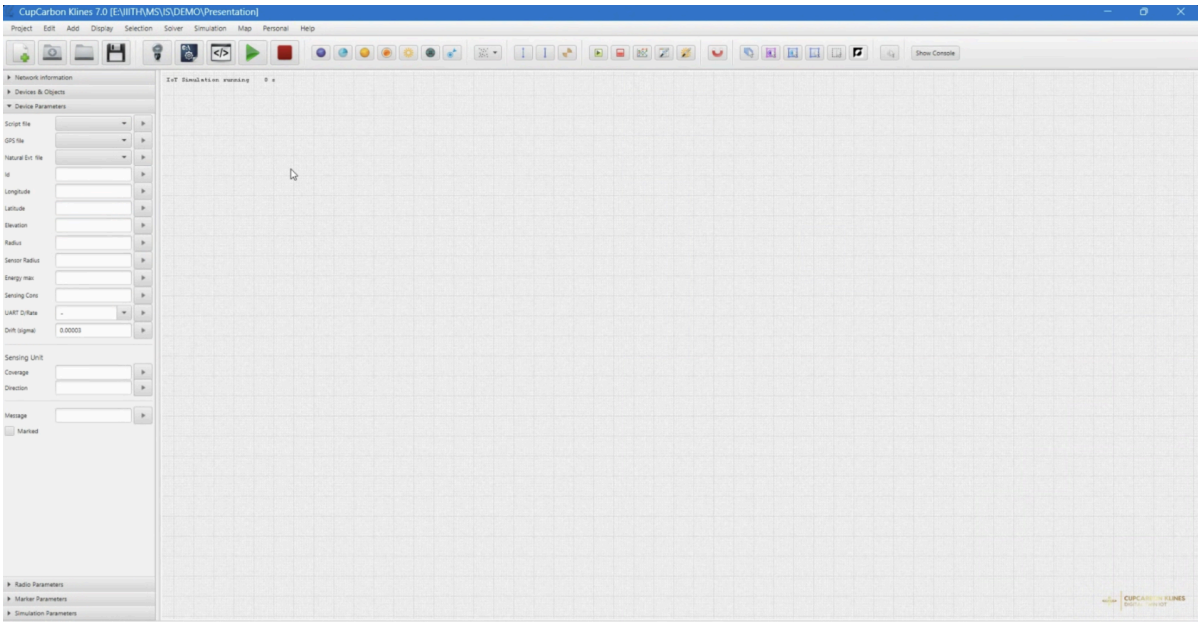
Simulate Motion Based Lighting System in Cup Carbon

Open the CupCarbon and create the project:



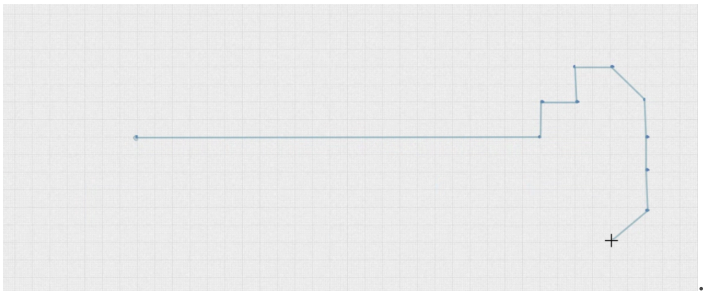
Now, go to the map and select the Notebook view:

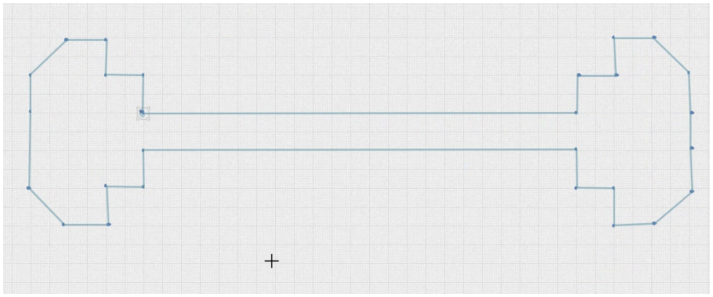




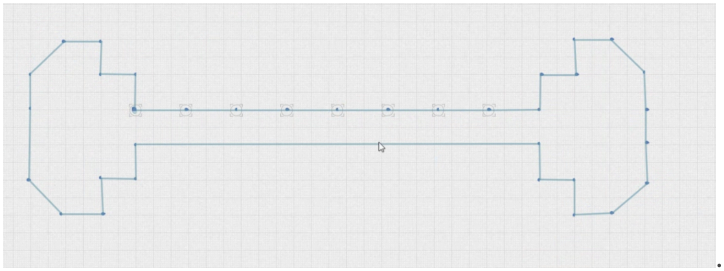
Feel free to explore and select the other views as well.

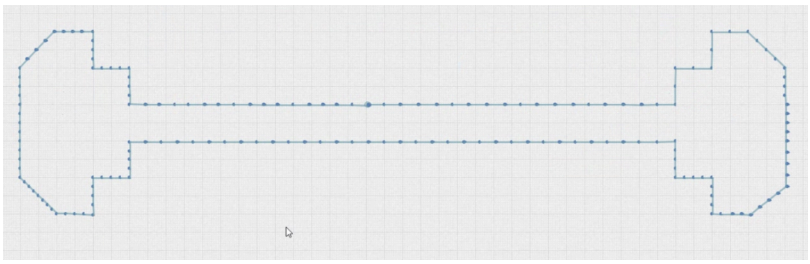
Now, let’s create the base path using markers. From the toolbar, select the icon with the sky-blue circle and an arrow, or you can also use the shortcut key number 8. Now you can see the **+** icon start clicking and draw the path, once you are done right click the mouse.



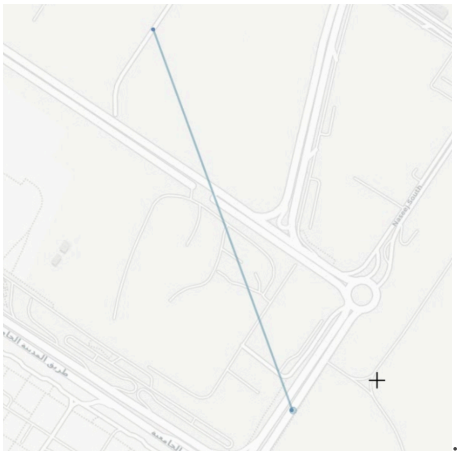


Once you get a base shape, it’s time to add more markers. These act as steps in the simulation, so the mobile node travels one dot per the given arrow speed. To start, click on the marker and keep pressing '**u**'.

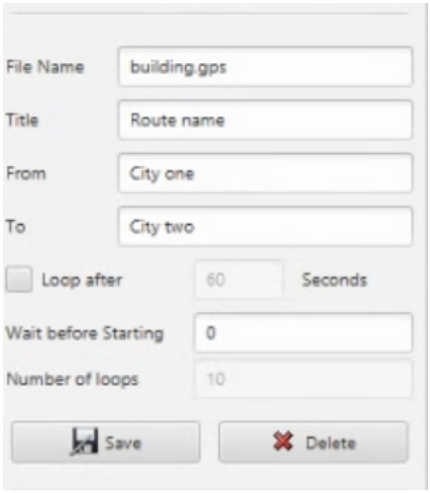




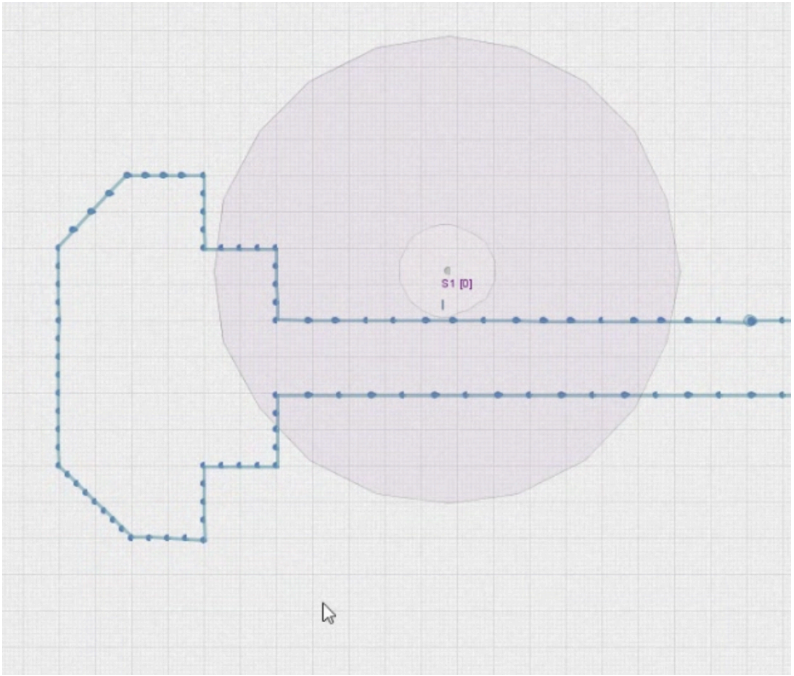
Add as many markers as you need, you can draw any shape, or you can choose map view and draw to points and click **Route From Markers** from the **Marker Parameters** section in the slide bar.



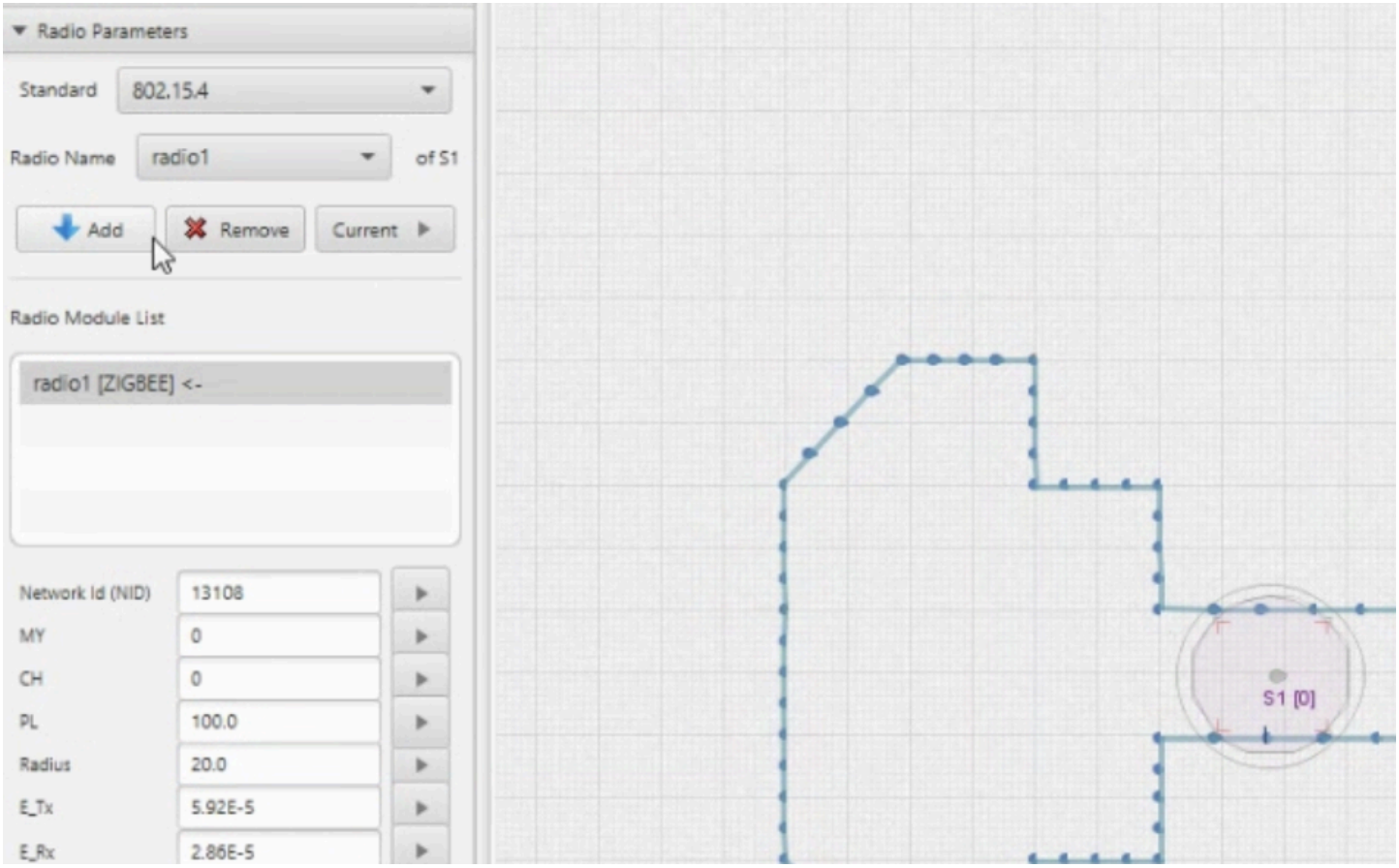
Save the markers drawn as the route from the side menu.



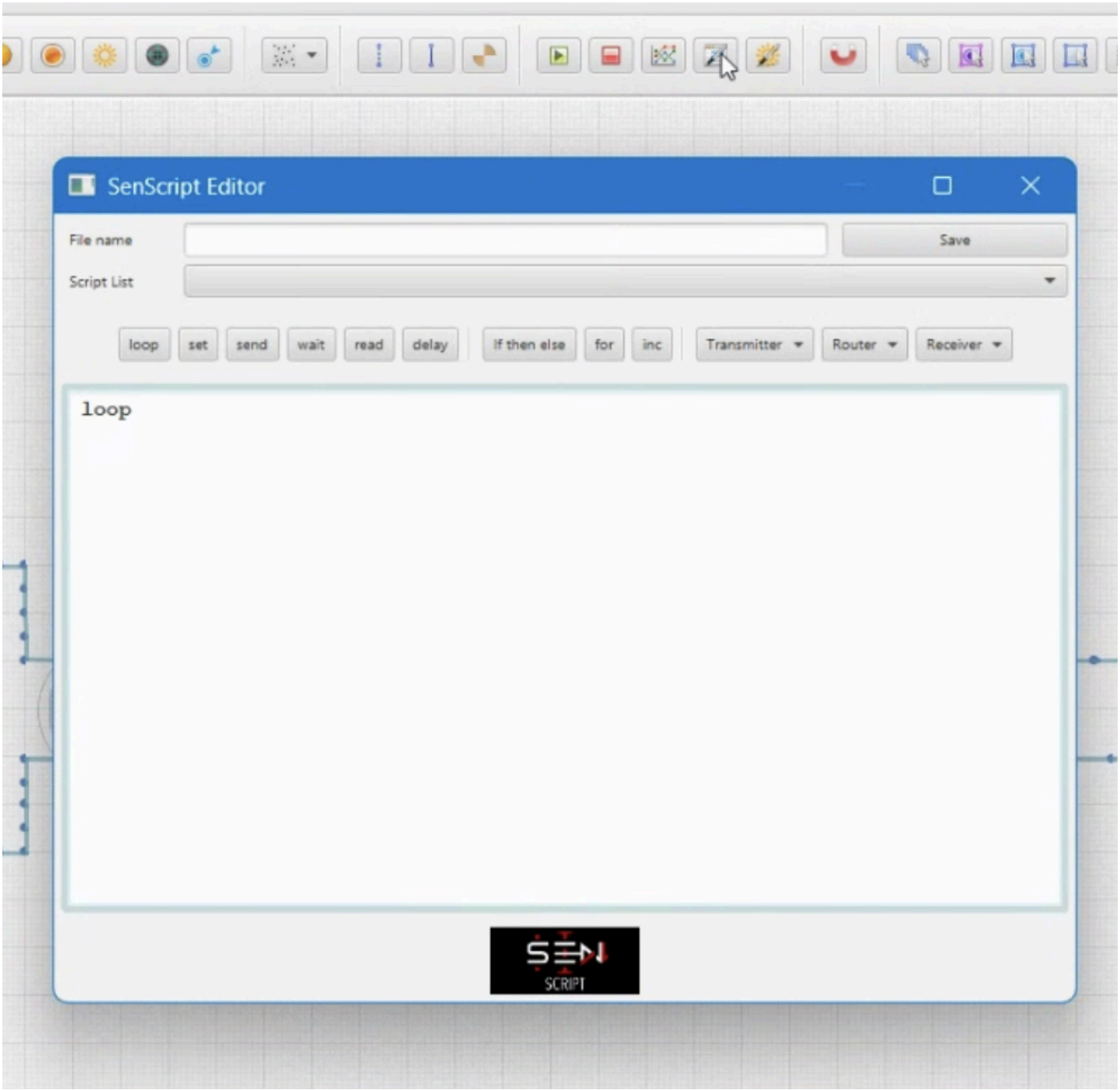
Now, it's time to add some sensor nodes. We will be using a total of three types: sensor node, IoT node, and mobile node. We will create two types within the sensor node: one for the controller and one for the sensor. to add the sensor node click on the purple circle from the toolbar or use the shortcut key '**1**'.



Let's now go deeper and make this a sensor node that is supposed to detect the activity. As this sensor will be placed on the path, we created the inner circle should be on the points of the so we will reduce the radius to **20** and then also will keep the radio radius **20**. Because keeping this bigger causes the other nearby sensors to be connected. We want the lights to be turned on only when the user is at that point. The same process will be followed for the controller/router. The difference between the sensor and the controller/router is sensor takes the readings, and the controller will forward the readings to the nearby nodes.



To make it alive, we need logic. So, click on the **SenScript Window** and add the code for both the sensor and the controller.



First, we will add the sensor code:

```
atget id id
loop
dreadsensor x
data message id x
send message 0 2
delay 1000
```

- **atget id id** → id = id, my or ch of the sensor node
- **loop** → Starting the loop section
- **dreadsensor x** → x=1 if the sensor detects an event (mobile), x=0, otherwise
- **data message id x** → This line prepares a **data message** to be sent. It packages the data gathered from the sensor (`x` is the data being read), and associates it with the **ID** of the node (`id`). This message will be sent to another node or system.

- **send message 0 2** → This sends the message from the sensor node to all nodes within its range. The **ID** of the node is set to **2** (this should be configured in the panel for the controller node). By doing this, we can **broadcast** the message to all nodes within range. If we hardcode the ID for each node, we would have to manually change it for every single node, which is inefficient. Broadcasting the message allows for dynamic communication across the network.
- **delay 1000** → This introduces a **delay** of 1000 milliseconds (1 second).

and save it with the extension **csc**.

For the controller/router node:

```
atget id id
loop
wait
read message
rdata message rid x
data message2 id x
send message2 * rid
```

- **atget id id** → id = id, my or ch of the sensor node
- **loop** → Starting the loop section
- **wait** → wait for the message
- **read message** → reads the message sent
- **rdata message rid x**→ This command extracts data from the **received message** (stored in `message`).
  The message is assumed to be a string formatted with fields separated by `#`.
  - It splits the message into parts.
  - The first part is assigned to `rid` (the sender's ID).
  - The second part is assigned to `x` (the data payload).
- **data message2 id x**→This line prepares a **data message** to be sent. It packages the data gathered from the sensor
- **send message2 * rid**→ Sends `message2` to **all nodes in range of** `rid` (the original sender).
  The `*` indicates broadcasting to all nodes within range of `rid`.

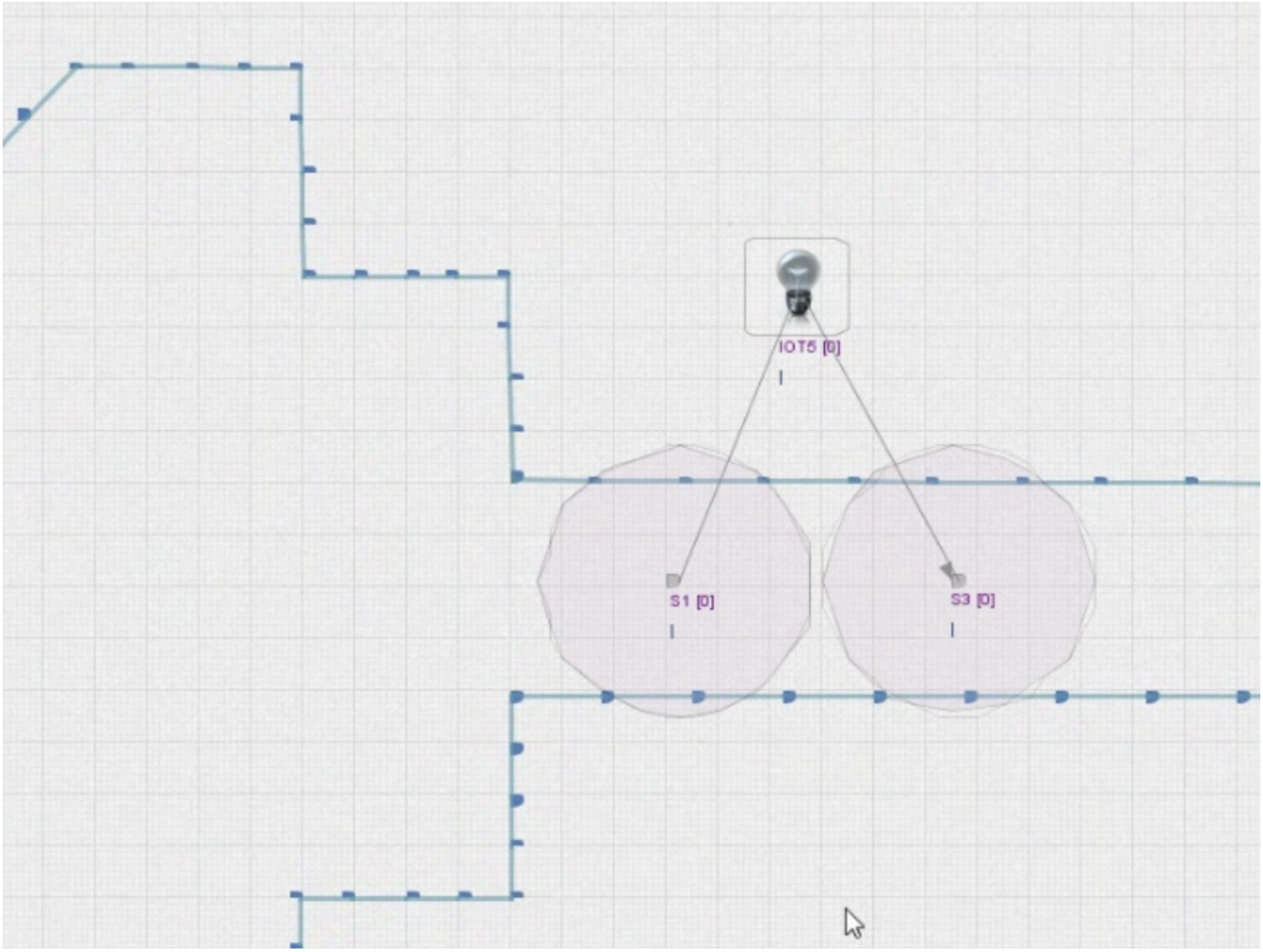
Finally, we will write the logic for the reciving node(IoT node):

```
loop
wait
read message
rdata message rid x

if(x==1)
        print "Detected"
        mark 1
else
        print "Not Detected"
        mark 0
end
```

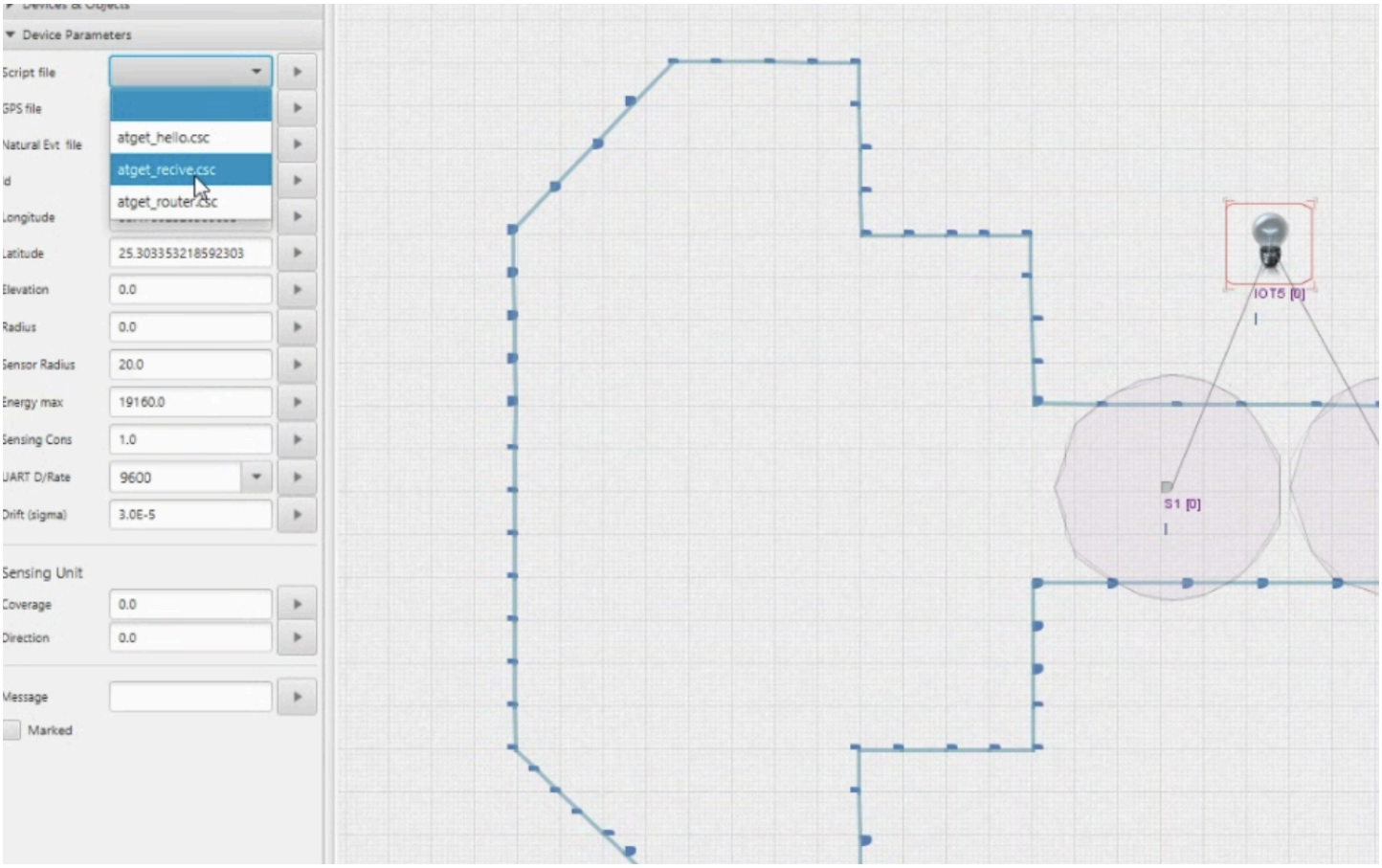
- **loop** → Starting the loop section
- **wait** → wait for the message
- **read message** → reads the message sent
- **rdata message rid x**→ This command extracts data from the **received message** (stored in `message`).
  The message is assumed to be a string formatted with fields separated by `#`.
  - It splits the message into parts.
  - The first part is assigned to `rid` (the sender's ID).
  - The second part is assigned to `x` (the data payload).
- **if condition** → This condition checks if the value received is 1 or not, if 1 it prints Detected and turn on the node else prints Not Detected and keeps the node off or turn the node off.

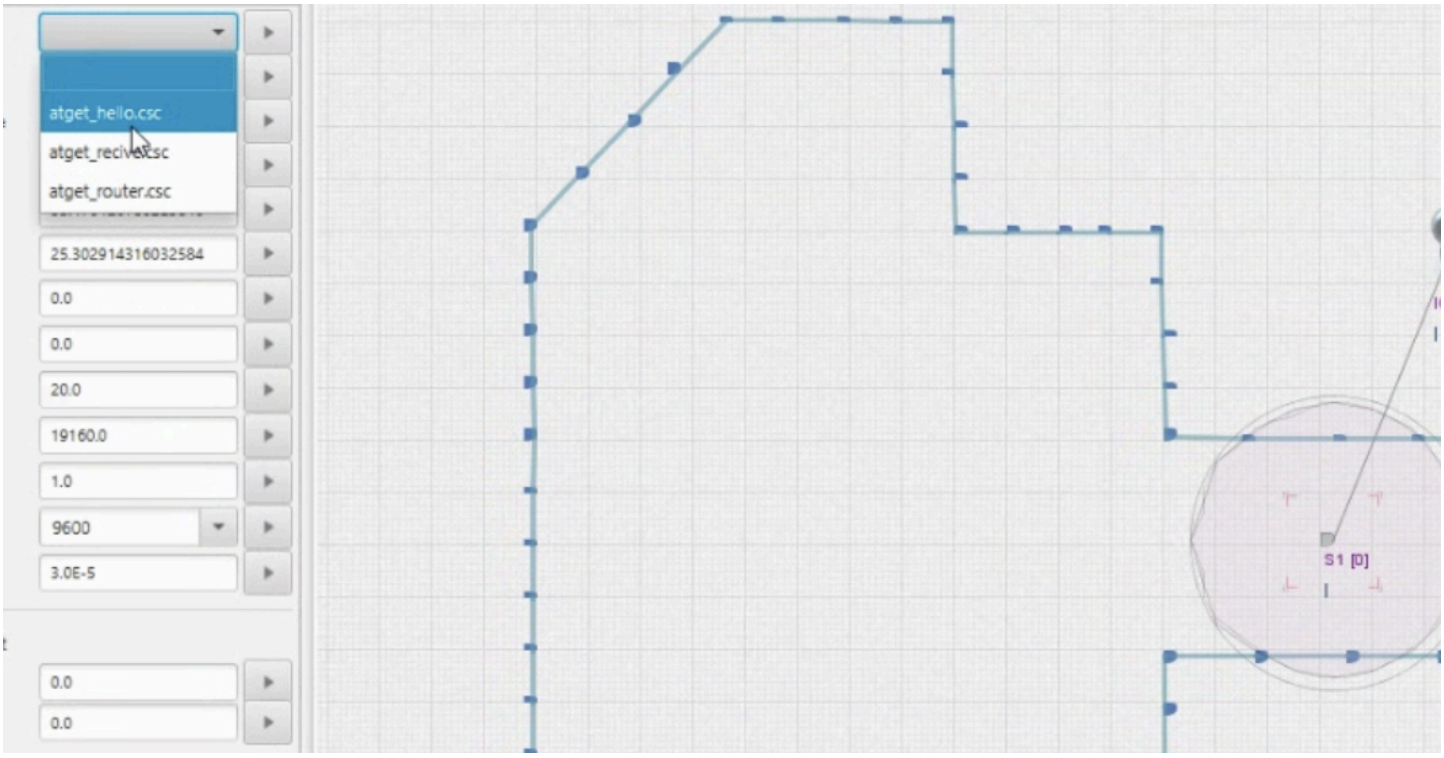
After adding the one more node and a IoT node(This can be added using the bulb icon in the toolbar), the screen looks like this:

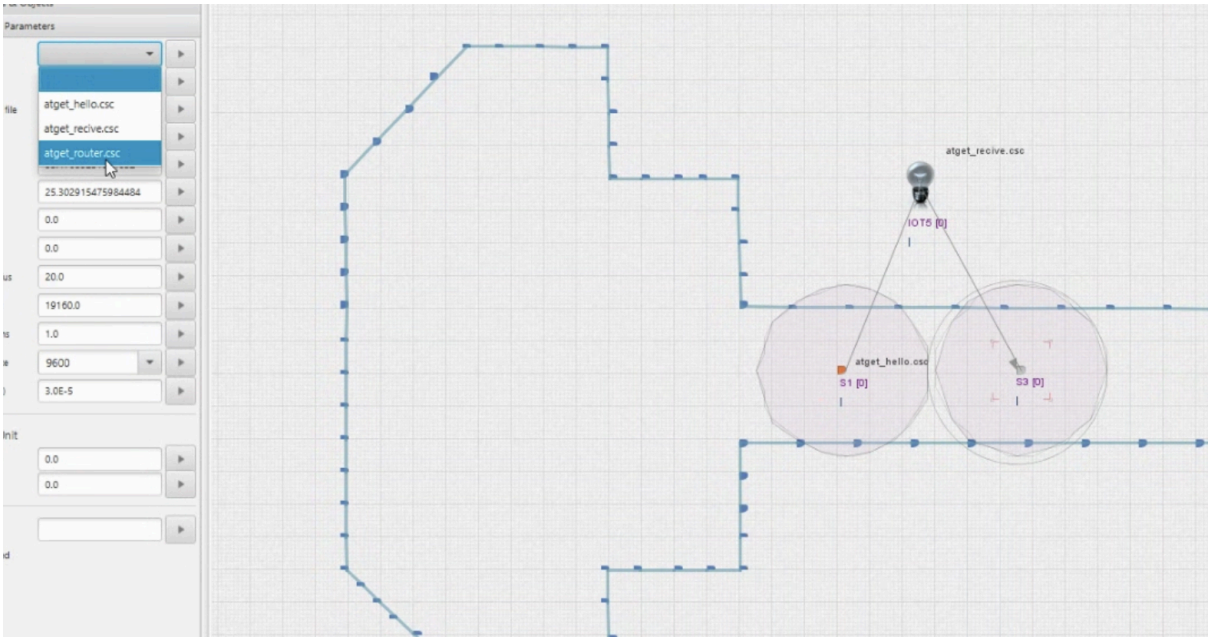


Arrows will be automatically added as the nodes are with in the range of the radio of the IoT node by default. now we have to assign the scripts to the nodes to make bring our model to the life.

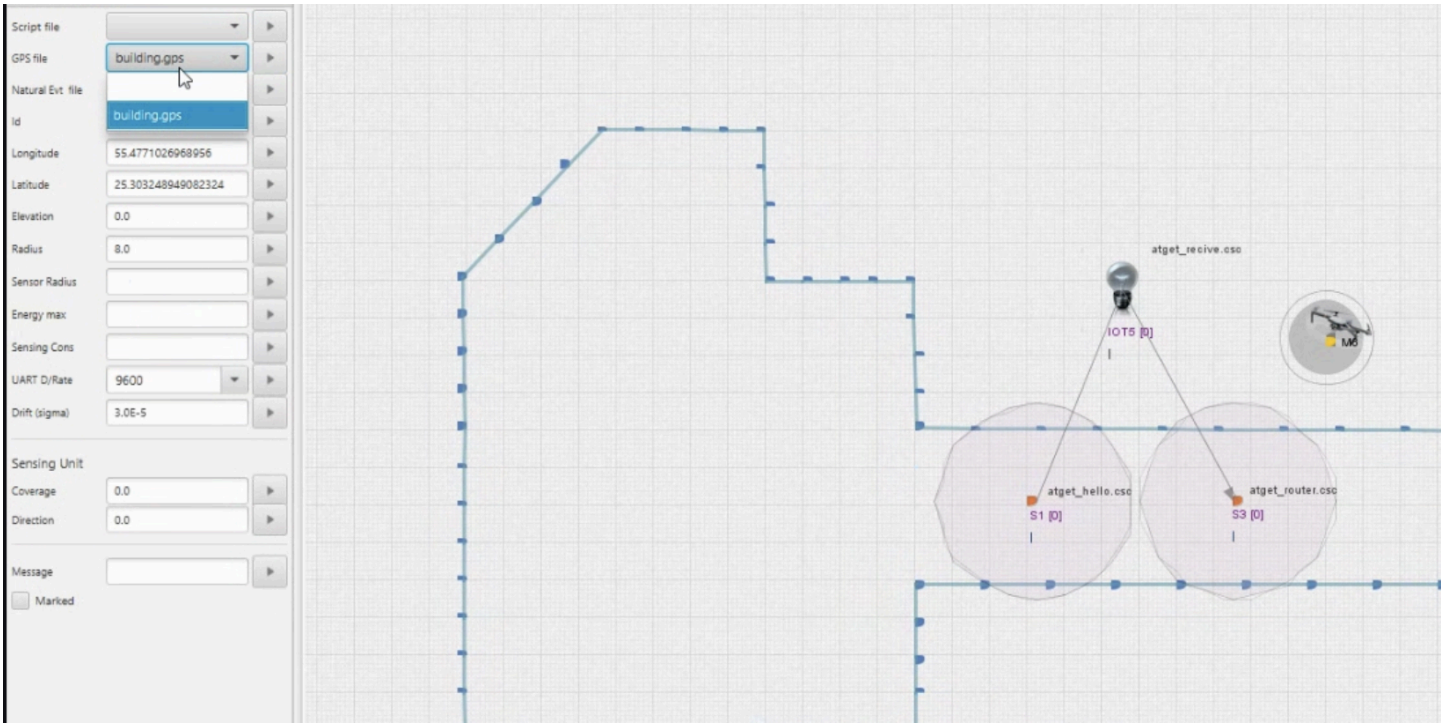
Now i will assign the sensor script to the node id S1 and the router script to the S3(this can be any node number) and to the IOT node my final receving script, to do this i will click on the node and then go to the **Device Parameters** section and select the respective script file.


Device Parameters
Script file
GPS file
Natural Evt file
Id
Longitude
Latitude
25.303353218592303
Elevation
0.0
Radius
0.0
Sensor Radius
20.0
Energy max
19160.0
Sensing Cons
1.0
UART D/Rate
9600
Drift (sigma)
3.0E-5
Sensing Unit
Coverage
0.0
Direction
0.0
Message
Marked
atget_hello.csc
atget_recive.csc
atget_router.csc
IOT5 [0]
S1 [0]

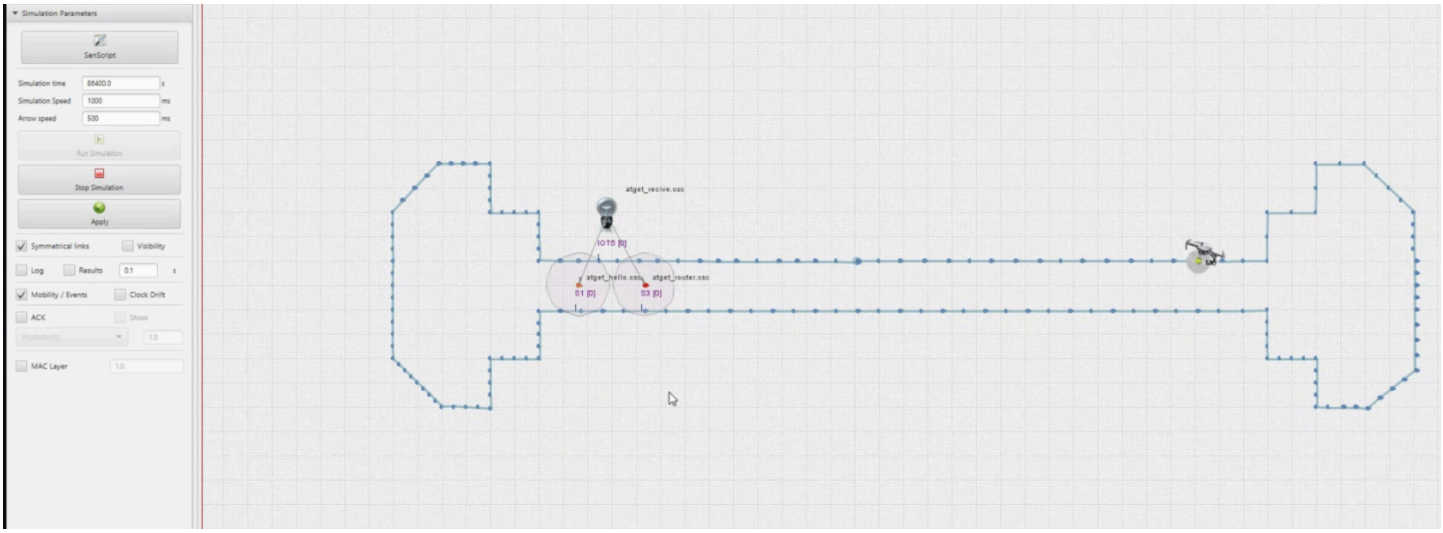

atget_hello.csc
atget_recive.csc
atget_router.csc
25.302914316032584
0.0
0.0
20.0
19160.0
1.0
9600
3.0E-5
0.0
0.0
S1 [0]

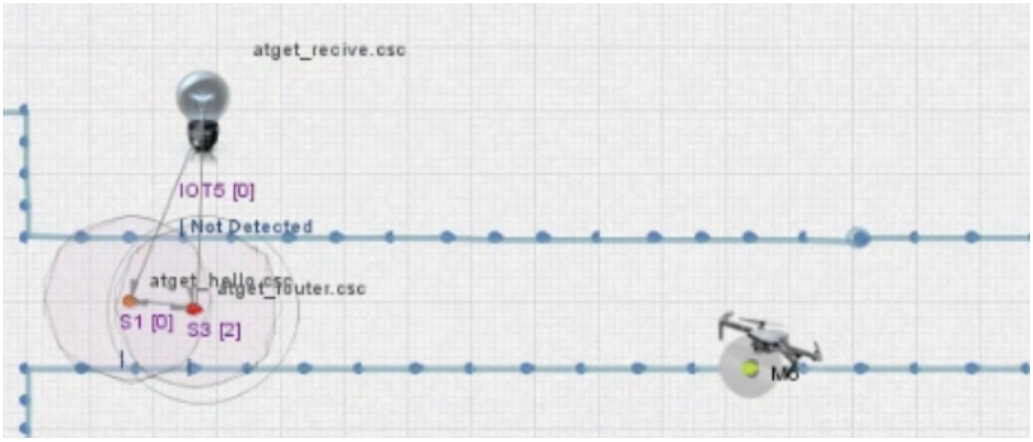


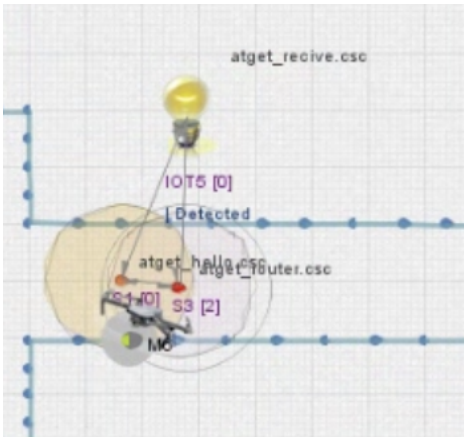
Make sure you also **set the MY value in the Radio Parameter to 2 for router node only.** After Assigning screen looks like this, confirm that scripts are loaded. The name of the loaded script will be shown in the side. we now add the mobile node and assign the gps file only to it(the orange dot indicates the gps file is added to it).



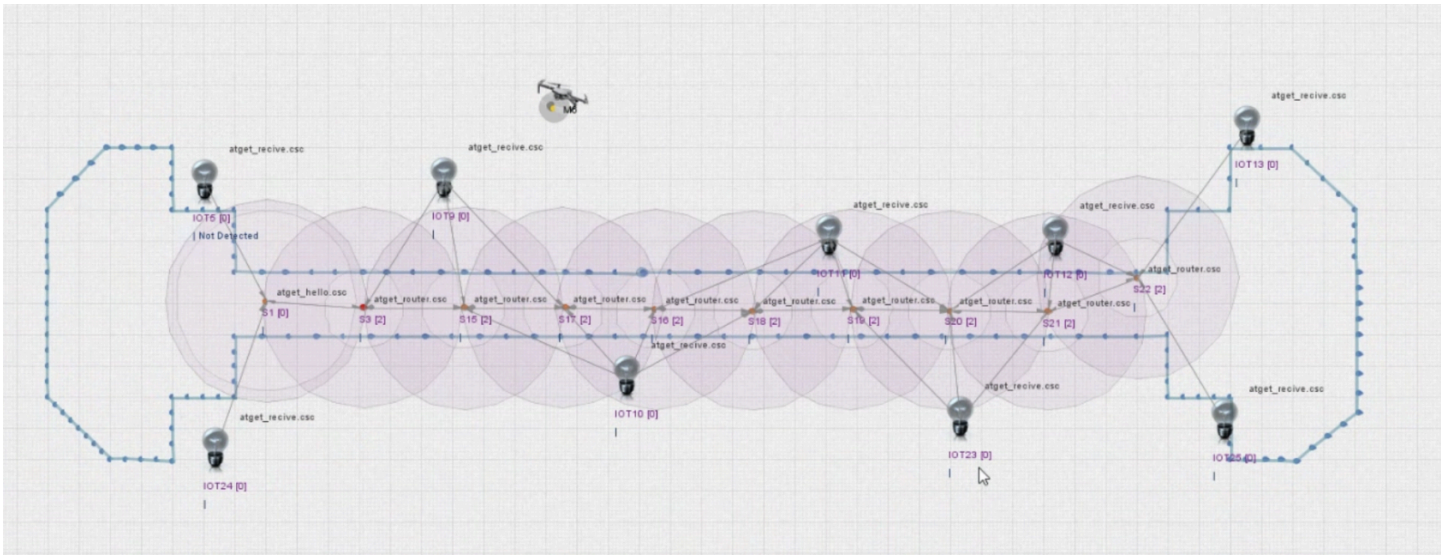
Now its time to run the simulation. Click on the main simulation button, once started then to make the mobile node running go to the **Simulation Parameters** and run the simulation. **Also, ensure that the router is connected to the sensor, if not move it close.**



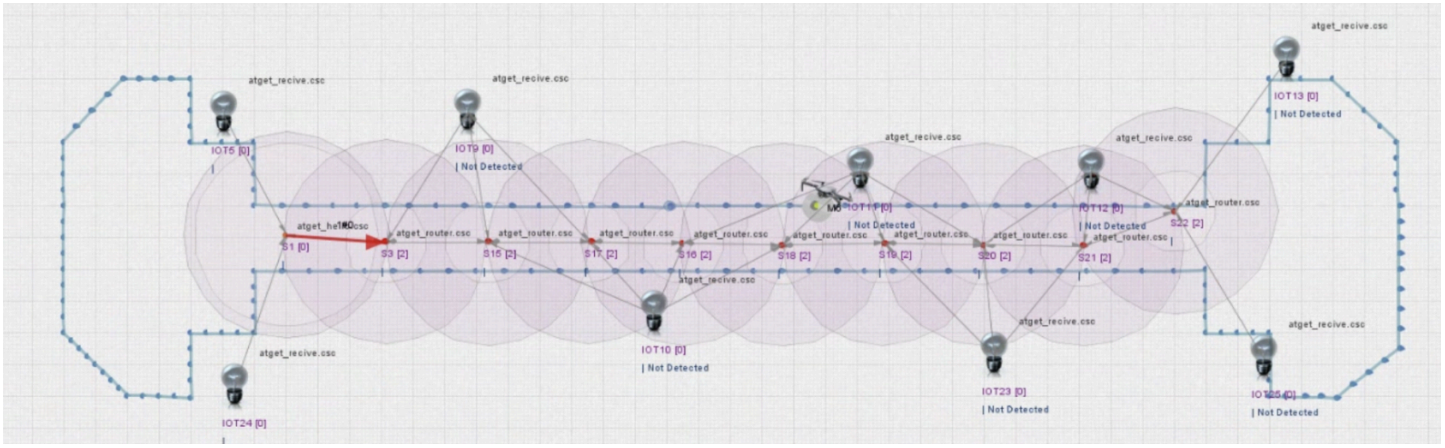


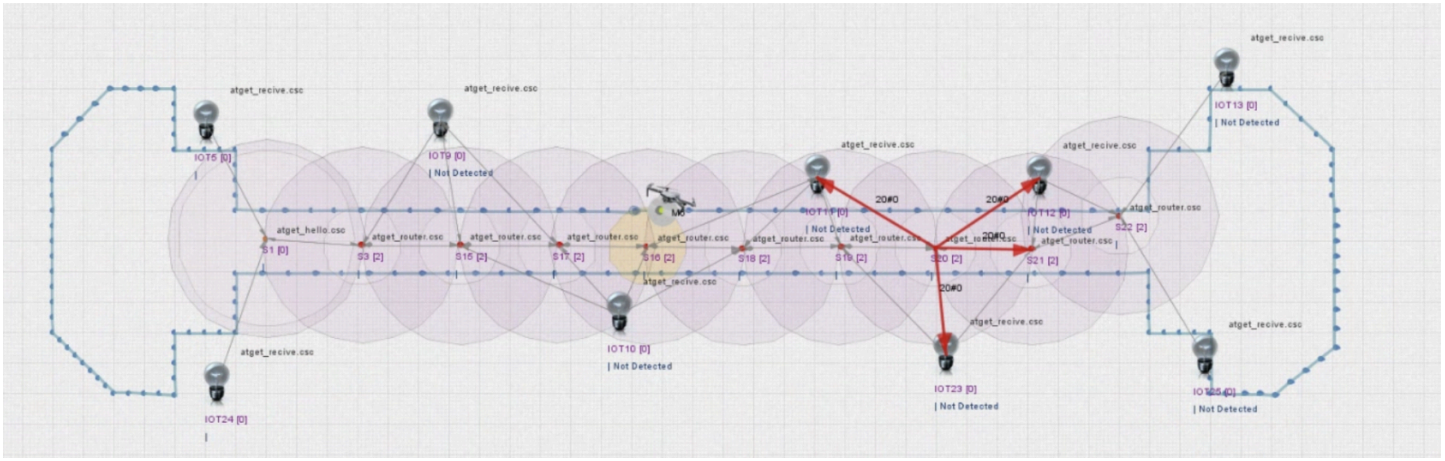


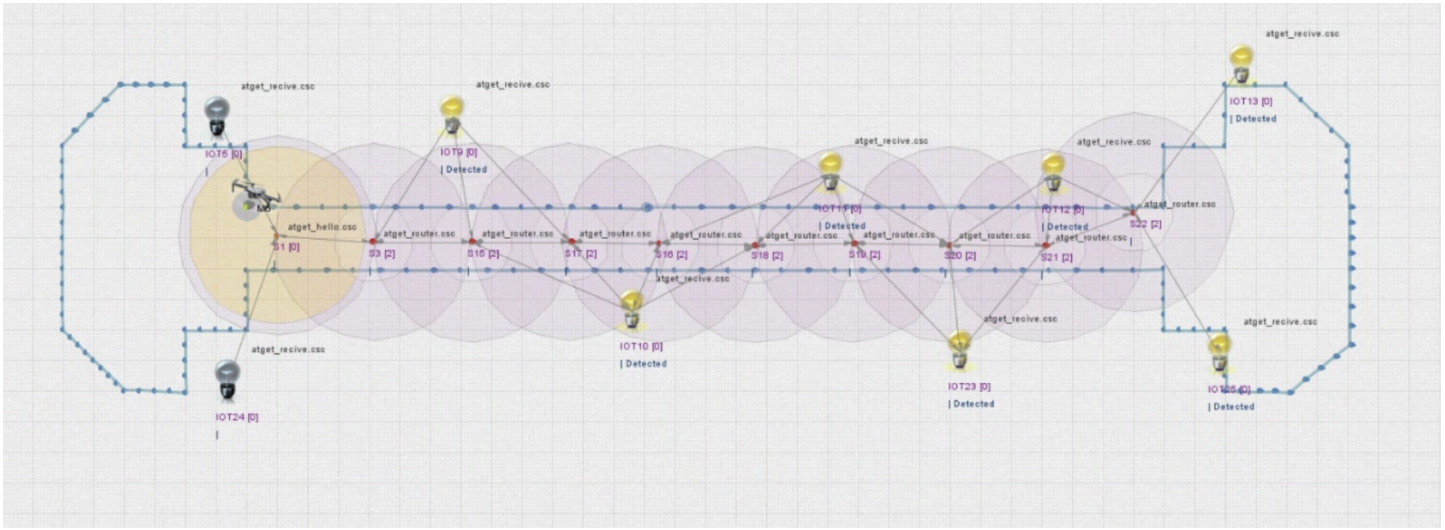
Now lets add more routers, lights and check our dynamic code logic.



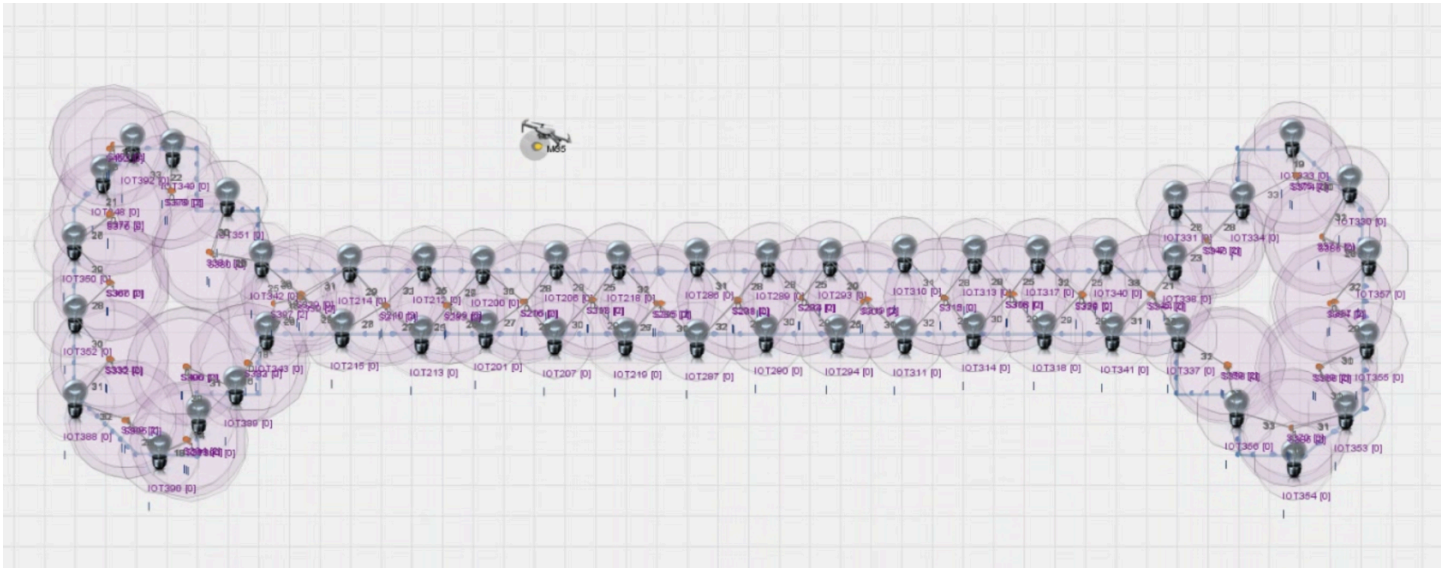
Now i multplied the router nodes keeping the single sensor node and all the bulbs are conncected to the router nodes, so whenever there is a moment at the sensor node all the bulbs will get turned on. From the sensor the information will be sent to the bulbs through the routers.



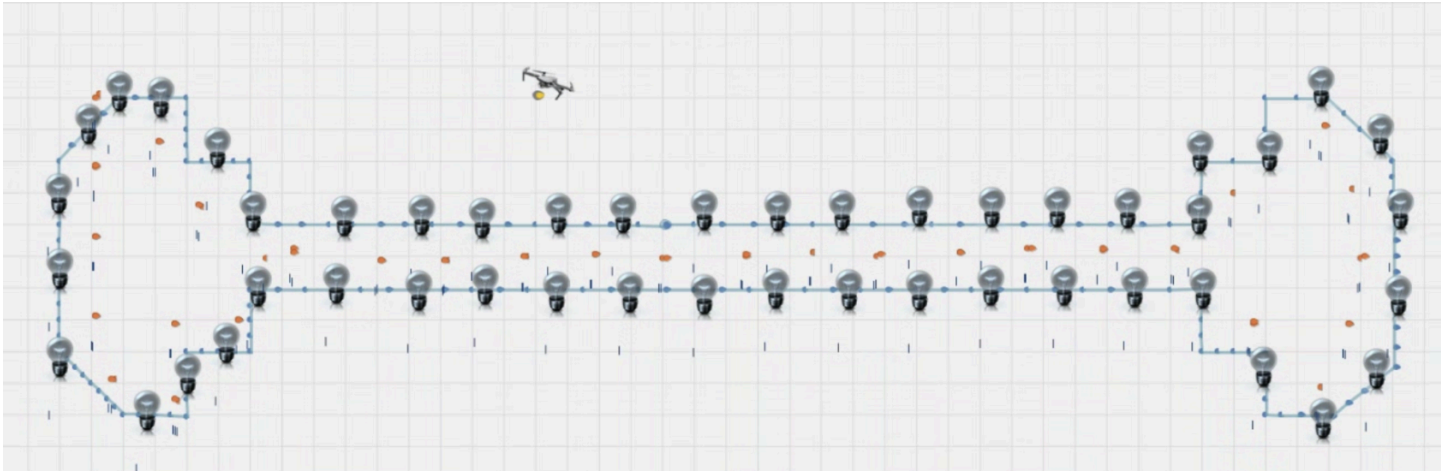




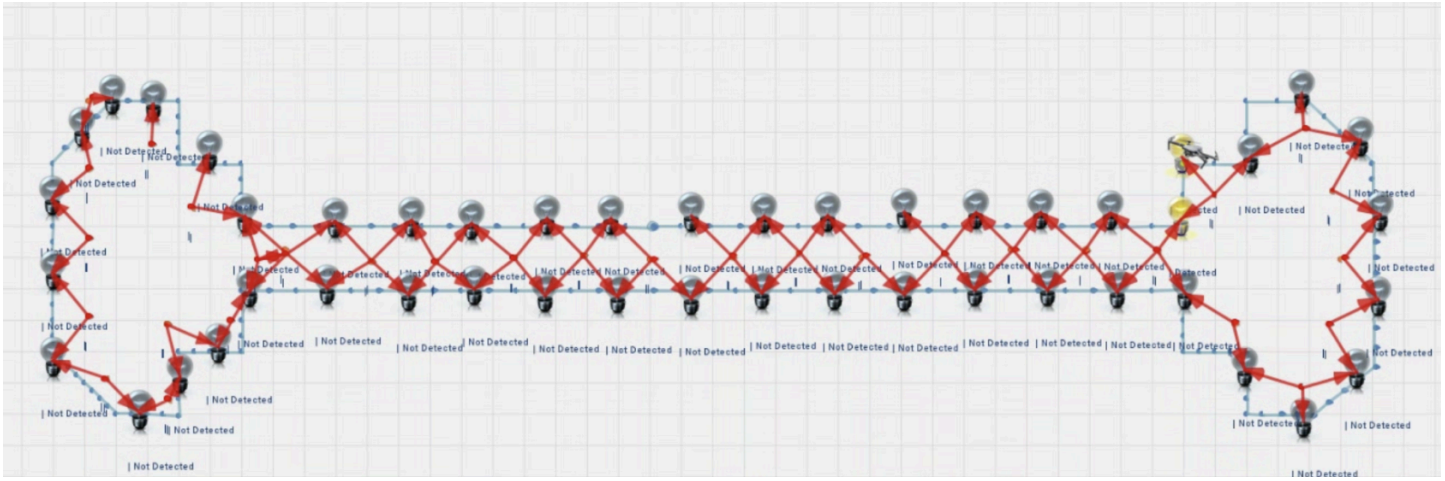
To make this more intresting, we can multiple the sensor nodes and the router nodes where the one sensor node will send the information to only one router and the router will send the information to the nearby bulbs, This simulates the motion based lighting. Here is the complete example:



Hiding all the details, it will look like this



During Simulation:



All the redlines indicate the data flow between the nodes. Now the whenever the mobile node is detected it will send the information to the router and the router sends to the nearby bulbs on command.

The cup carbon code files for the shown example can be found at: https://github.com/likhithkanigolla/IIITS-Demo/tree/main/FinalCupCarbon-Demo

PythonPDEVS

PythonPDEVS Introduction

PythonPDEVS is a Python implementation of the **Parallel Discrete Event System Specification (PDEVS)** formalism. It provides tools to model and simulate systems as a combination of **atomic** and **coupled models**, enabling hierarchical and modular system design.

Atomic Models:

- The **basic building blocks** of a DEVS system.
- Define **state variables**, **internal and external transitions**, **output functions**, and **time advance** functions.
- Handle input and output events independently.
- Each atomic model simulates the behavior of a single component.

Coupled Models:

- Allow **combining multiple atomic models (or other coupled models)** into a larger system.
- Define the **connections** (couplings) between submodels.
- Provide **hierarchical composition**, enabling complex systems to be built from simpler components.
- Do **not** define behavior themselves, but manage the structure and communication of submodels.

In PythonPDEVS, you build a system by first creating **atomic models** (with logic and state changes), then integrating them into a **coupled model** to define how they interact and exchange events.

PythonPDEVS

PyDEVS Motion Light System

This project implements a motion-activated lighting system using the Parallel DEVS (Discrete Event System Specification) formalism through PyDEVS.

System Overview

The system simulates a smart lighting setup where motion sensors trigger lights through a routing component. It demonstrates the principles of discrete event simulation with loosely coupled components communicating through well-defined interfaces.

File Structure and Concepts

`model.py`

This file defines the top-level coupled DEVS model (`GeneratedModel`) that composes the entire simulation. In DEVS formalism, a coupled model contains:

- Components (atomic or coupled models)
- Input and output ports
- Coupling relationships between components

The `GeneratedModel` class instantiates the motion sensor, router, and light components, then establishes the connections between them to form the complete system topology.

`motionsensor.py`

This file implements the `MotionSensor` atomic DEVS model that simulates a physical motion sensor. Key concepts:

- `MotionSensorState`: Maintains the state variables for the sensor component
- `MotionSensor`: Implements the DEVS atomic model with:
  - `__init__`: Sets up the sensor with its name, initial state, and ports
  - `timeAdvance`: Determines when the next internal transition will occur
  - `outputFnc`: Generates output events when motion is detected (randomly produces 0 or 1)
  - `intTransition`: Updates the component's state after an internal transition
  - `extTransition`: Handles external events (input messages from other components)

The motion sensor generates randomized detection events (0 or 1) that are sent to the router component through its output port.

`router.py`

This file implements the `Router` atomic DEVS model that processes and routes messages between components. Key concepts:

- `RouterState`: Maintains the state variables for the router component
- `Router`: Implements DEVS functions for routing messages:
  - Routes messages from the motion sensor to the appropriate light
  - Acts as a middleware component that can implement filtering or transformation logic
  - Follows DEVS protocol with timeAdvance, outputFnc, intTransition, and extTransition functions

The router demonstrates the concept of message passing and decoupling in event-based architectures.

`light.py`

This file implements the `Light` atomic DEVS model that represents a physical light that turns on and off based on motion sensor input. Key concepts:

- `LightState`: Maintains the state variables for the light component
- `Light`: Implements the DEVS atomic model:
  - Accepts input from the router
  - Changes state based on the motion detection value
  - Simulates the behavior of a light (on/off) based on sensor readings
  - Implements timeouts to turn off after a period of no motion

The light component shows how output devices respond to events in a discrete event system.

`simulate.py`

This file contains the simulation execution code. Key concepts:

- Creates an instance of the `GeneratedModel`
- Configures the PyDEVS simulator with parameters:
  - Termination time (how long to run the simulation)
  - DEVS formalism type (classic DEVS)
  - Verbosity settings
- Redirects output to a log file
- Executes the simulation via `sim.simulate()`

This file demonstrates how to set up and run a discrete event simulation using the PyDEVS library.

`style.css`

This file contains the styling for the web-based visualization of the system. It defines:

- Layout of the visualization interface
- Styling of components and connections
- Interactive elements for the visualization dashboard

The CSS supports the visualization aspect of the simulation, helping users understand the system structure and behavior.

DEVS Conceptual Framework

The system follows the DEVS formalism which includes:

1. **State-Based Modeling**: Components maintain states that change over time
2. **Event-Driven Execution**: System advances based on events rather than fixed time steps
3. **Hierarchical Composition**: Simple models are combined to create complex systems
4. **Well-Defined Interfaces**: Components interact only through input/output ports
5. **Time Advance Function**: Each component determines when its next internal event occurs

Simulation Flow

1. The motion sensor periodically generates motion detection events (0 or 1)
2. These events are sent to the router component
3. The router forwards the events to the appropriate light component
4. The light changes its state (on/off) based on the received values
5. The simulation continues until the specified termination time is reached