SEVENTH FRAMEWORK PROGRAMME

FP7-ICT-2013.3.4

Project acronym:	**AMADEOS**
Project full title:	Architecture for Multi-criticality Agile Dependable Evolutionary Open System-of-Systems
Grant Agreement no.:	610535

Partners:

[1] Università degli Studi di Firenze

[2] Technische Universitaet Wien

[3] Université Grenoble Alpes

[4] ResilTech S.r.l.

[5] Thales Nederland Bv

[6] European Network For Cyber Security Cooperatief Ua

SUPPORTING FACILITIES USER GUIDE

Revision of the document:	01
Responsible partner:	RES
Contributing partner(s):	
Reviewing partner(s):	-
Document date:	22/03/2016
Dissemination level:	RE

Change Records

Revision	Date	Changes	Authors	Status[1]
0.0	12/02/2016	Draft version of the document	RES	NV
0.1	22/03/2016	Updated version for AB meeting	RES	NV

[1] V - Valid, NV - Not Valid, R - Revision

TABLE OF CONTENTS

LIST OF FIGURES

Definitions and Acronyms

Acronym	Definition
SoS	System of Systems
EV	Electric Vehicle
CP	Charging Point
CSO	Charging Station Operator
LMO	Load Management Optimizer
CS	Constituent System

1 INTRODUCTION

The aim of this document is to provide a user guide for the supporting facilities used to design and simulate the behaviour of a generic SoS, according to the AMADEOS concepts.

The selected tool for designing a generic SoS is Blockly [1]. Chapter **Error! Reference source not found.** will present the main features of Blockly and will provide some examples of its use.

Blockly was selected because it meets the following requirements in the AMADEOS project.

- It can be used as a generic SoS design tool
- It is intuitive and simple for the SoS designer
- It can be used to specify and track requirements
- It can be used to model a SoS from different viewpoints/building-blocks:
- Architecture, Communication, Dependability, Dynamicity, Emergence, Evolution, Interface, MAPE, Multi-criticality, Security, and Time,
- It is able to exploit the SysML profile (see D2.2 [2], and D2.3 [3] for the final version)
- It can be used to check compatibility between constituent systems (check of RUMI on a syntactical level)
- It implements constraints to avoid design errors.

Blockly makes the design intuitive and simple because it is based on visual blocks.

Furthermore, to avoid design mistakes:

- it can warn messages/interface mismatch between SoSs
- it can warn criticality mismatch between services.

Blockly has features to support code and XML generation. Python is the language selected for code generation from Blockly.

Python has been chosen as it has many powerful features:

- Extensive standard library and powerful data types
- Friendly string manipulation
- Easy to begin with, easy to read
- Portability
- Can be compiled and integrated in different languages like C, C++, Java and C#
- Big supporting community

The rest of the document is structured as follows: Section 2 describes the use of supporting facility tool to model an SoS using AMADEOS concepts. Section 3 describes how to simulate a SoS using the code generated from the SoS model. Section 4 describes how a model made through the supporting facility can be exported to PlantUML.

2 SUPPORTING FACILITY TOOL USING BLOCKLY

This chapter describes the main features of supporting facility tool and how to design SoSs. Blockly is an open source library for building visual programming editor [1]. It is used to build supporting facilities to design a generic SoS, in accordance with the AMADEOS conceptual model.

Blockly's main features are:

- Fast, and only a web browser is needed.
- Intuitive and simple for the designer.
- Easily extendable with custom blocks.
- Able to check constraints (both user defined, and pre-defined) and warn user when user makes a mistake during modelling.
- Support code and XML generation.

Supporting facility tool with Blockly is aimed to provide a simple and intuitive interface to model a SoS with minimal training to the designer. It has been customized to be used for SoS modelling by importing the SysML profile for Amadeos. All version of the supporting facility tool can be accessed at the following link:

http://blockly4sos.resiltech.com.

The blocks in the tool are imported from the AMADEOS SysML profile provided by an expert. Below is the flow of using the supporting facility:

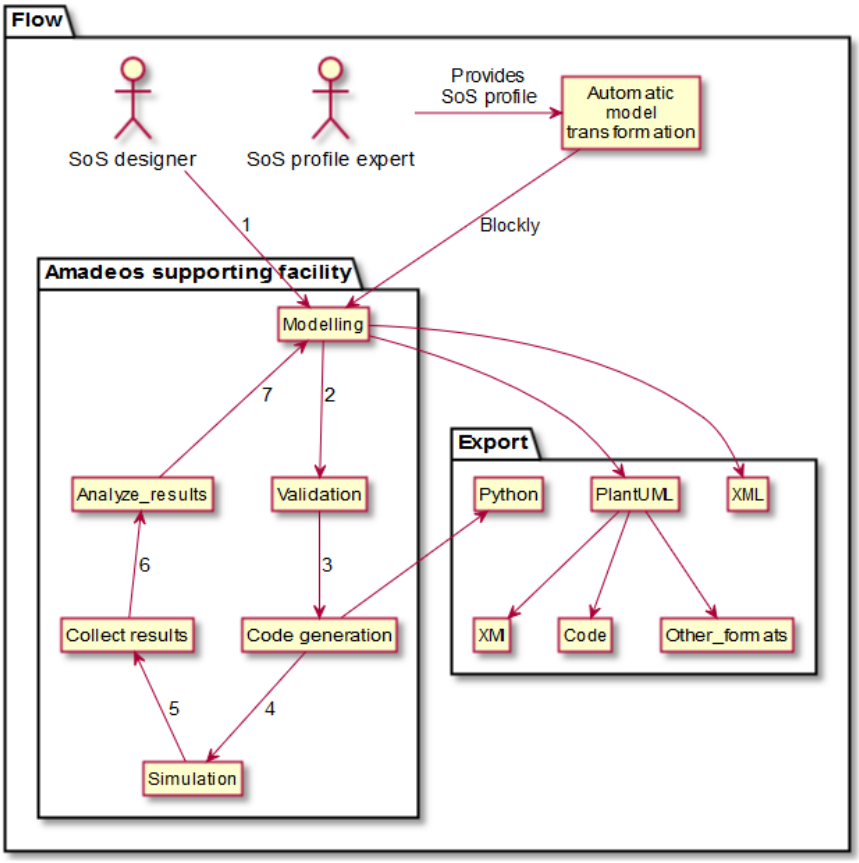

Figure 1 - Flow of using the supporting facility

The SoS designer need not have any knowledge of SysML/UML; the only prerequisite is high-level knowledge about the profile and knowledge of the supporting facility tool.

The instantiated models created using the supporting facility can be exported to PlantUML (as an object diagram) and Python executable code. PlantUML is a simple textbased UML format which can be readily integrated with many tools; the full list of supported tools is available at http://plantuml.com/running.html.

Example, the PlantUML model can be **viewed** in Eclipse (Figure 2) using the plugin provided at (http://plantuml.com/eclipse.html). Though, full interpretability between tools is an ongoing research topic and is under investigation.

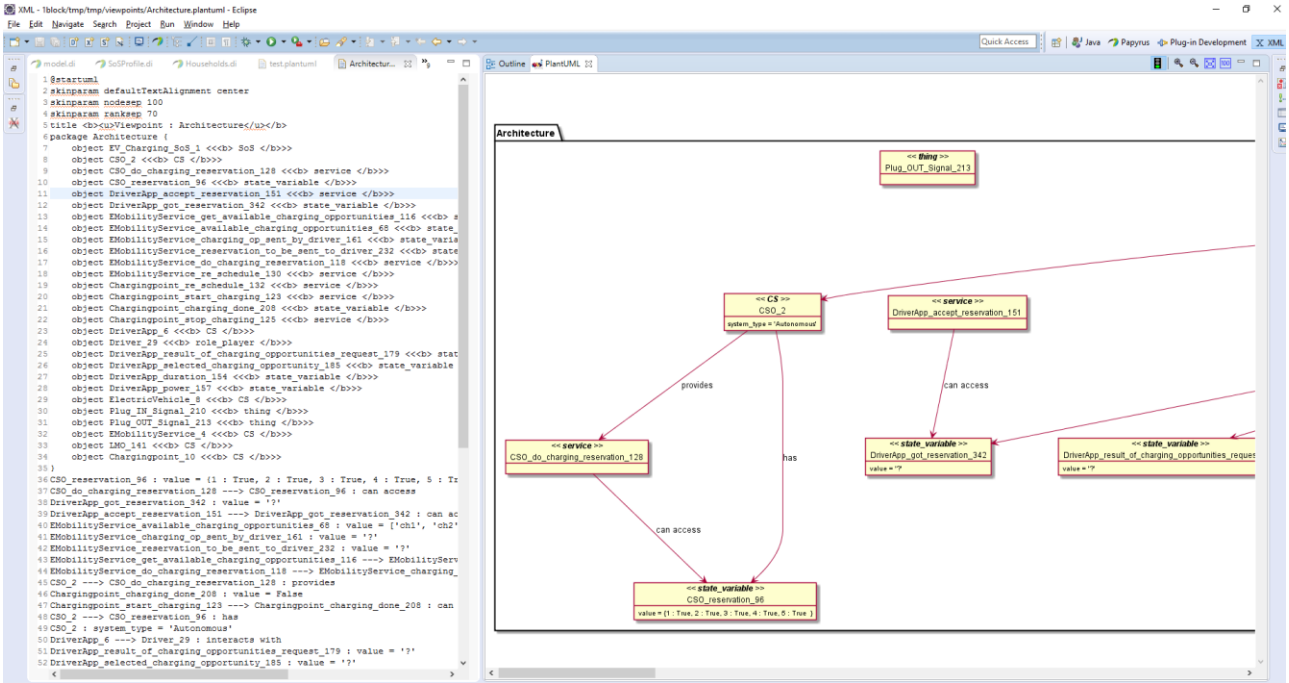

Figure 2 - View PlantUMl in Eclipse

2.1 DESIGN A SOS USING THE SUPPORTING FACILITY TOOL (STATIC MODEL)

In order to design a SoS the user have to access with a browser to the Blockly web page (http://blockly4sos.resiltech.com).

Figure 3 shows the initial page of Blockly latest version.

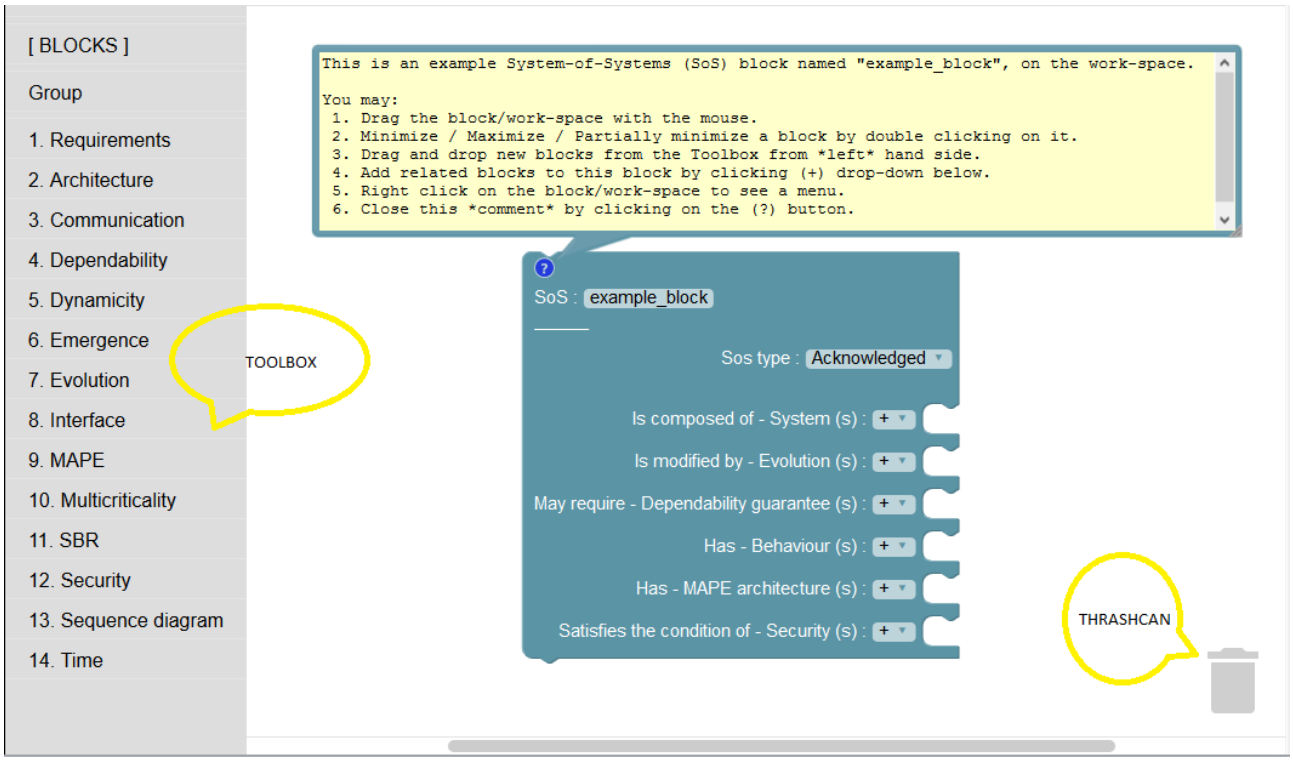

Figure 3 –Supporting facility tool homepage (with toolbox and thrashcan).

From this page the user can access to all the features of Blockly.

On the left of the page there is the list of all viewpoints blocks needed to design a SoS.

For example, by clicking on “2. Architecture”, all the blocks related to the architectural view of a SoS are shown (see Figure 4).

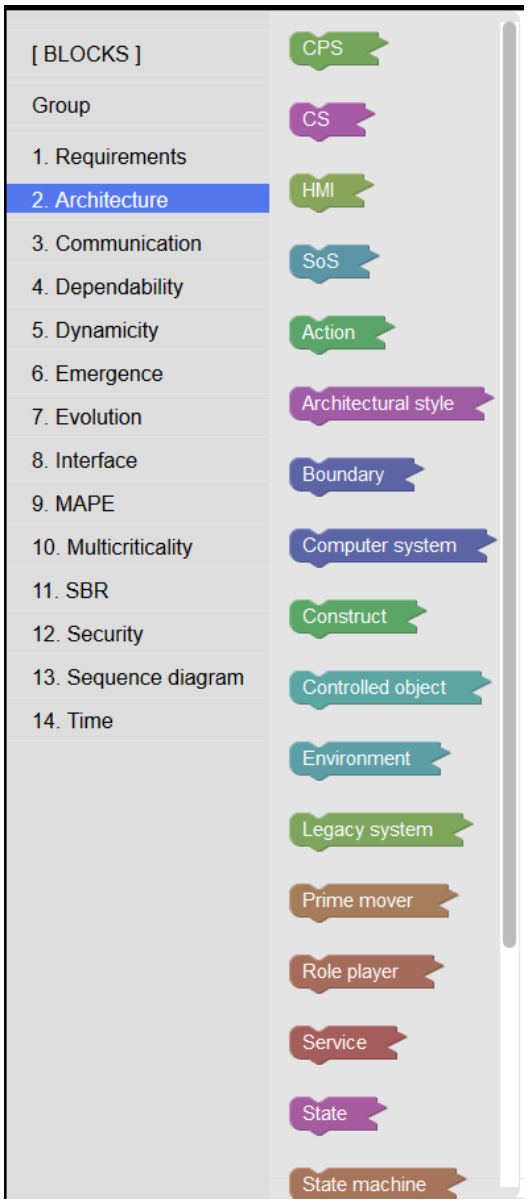

Figure 4 – Architecture viewpoint related blocks on toolbox

Similarly, clicking on the others items of the left menu, all the related blocks are shown.

2.1.1 Add and delete a block in a SoS

There are two ways to add a block in a SoS:

1. Drag and drop from the menu
2. From the dropdown on an existing block

For example, if the designer wants to add a Constituent System (CS), he/she can click on “2.Architecture” on the left menu, select the CS block (when selected, the block is yellow highlighted) and drag and drop the block as shown in Figure 5.

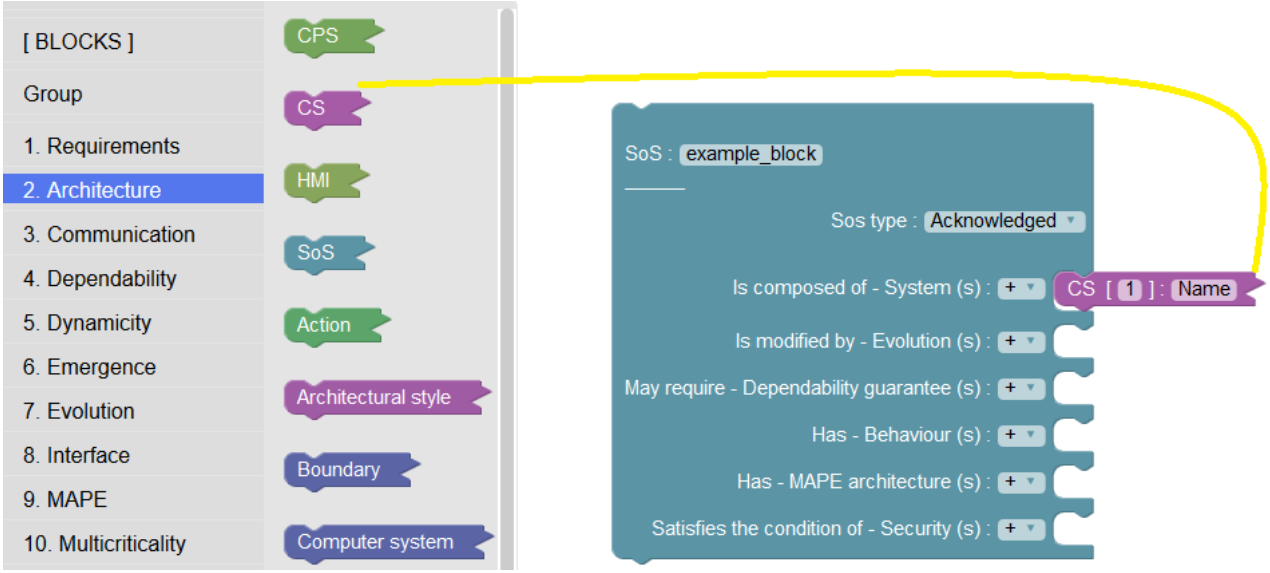

Figure 5 – Drag and drop a block.

A faster way to add a block is through an existing block. For example, with reference to Figure 3, if the designer wants to add a CS block to a SoS, the user can click on the "+" related to "Is composed of – System (s)" on the SoS block and select the CS item from the box that appears as shown in Figure 6.

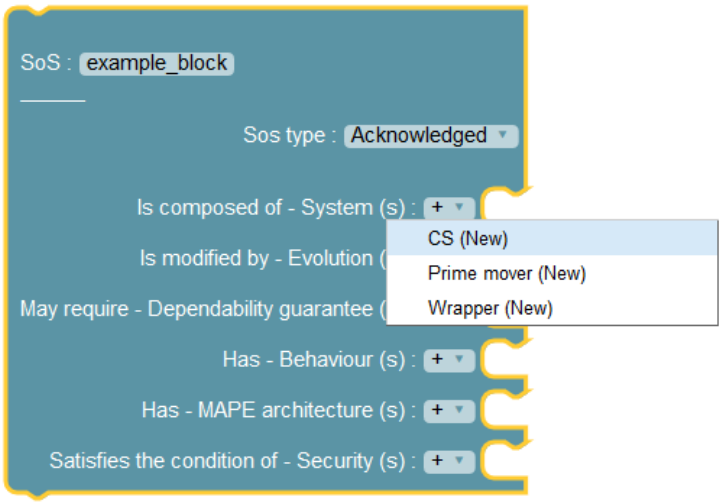

Figure 6 – Adding a block CS to "example_block" from dropdown menu

The result is the same as dragging and dropping a block.

To delete a block the designer can right click on the block to delete and press the keys "Canc/Delete" from the keyboard. A dialogue box will appear as shown in Figure 7. To delete the block the user can click on "Ok".

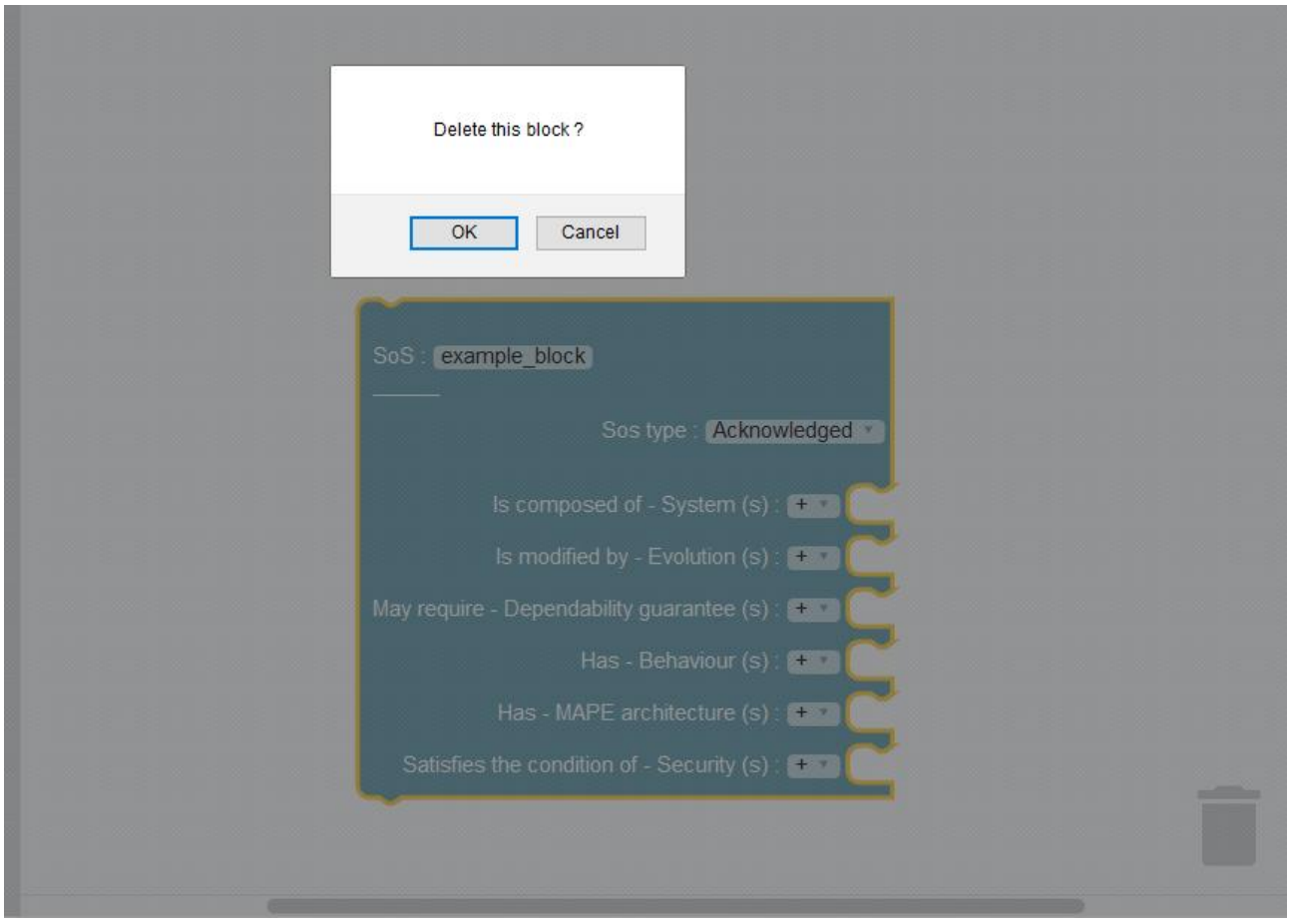

Figure 7 – Delete a block (1).

Another way to delete a block is select a block, and right click, and select the item “Delete Block” from the box that appears as shown in Figure 8.

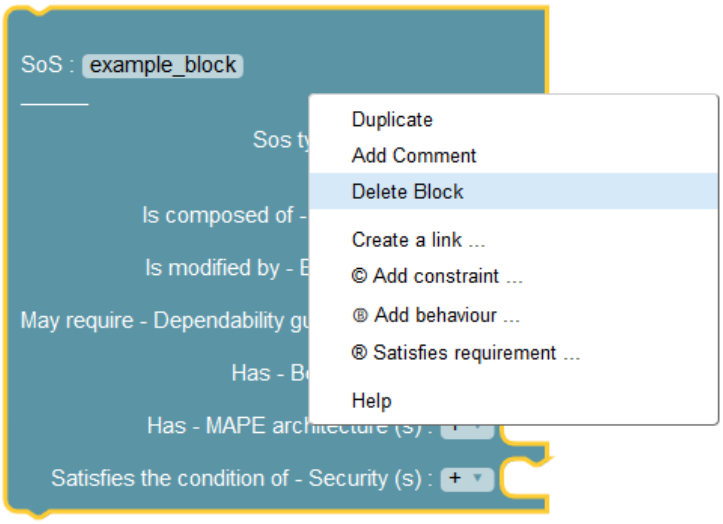

Figure 8 – Delete a block (2).

2.2 REQUIREMENT MANAGEMENT

In this section, we present an example that explains requirements management in a SoS. Before adding actual blocks, a good SoS design starts with SoS requirements.

Requirements can be of two types (1) simple reqirement, (2) Composote requirement. A simple reqirement is a single requirement consisting of requirement description, ID, and title; where as a composite requirement may consist of several simple/composite requirements (Figure 9).

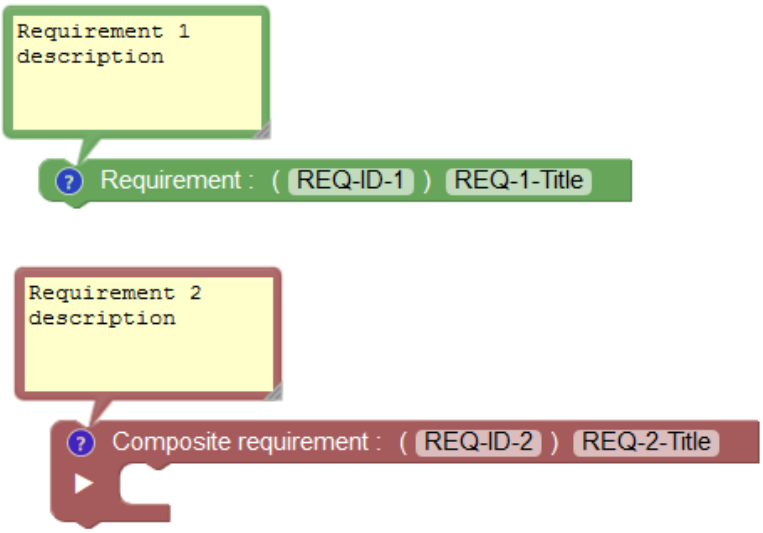

Figure 9 - Types of requirements

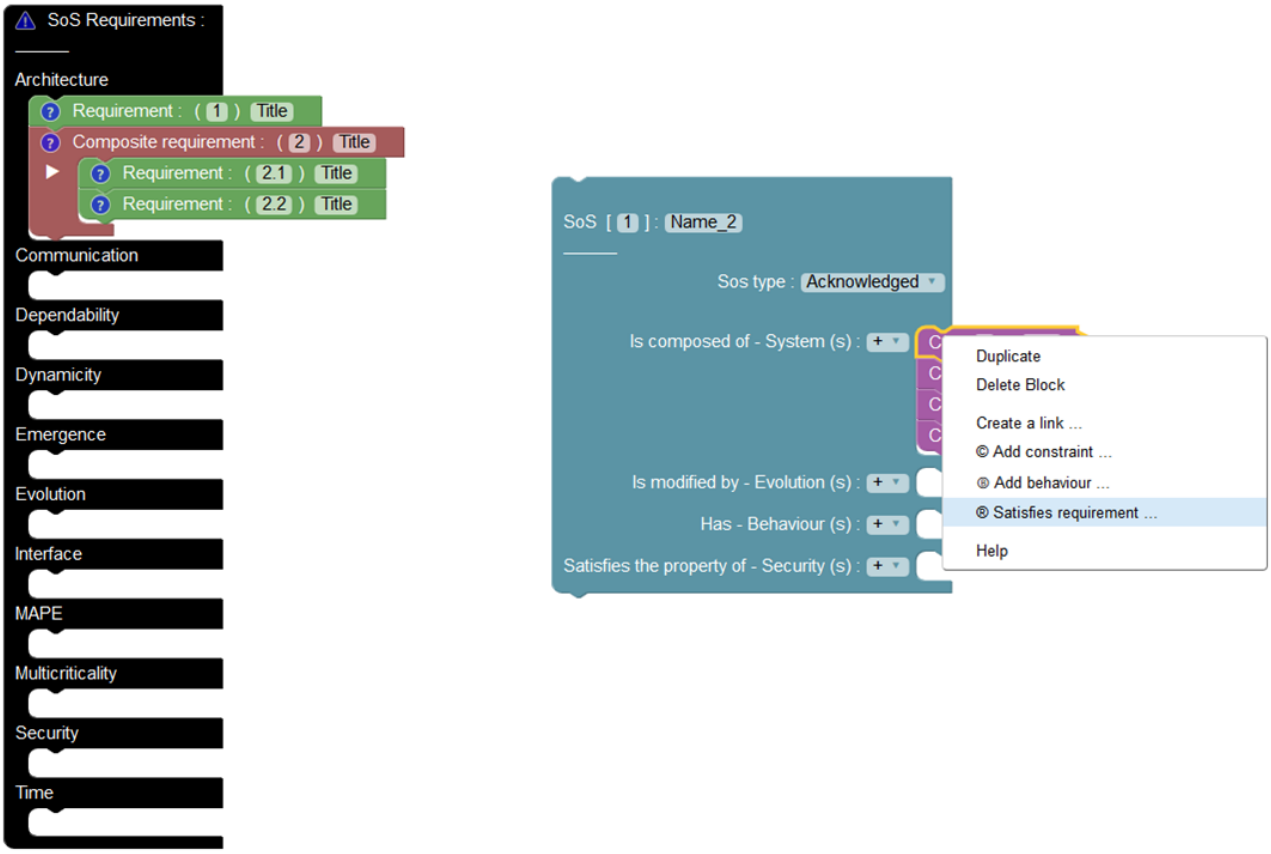

Figure 10 – Requirements management (1).

Requirements can be specified for each viewpoint/building-block as given in Figure 10. In Figure 10 we show that any block can satisfy a requirement. In this example a CS can satisfy architectural requirement 2.1 as shown below.

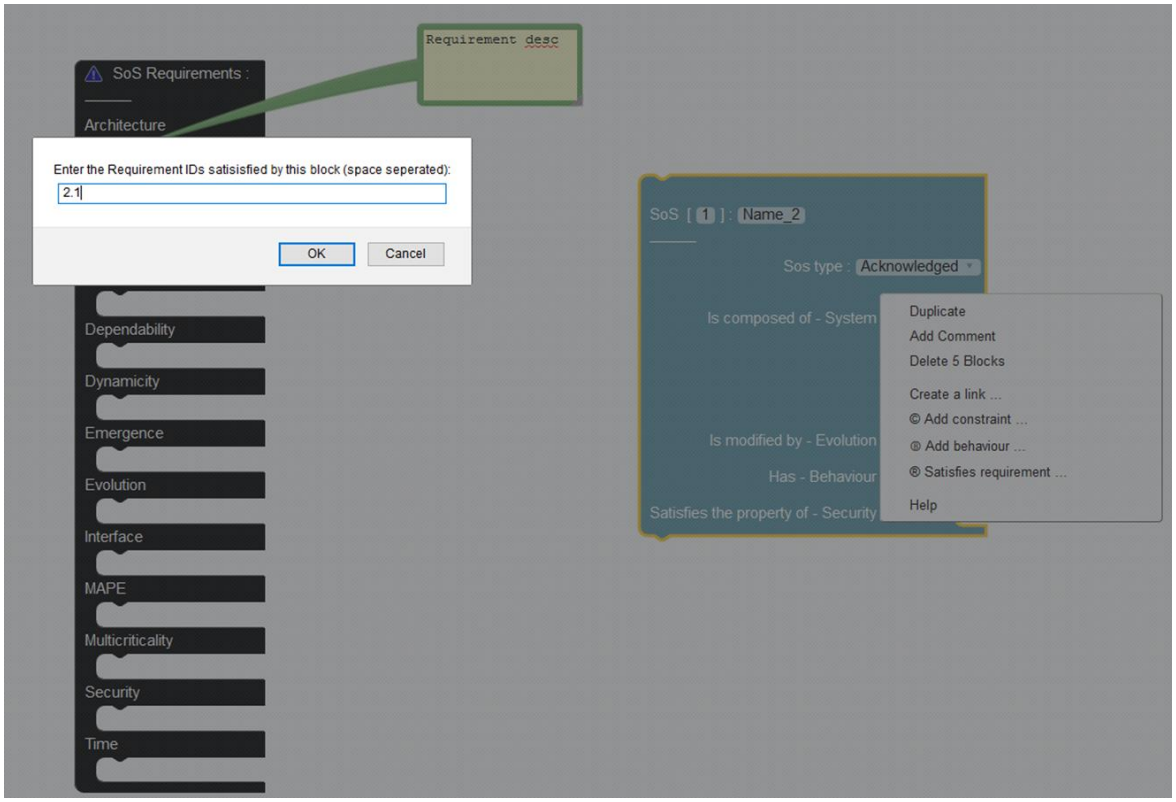

Figure 11 - Requirements management (2).

Requirements are identified by IDs and the tool provides traceability to requirements (i.e.: which requirements are satisfied by a block, and which blocks satisfy a given requirement) as given below:

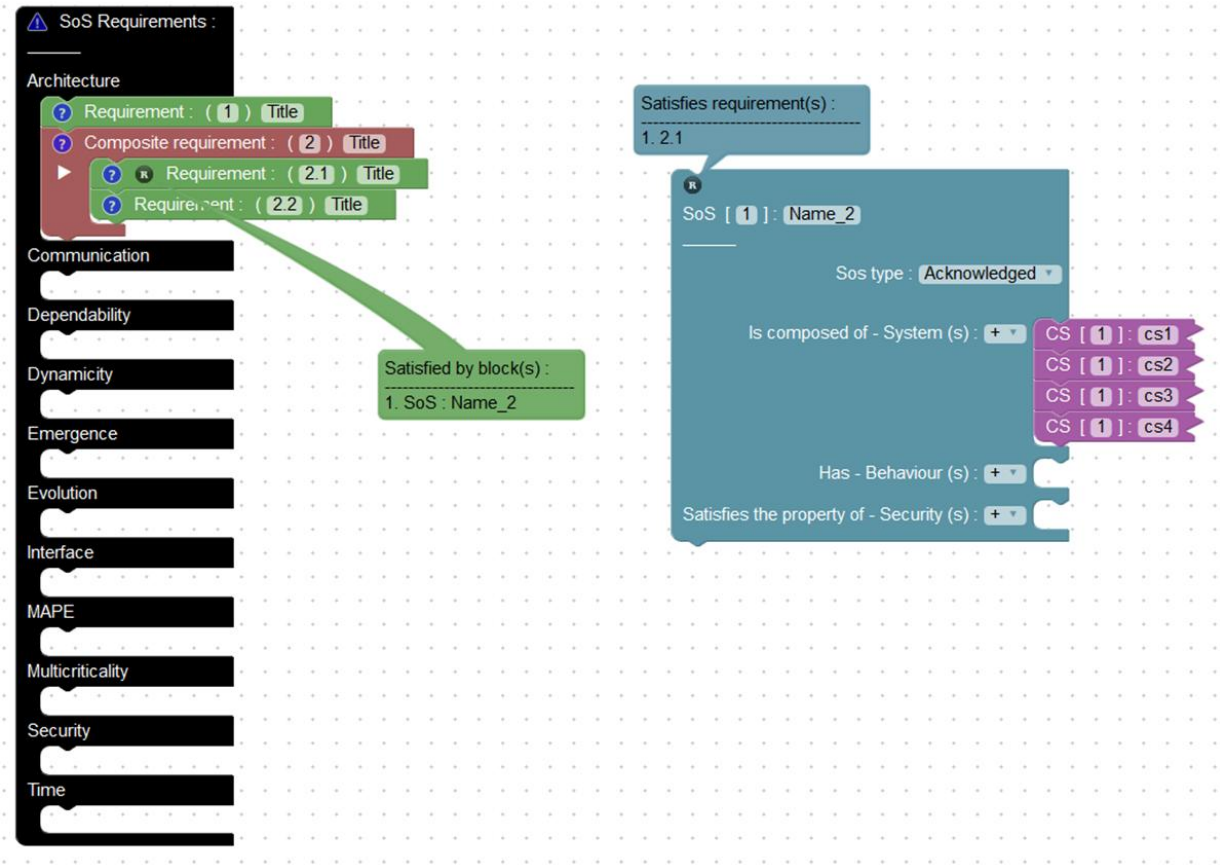

Figure 12 - Requirements management with traceability.

2.2.1 Collapsing blocks

A block can be collapsed by double clicking on it. A non-collapsed block when double clicked shows only the blocks defined in the block (i.e. partially collapsed). When a partially collapsed block is double clicked, it becomes fully collapsed block (as shown below).

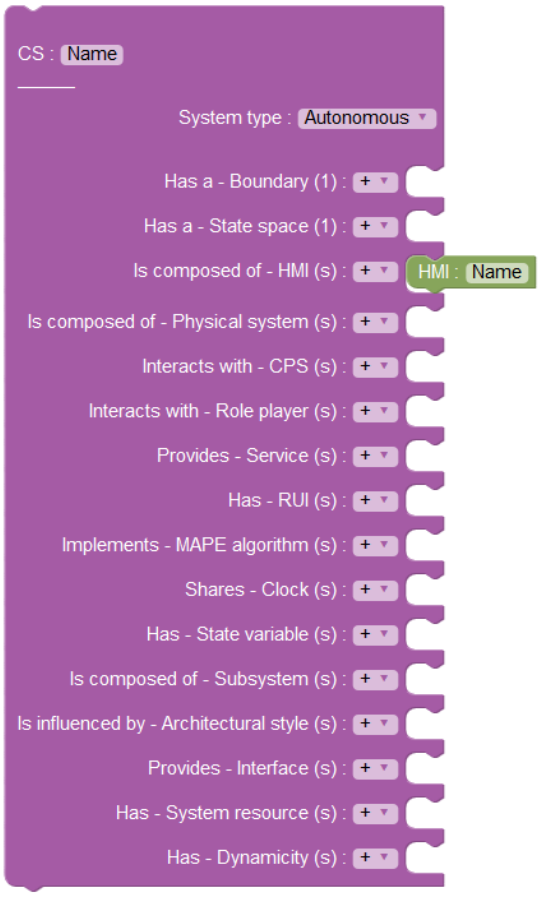

Figure 13 – A non-collapsed block.

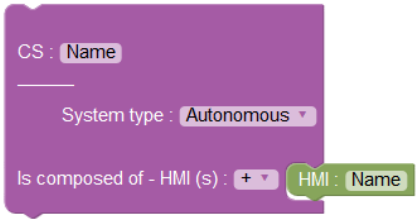

Figure 14 – Partially collapsed block.

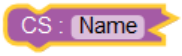

Figure 15 – Fully collapsed block.

2.2.2 Constraints in blocks

Design constraints can be written in JavaScript to avoid design mistakes. The failure of constraints will result in block color changed to black. For example:

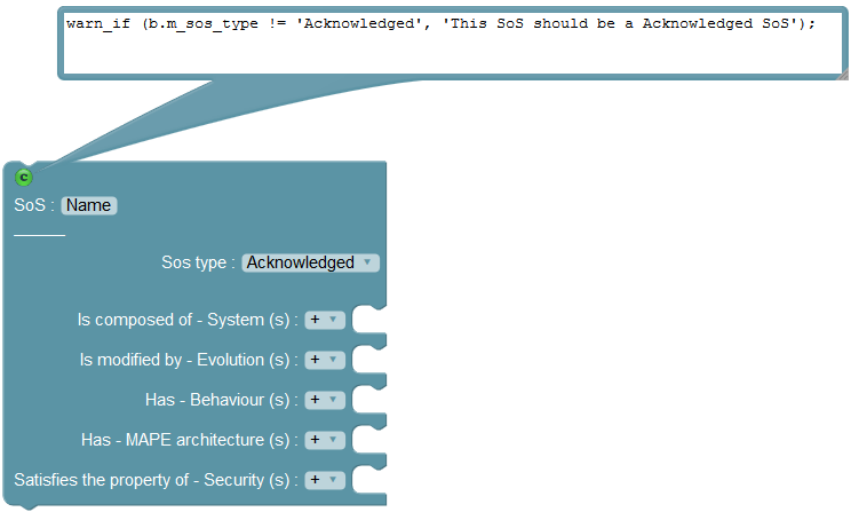

Figure 16 – A block with constraints satisfied.

Figure 17- A block with failed constraints

(warn_if (b.m_sos_type != 'Acknowledged', 'This SoS should be a Acknowledged SoS');).

Whenever the constraints are satisfied, the block changes back to its normal color. Constraints are written using member variables of the block. Each member variable of block starts with "**m**_" or with "**relationship**"_"**type**" name. Also, each block has a variable "***of_type"*** which represents the block type.

For example a SoS block given below has the following variables:

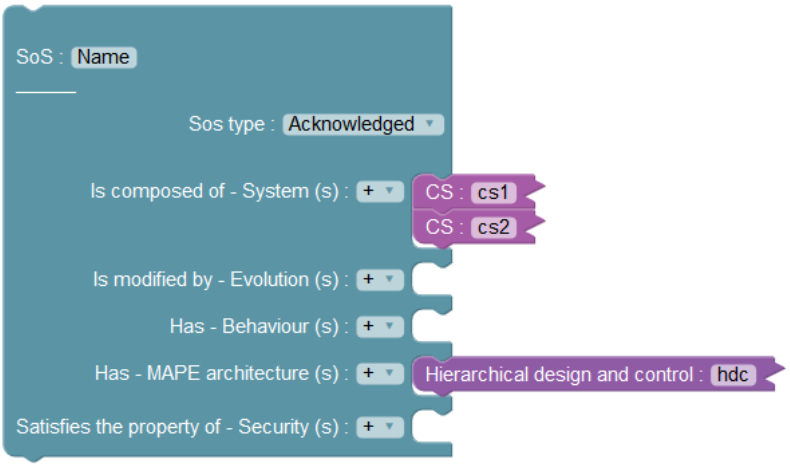

Figure 18 – Example of variables in a block (1).

1. ***m_sos_type*** = 'Acknowledged'
2. ***is_composed_of_system*** = ***m_system*** = [array of 2 objects]
3. ***is_modified_by_evolution*** = ***m_evolution*** = [] // empty array
4. ***has_behaviour = m_behaviour*** = [] // empty array
5. ***has_mape_architecture = m_mape_architecture*** = [array with one element]
6. ***satisfies_the_property_of_security = m_security*** = [] // empty array

A block with single inputs, such as the block depicted in Figure 19, has variables:

1. ***m_system_type*** = 'Autonomous'
2. ***has_a_boundary = m_boundary*** = Boundary object

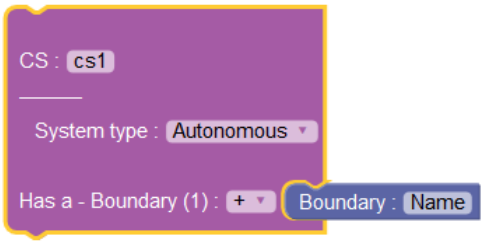

Figure 19- Example of variables in a block (2).

while the block in Figure 20 has ***has_a_boundary = m_boundary*** = None.

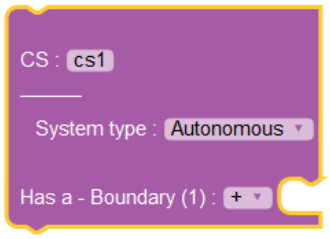

Figure 20- Example of variables in a block (3).

2.2.3 Grouping for modular SoS design

Compatible blocks can be grouped in to groups to modularize the design. For example, all CSs can be grouped together as shown below:

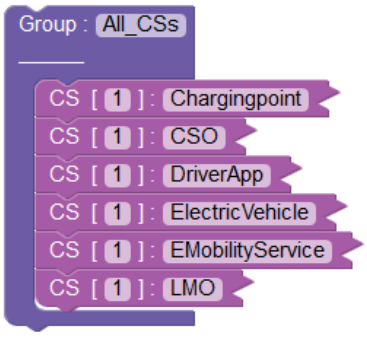

Figure 21 - Group of CSs

The group helps in organizing the model into meaningful groups. Also, when a block of a group blocks is refered, the group is shown to distinguish it from other blocks (***Figure 22***).

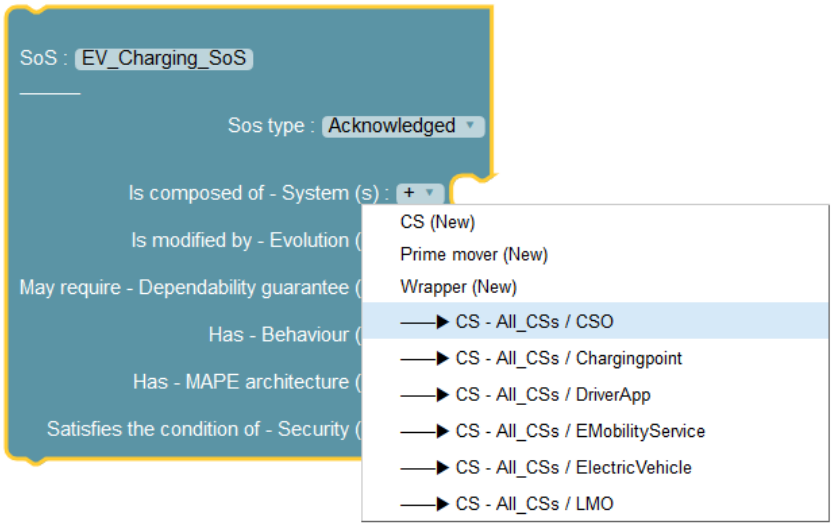

Figure 22 - Referring to the blocks of a group

2.2.4 Adding a link to CS

One way to design a SoS is by using links to existing blocks. Creating links can help reuse an existing block, this is different from copy-pasting a block. Links are reference to the actual block. Example: CSs can be created elsewhere and only links can be added to SoS (as shown below).

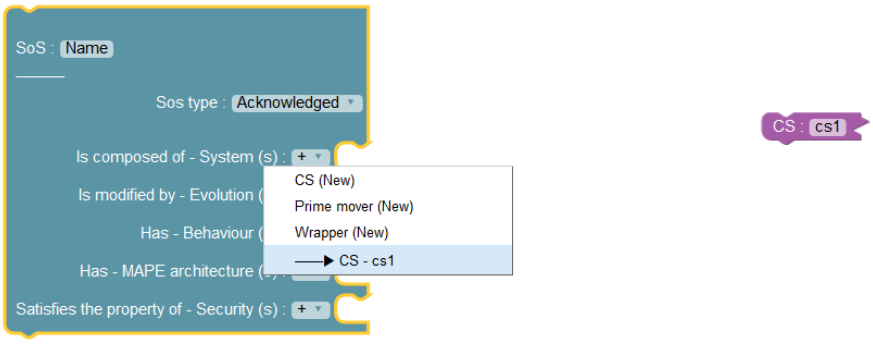

Figure 23- Reusing a block using links (1).

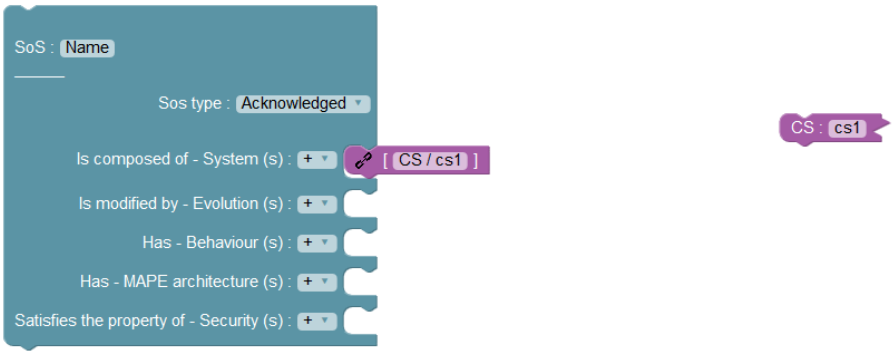

Figure 24- Reusing a block using links (2).

2.3 MODEL QUERYING

On large models it may be difficult to visualize the entire SoS, and then the need for custom viewpoints arises. Here below there is an example of a model with several CSs, RUMIs, and services.

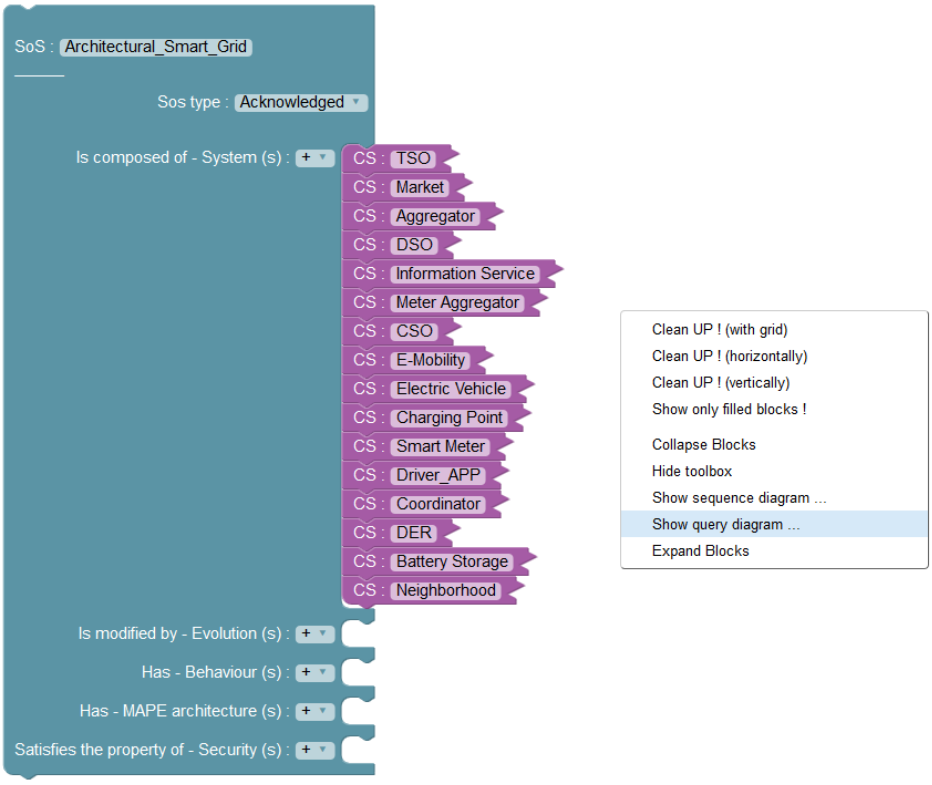

Figure 25 – Model querying large models (View query diagram).

To query a model, a user can right click on workspace and choose “show query diagram” as shown in (Figure 25).

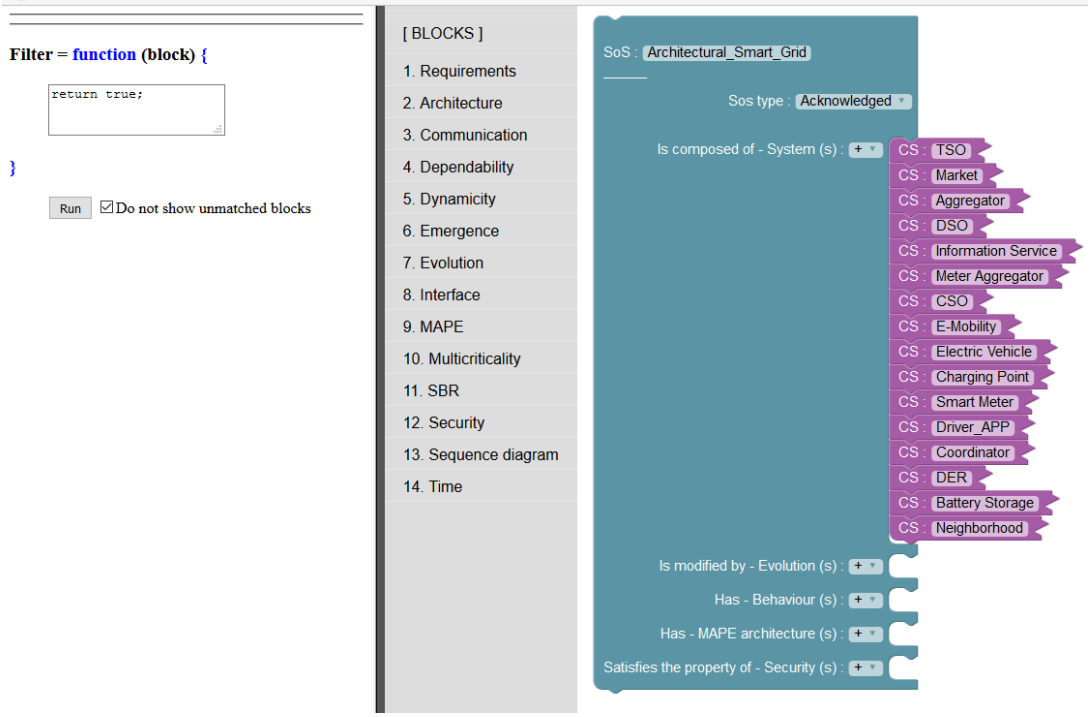

Figure 26 - Model querying large models (example query “select all”).

In the query diagram, user can write a filter function for querying the model. For example ***return true;*** indicates that no filtering is required (ie: show all blocks); which results in the following graph:

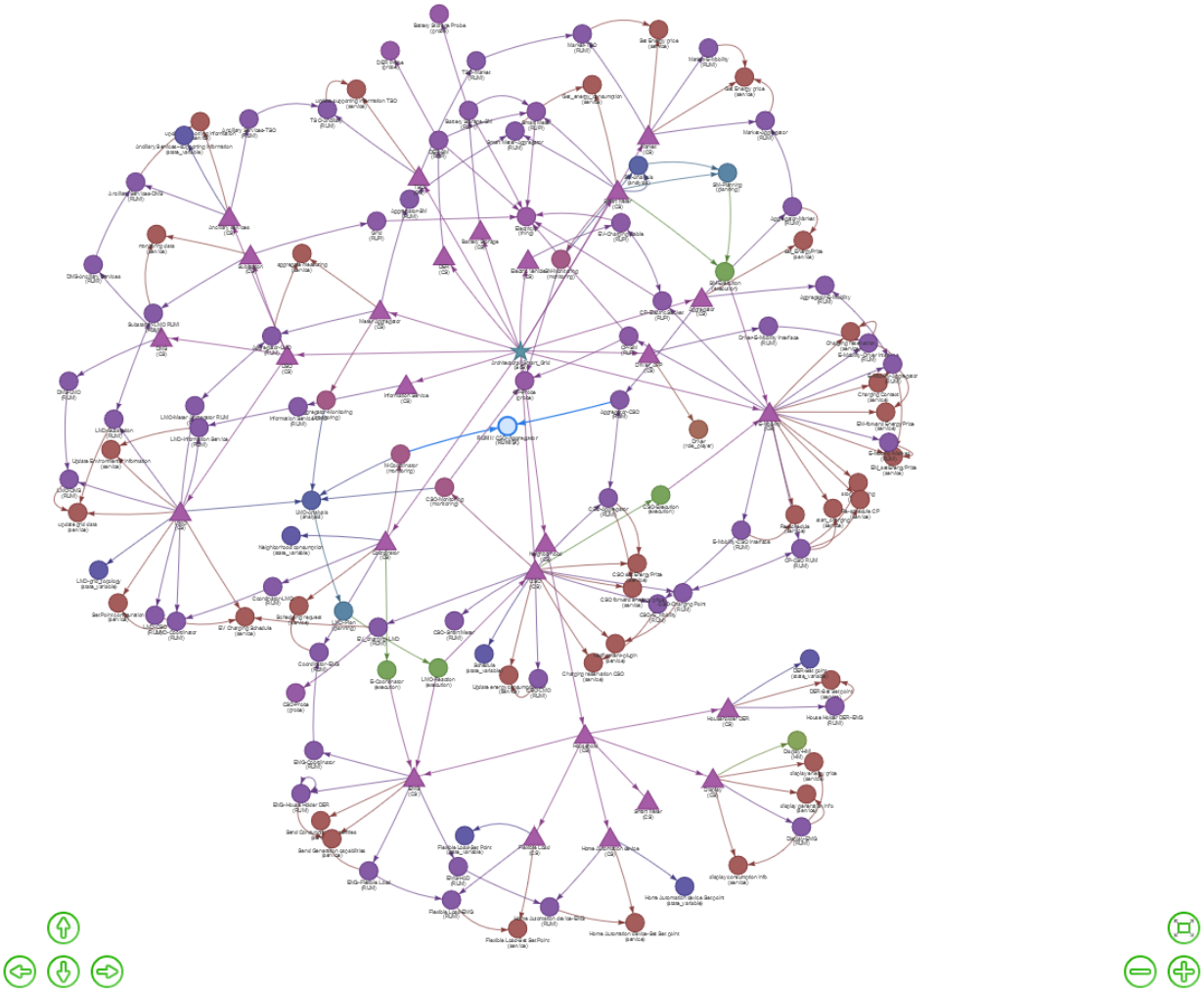

Figure 27 – Result of "select all" query.

Using the filter ***return b.of_type == 'RUMI'***, which indicates to highlight blocks of type RUMI, it returns the graph depicted in Figure 28.

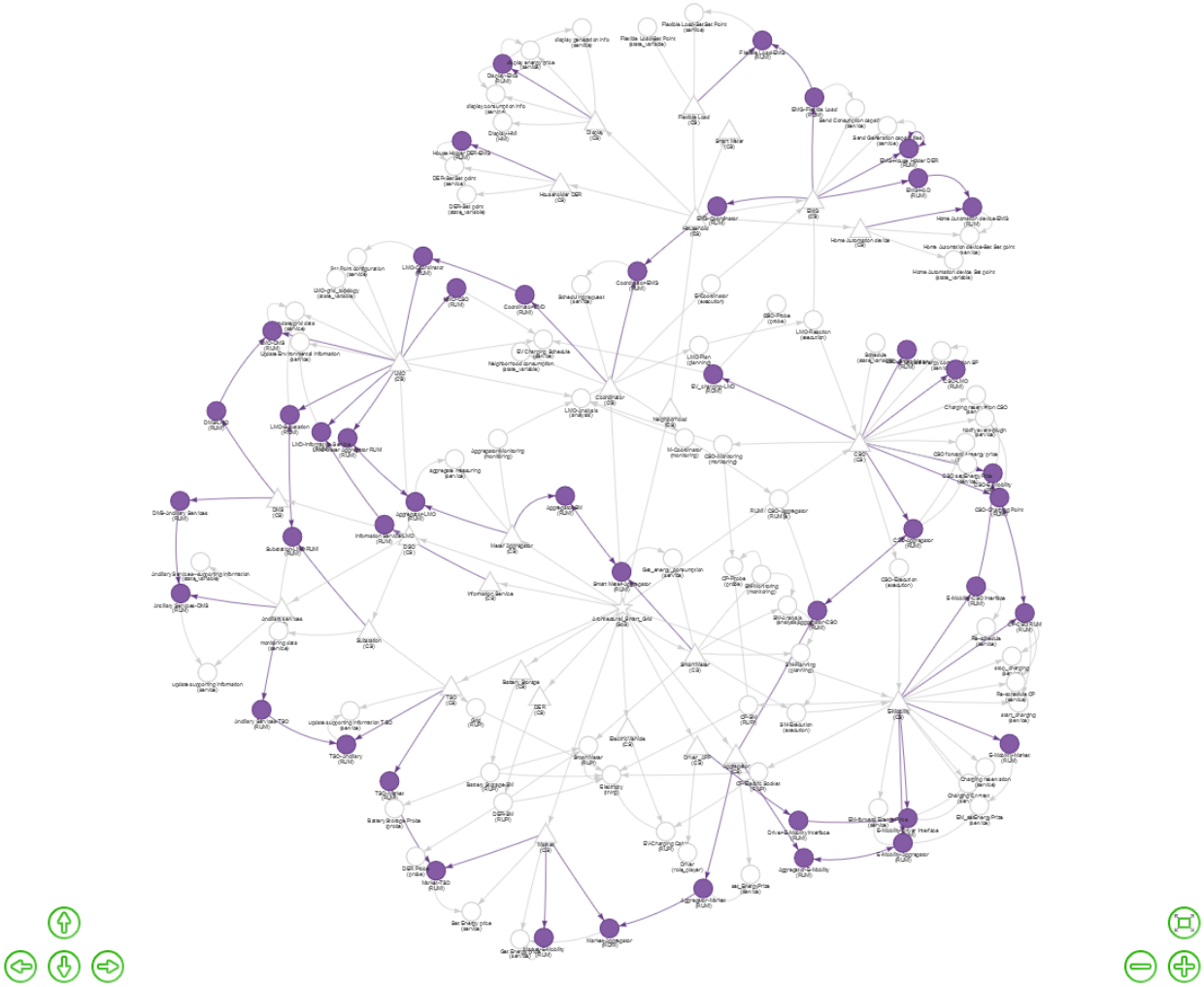

Figure 28 – Result of "return b.of_type == 'RUMI'***" query (select all RUMIs)***

Model querying helps in visualizing custom viewpoints of SoS and can be useful in identifying issues in SoS design.

2.4 SEQUENCE DIAGRAMS

In this section we present a simple example that explains how to create a sequence diagram for a SoS model.

Sequence diagrams can be used to test a specific scenario of SoS. To create a sequence diagram, the designer can right click on the workspace and select the item "**Show sequence diagram**" from the box, which appears as depicted in Figure 29. On the right side of the browser a Sequence Diagram Area appears as shown in Figure 30.

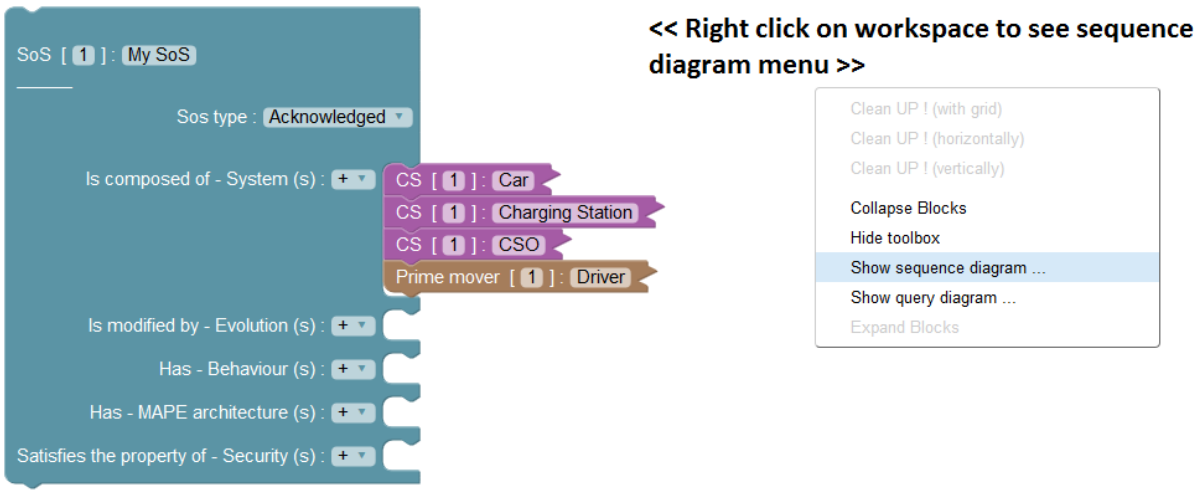

Figure 29 – Access to sequence diagram menu.

To create a sequence diagram, the designer must select the first (source) block with the mouse (in this example we select the Prime mover block) and the block is highlighted in yellow color.

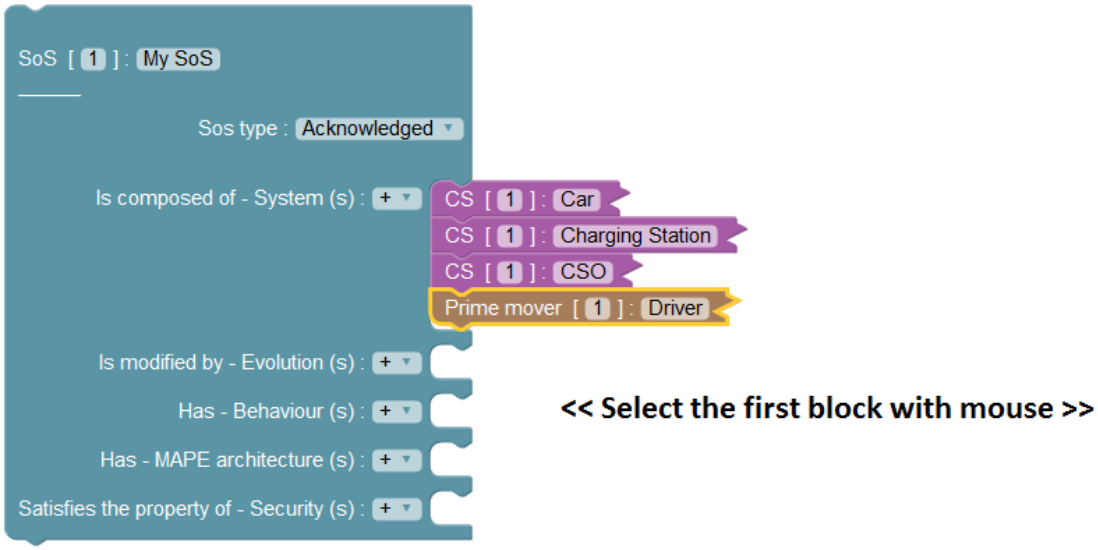

Figure 30 – Select the first block to create a sequence diagram.

Then select the destination block with “CTRL” key and mouse, as shown in Figure 31.

Now the designer must select the type of message (Sync or Async) from the right menu of the Sequence diagram area, and type the text of the message (in this example is MSG).

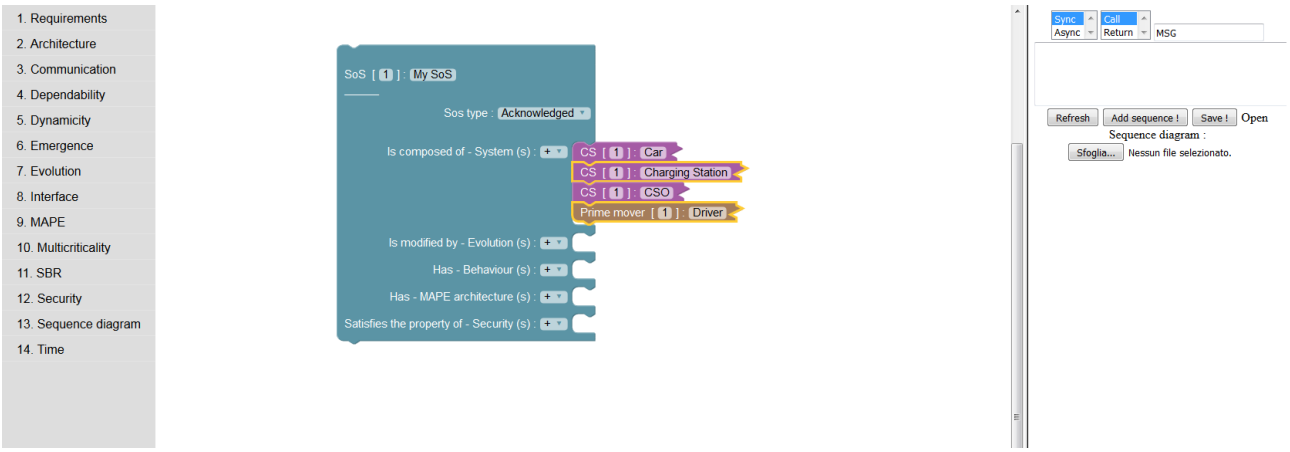

Figure 31 – Select the type of message to destination.

To generate the sequence diagram the designer must click on the button “Add sequence!” from the sequence diagram area and the sequence diagram is created as shown in Figure 32.

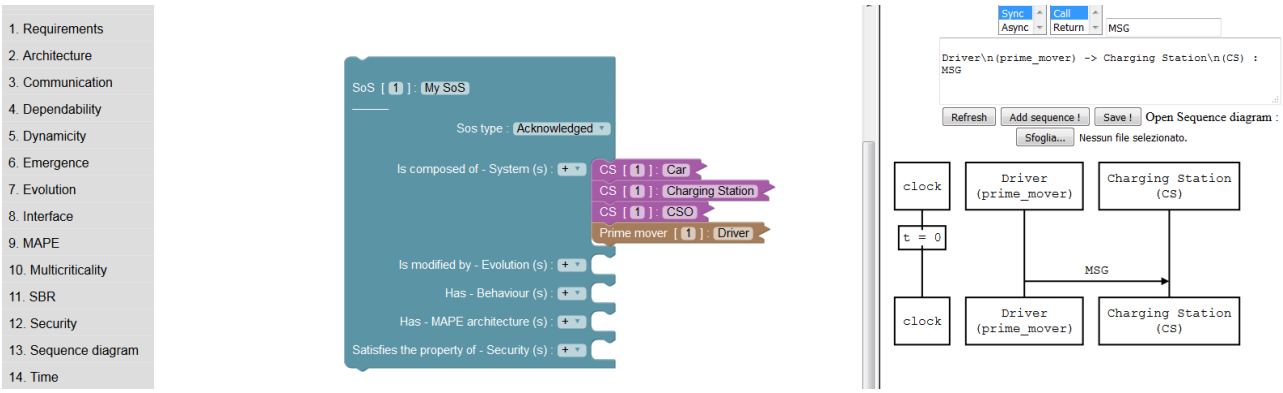

Figure 32 – Sequence diagram creation.

Currently the “TextArea” (containing sequence diagram code) is for debugging purpose only and please do not enter any text in it.

Another way to create a sequence diagram is through blocks from the flyout menu. These blocks allow creation of restricted sequence diagrams as per the AMADEOS framework, which can be easily converted to executable code. Figure 33 shows selection of sequence diagram related blocks in flyout. A sequence diagram may consist of sub-sequences and sub-sequences can consist of restricted actions required to build an executable sequence diagram.

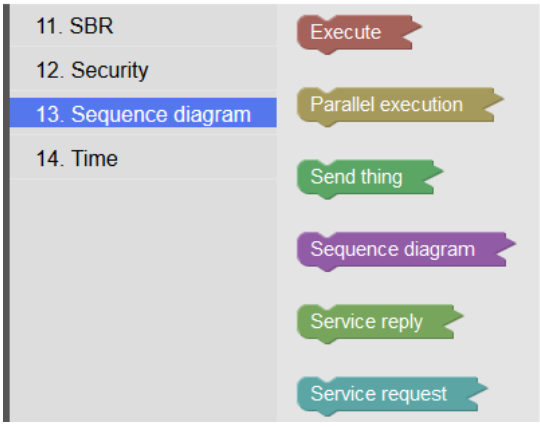

Figure 33 – Flyout menu for sequence diagram.

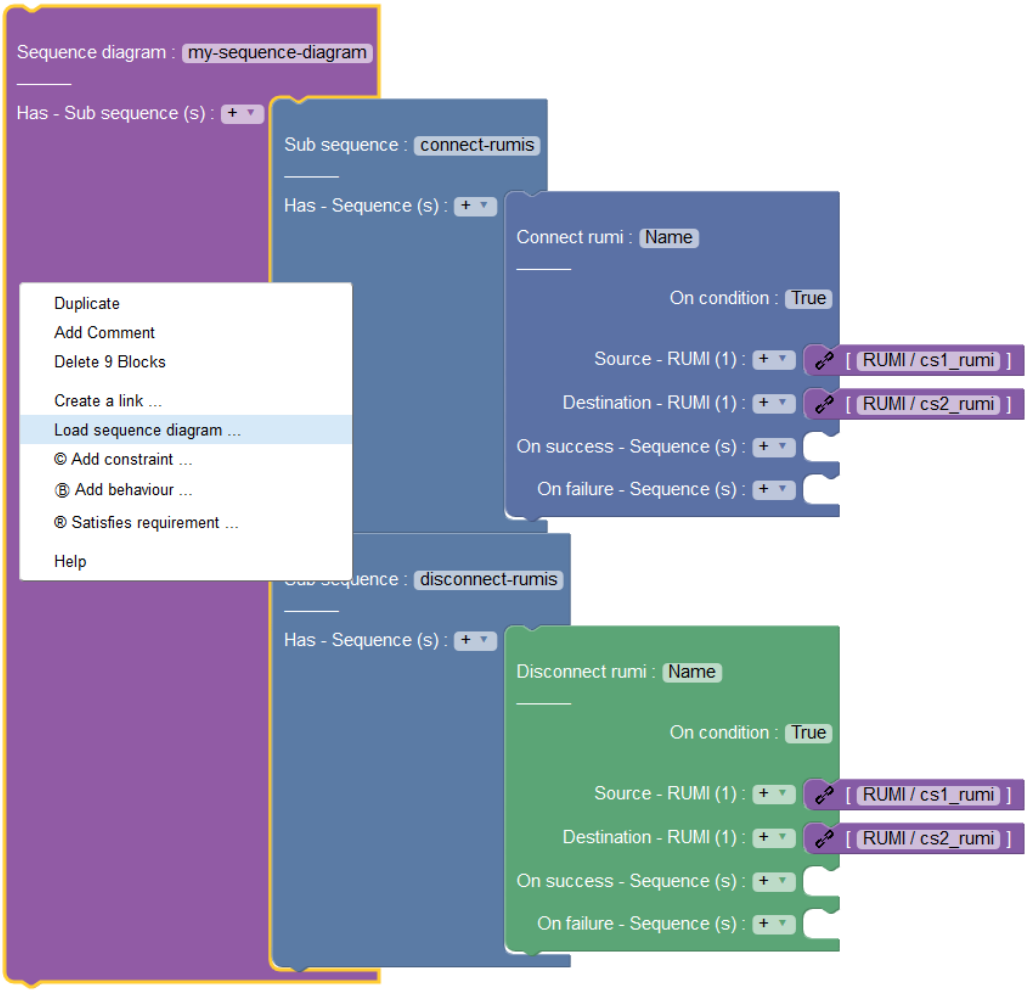

Figure 34 – Right click on sequence diagram and load it.

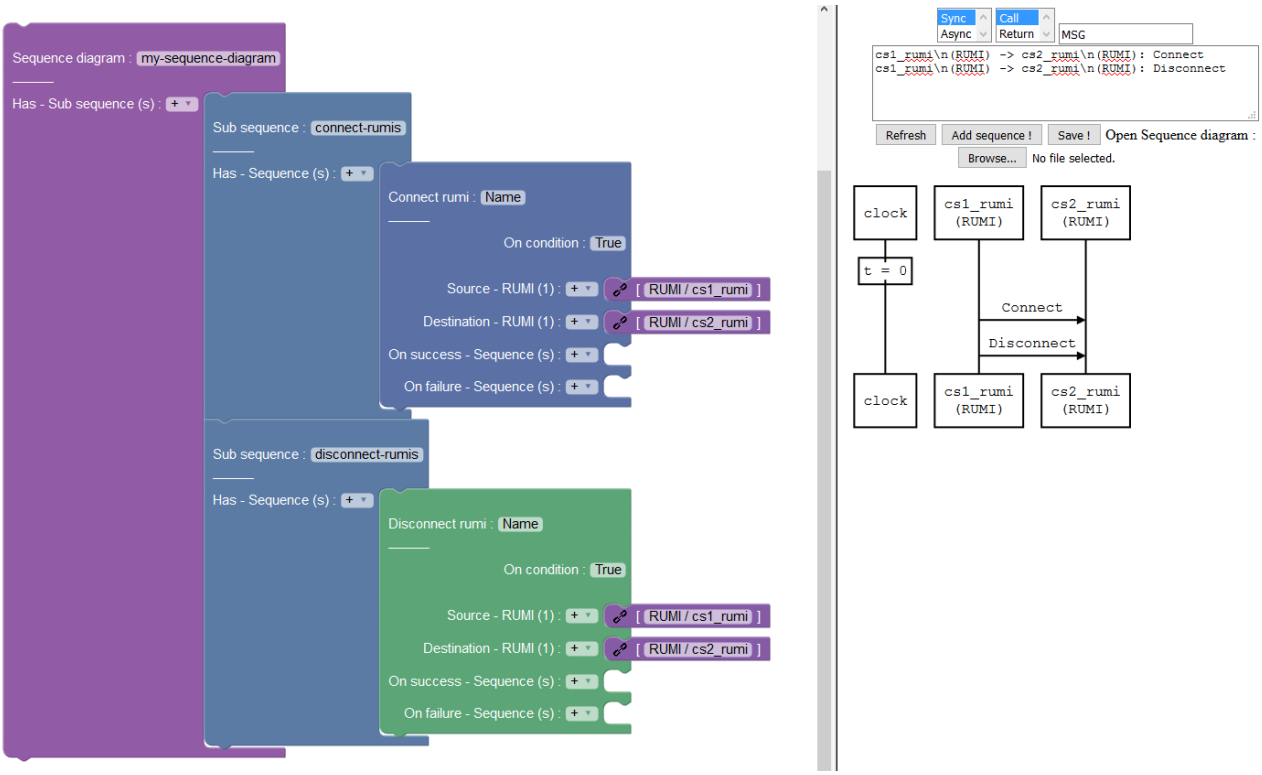

Figure 35- Example of restricted sequence diagram using AMADEOS concepts.

2.5 SERVICES AND RUMI (DYNAMIC BEHAVIOUR)

In this section we present how to add a Service and a RUMI to a CS. Double clicking on the CS, the CS block expands as shown in Figure 36.

To add a Service, the user can click on "+" related to the item "Provides – Service (s)" and choose a Service or Critical Service (or both) from the box that appears as shown in Figure 36.

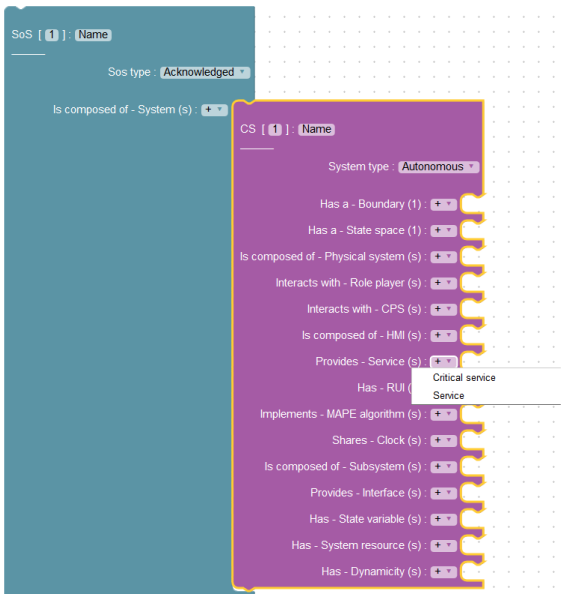

Figure 36 – Adding a Service/Critical Service.

To add a RUMI, the designer can click on “+” related to the item “Has – RUI (s)” of CS block and select the item RUMI from the box that appears.

Double clicking on the RUMI block, the block expands as shown in Figure 37.

In order to provide a defined service through a RUMI the user can click on “+” related to the item “Provides exchange of – Service (S)” of RUMI block and select the service (s) from the box that appears as shown in Figure 37.

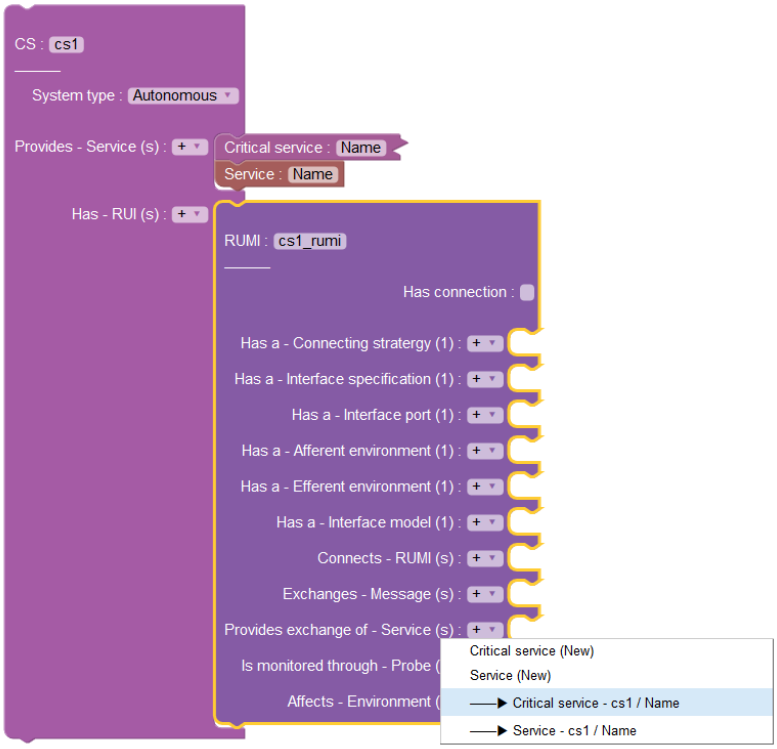

Figure 37 – Providing services through a RUMI.

The result is reported in Figure 38. In this example, the RUMI provides both the services previously defined.

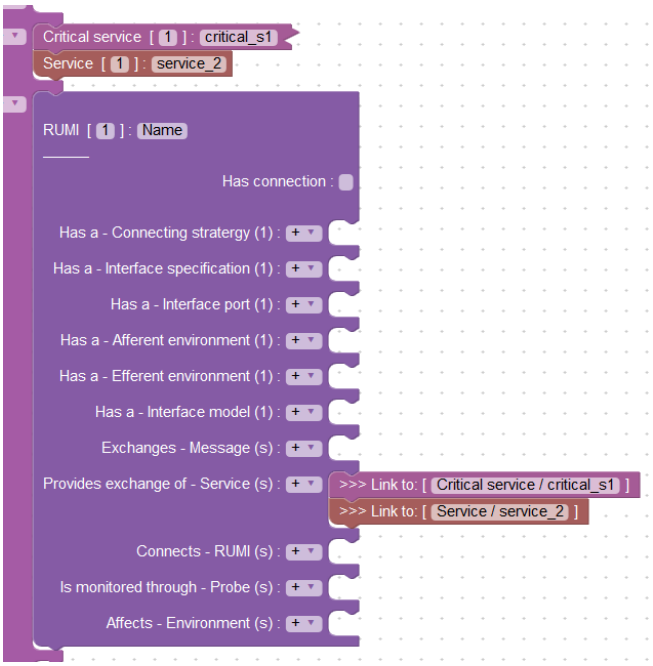

Figure 38 – Services provided by the RUMI.

For simulation a service must have a behavior, else it returns "*Empty service"* by default. To add a behavior, the user can right click on the interested Service block, and select the item "@ Add behavior" as shown in Figure 39.

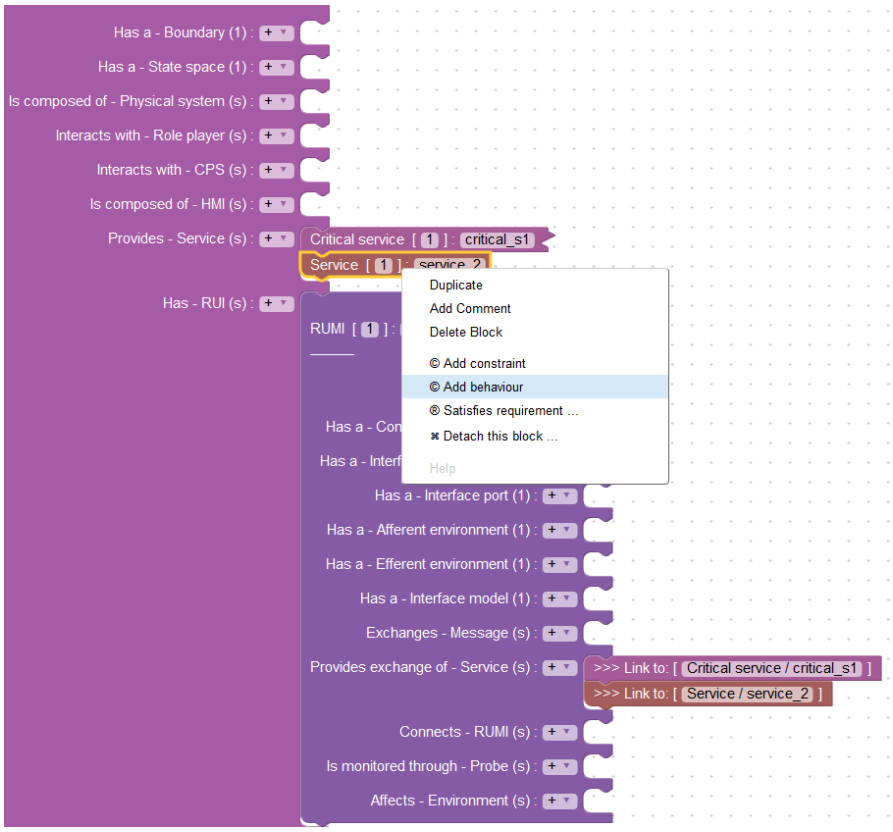

Figure 39 – Adding a Service behaviour.

When the user selects the item “@ Add behavior from the box” the result is depicted in Figure 40. From Figure 40 you can see that it appears an icon button related to the selected service.

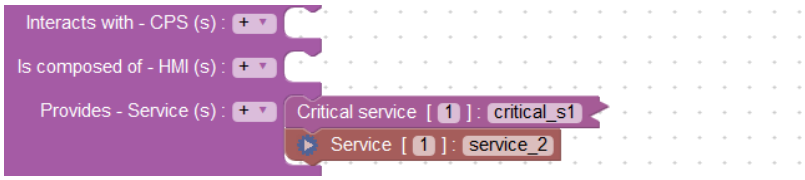

Figure 40 – Service behaviour added.

To define the behavior of the service in Figure 40, the designer can click on the related button icon. Then a text box appears, in which the designer can write python code to define the behavior of the Service - as shown in the example depicted in Figure 41.

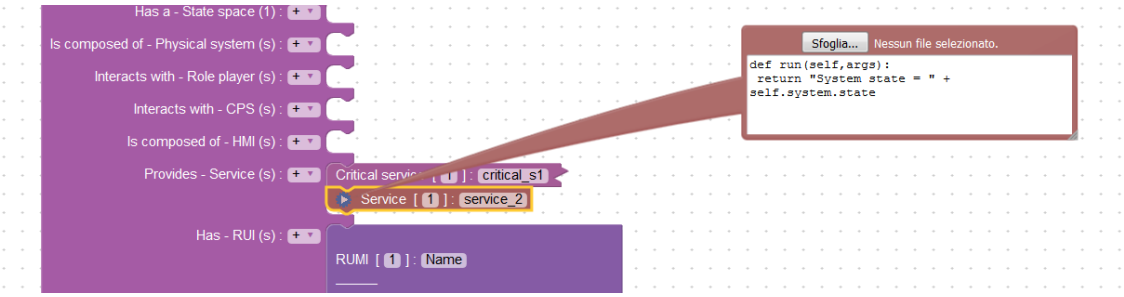

Figure 41 – Service behaviour definition.

Each service is implemented as a separated thread.

2.6 SIMULATION CODE GENERATION

In this section we presented how to perform the simulation code generation of a SoS.

Let us suppose we have created the SoS shown in Figure 42.

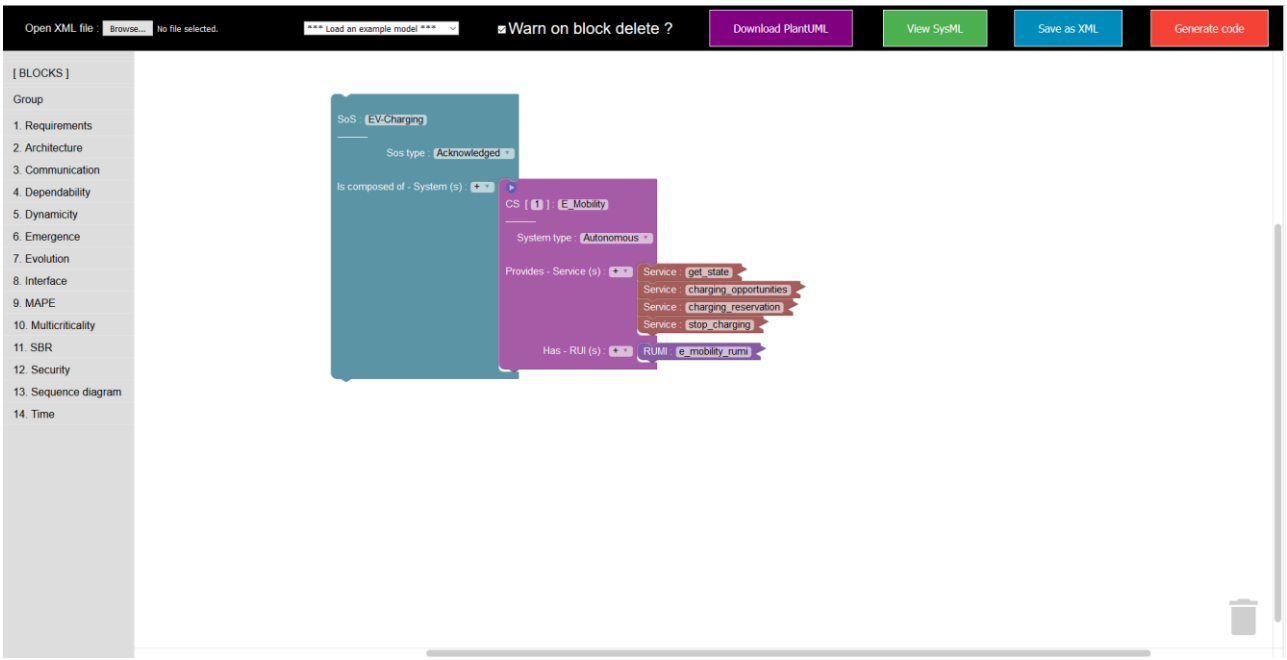

Figure 42 – Example of SoS for code generation.

In order to generate the code related to the SoS shown in Figure 42, the designer must click on the button “Generate code”. The result is shown in Figure 43 : clicking on “Generate code” a .zip file named SoS-Simulation.zip is generated.

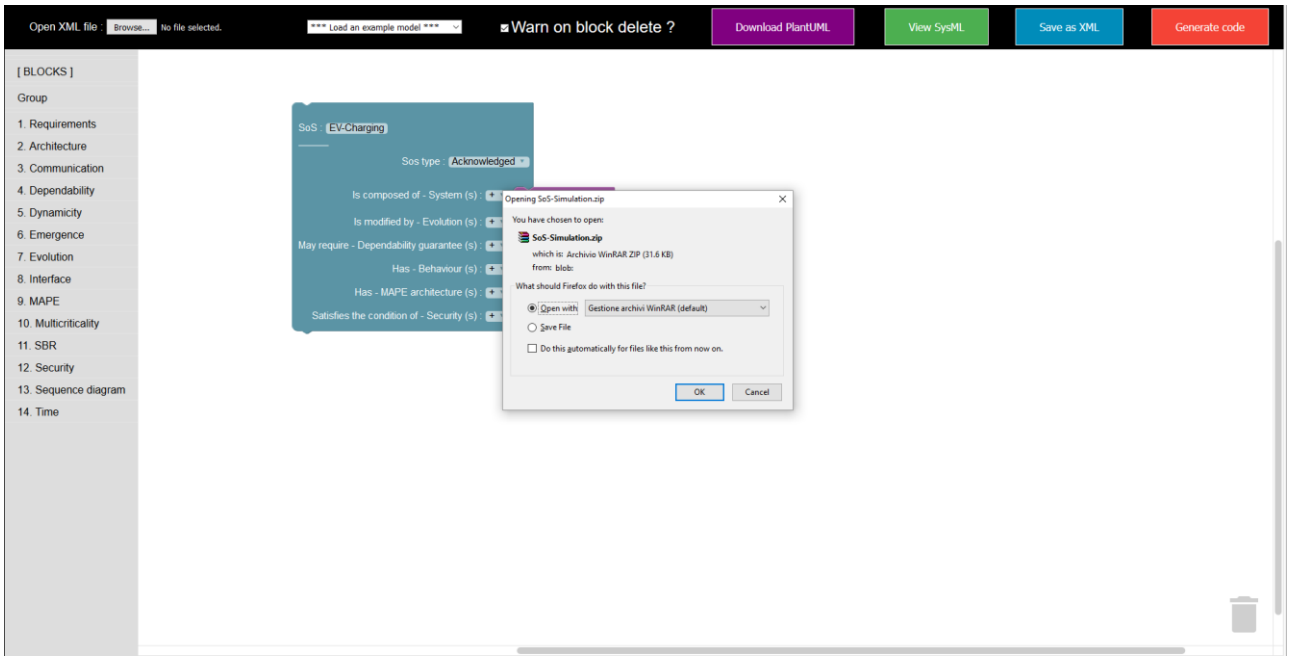

Figure 43 – Coding generation.

As shown in Figure 44, opening the SoS-Simulation.zip, you can see that the archive contains:

- a folder *src* that contains the Python source code
    - amadeos.py (contains all the constructor code)

  - sos.py (the main code for running SoS simulation)
  - sos_gui.py (which interacts with the user)
  - model_behaviour.py (which contains all the behaviors defined by user)
- two files:
  - run-on-unix.sh
  - run-on-linux.bat

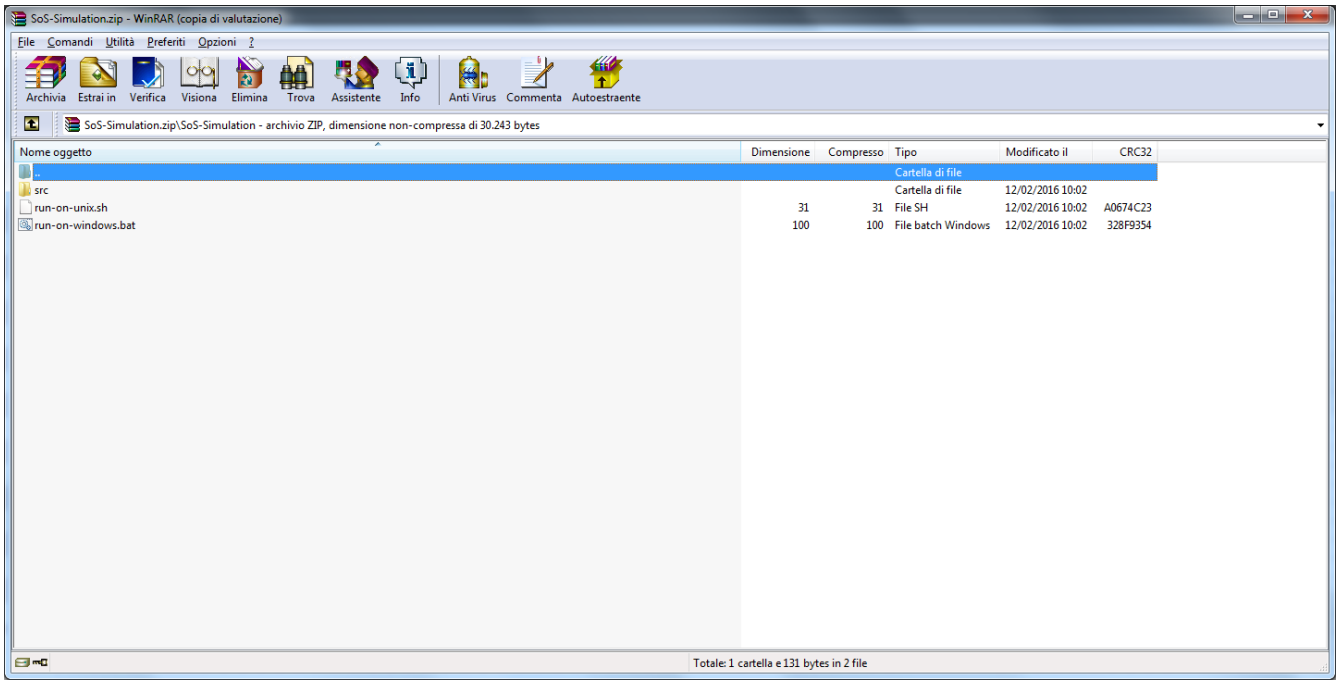

Figure 44 – Opening the SoS-Simulation.zip file.

If the code must be running on Windows, the user has to change the path of Python in the file *run-on-windows.bat* contained in SoS-Simulation.zip file (in the example shown in Figure 45, the path of Python is c:\python27\pythonw.exe).

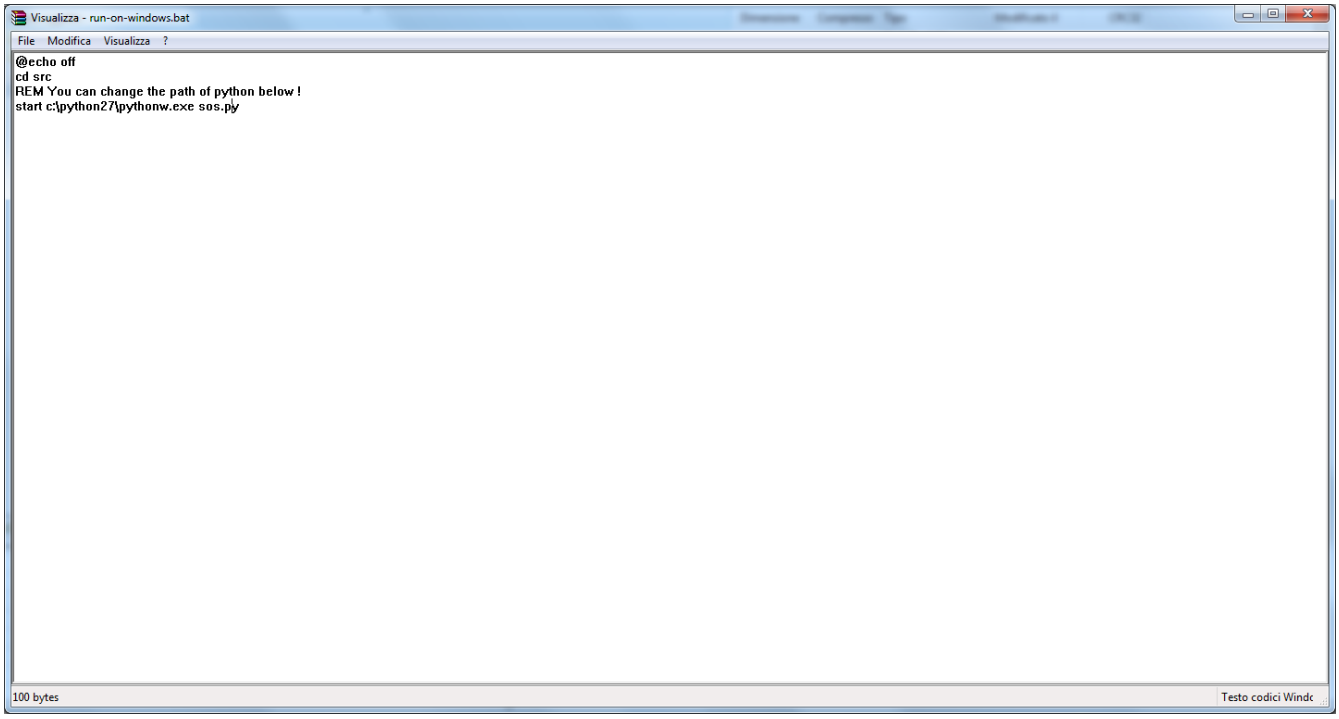

***Figure 45 - run-on-windows.bat* file.**

You can find the latest 2.7 version at https://www.python.org/download/releases/2.7/

3 SIMULATION

This chapter describes the steps to perform a simulation of a SoS (shown in Figure 42).

After generating the code as described in §2.6, extract the folder *SoS-Simulation* contained in *SoS-Simulation.zip* file, modify the *run-on-windows.bat* in the folder if is necessary (see Figure 45).

If the simulation is to be executed on Windows10 (in this example Window Italian version), right click on the file *run-on-windows.bat* and select “Properties”, as shown in Figure 46.

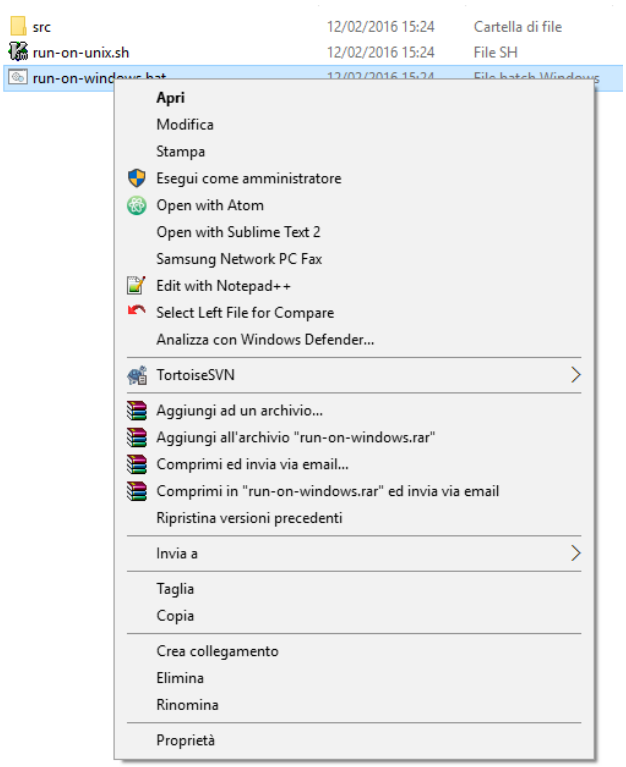

Figure 46 – Access to the properties of run-on-windows.bat file.

Then unblock the bat file executions checking the item “Annulla Blocco/ Unblock” as shown in Figure 47 and click “Applica/Apply” and “Ok”.

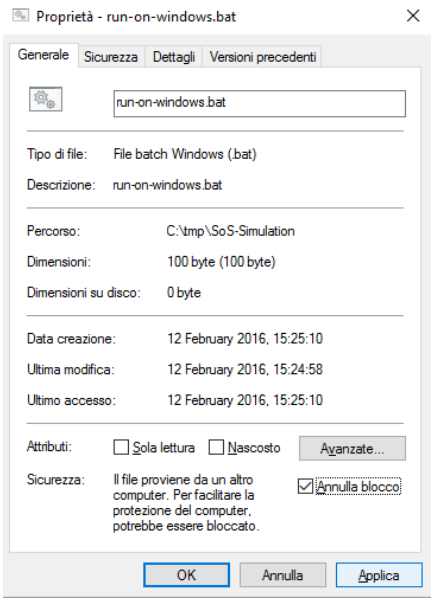

Figure 47 – Unblock the execution of a bat file.

Then double click on *run-on-windows.bat* as shown in Figure 48.

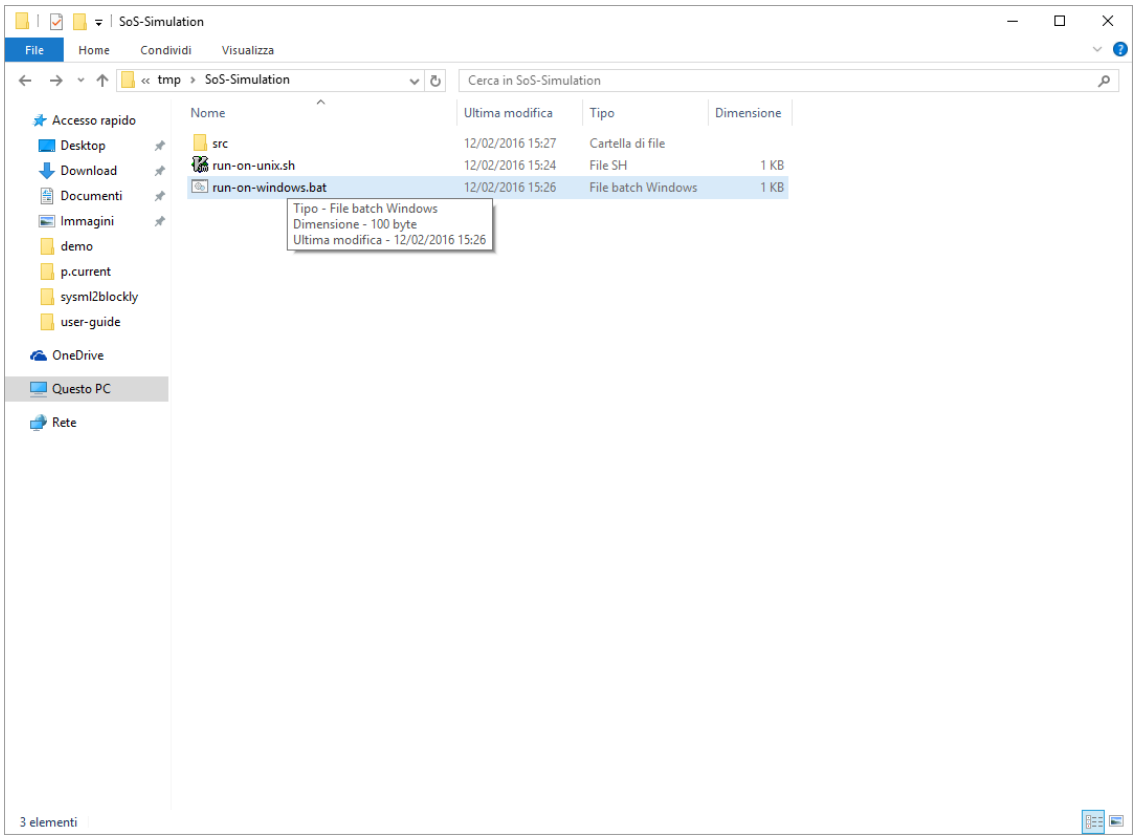

***Figure 48* – Execute the *run-on-windows.bat* file.**

After double clicking on the .bat file, the SoS simulator will be shown as depicted in Figure 49.

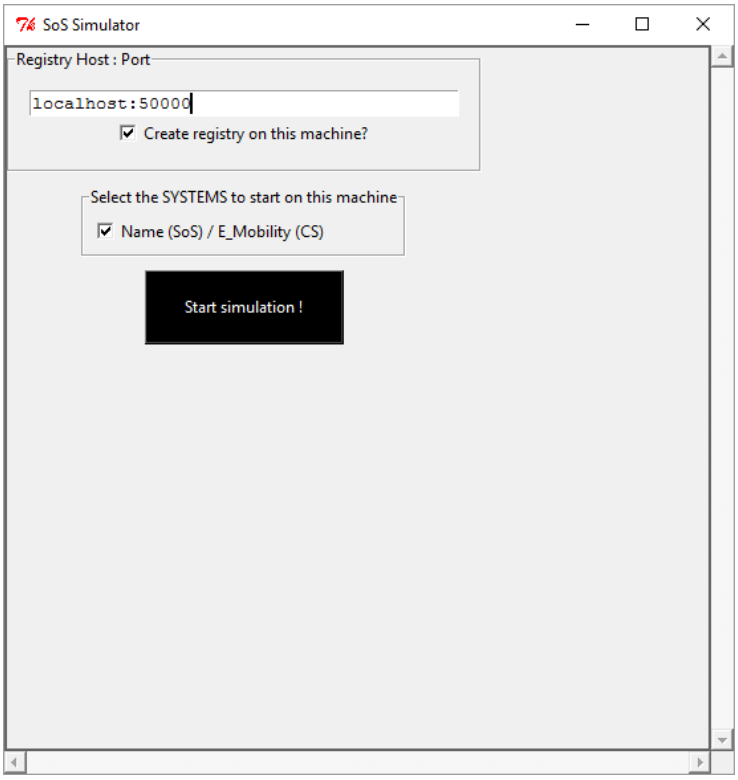

Figure 49 – SoS simulator.

Click on the button “Start simulation!” will show the window shown in Figure 50.

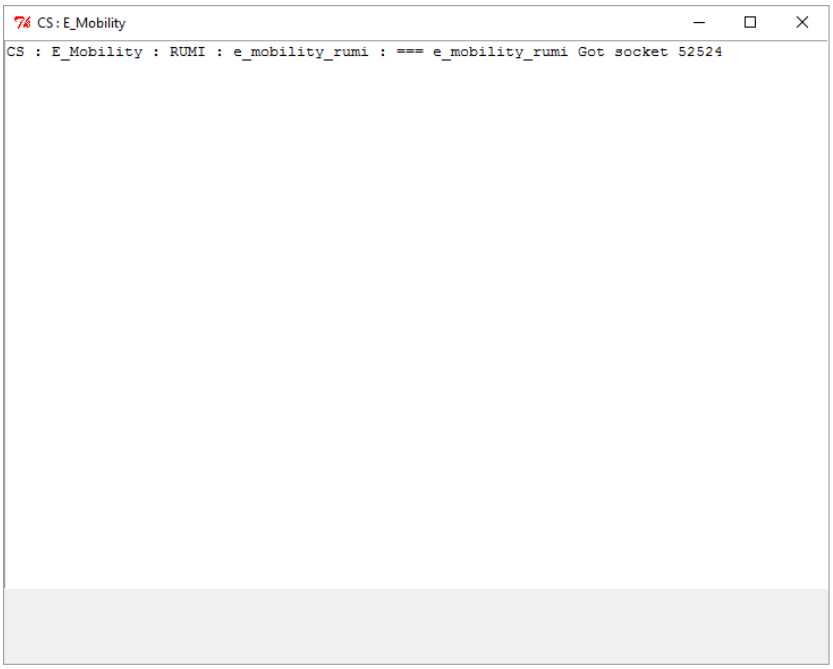

Figure 50 – Intital log of simulation.

In Figure 50 you can see that CS E_Mobility has an active RUMI and can accept connections on Socket number 52524.

In order to request a service from the E_Mobility, the user can open a client (e.g.*putty*) and set Host Name=127.0.0.1, Port= 52524 (the port number must be the same of Socket number), and connection type Raw, as shown in Figure 51.

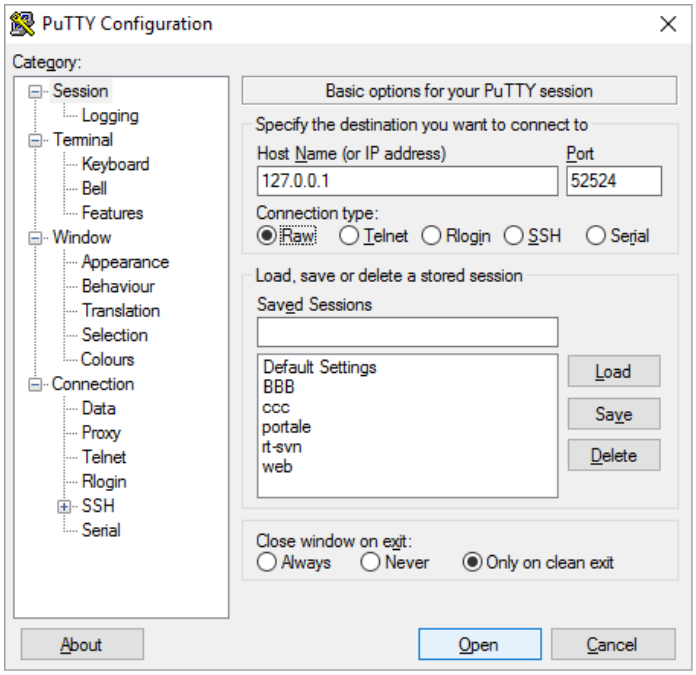

Figure 51 – Putty configuration.

Clicking Open will appear the windows as shown in Figure 52

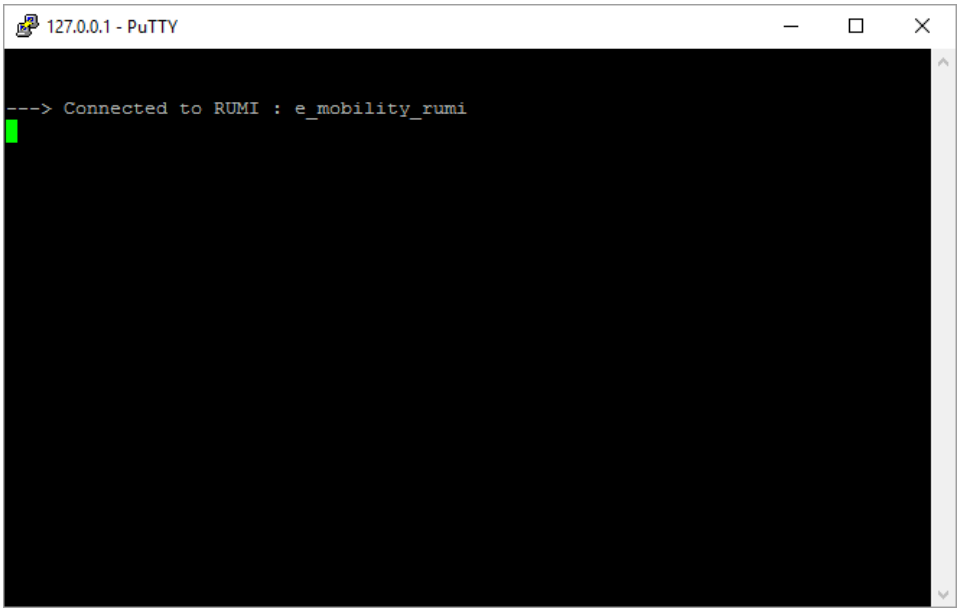

Figure 52 – Client connected to the RUMI of CS E_Mobility.

Figure 52 shows that the client is connected to the RUMI named *e_mobiliy_rumi*.

The user now can request a service provided by the RUMI typing on the windows depicted in Figure 52 the name of the service (e.g., get_status) as shown in Figure 53.

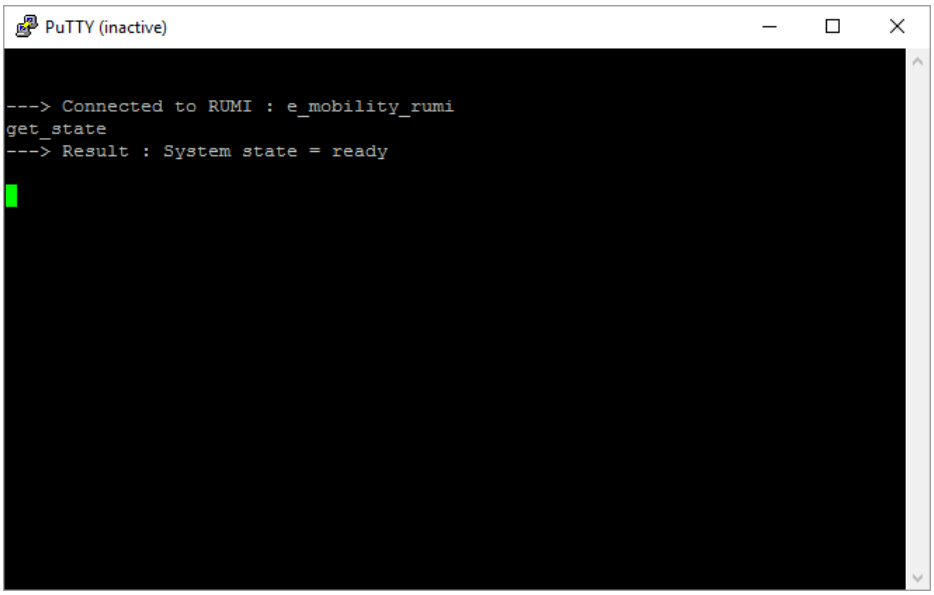

Figure 53 – Client request and server response.

Figure 53 shows the service requested from the client (in this example get_state) and the response from the E_mobility CS.

If the user wants to request another service a putty session must be restarted because the E_mobility closes the socket for the first client request and open another socket for further requests.

Figure 54 shows an example of simulation log, from which quality metrics may be computed.

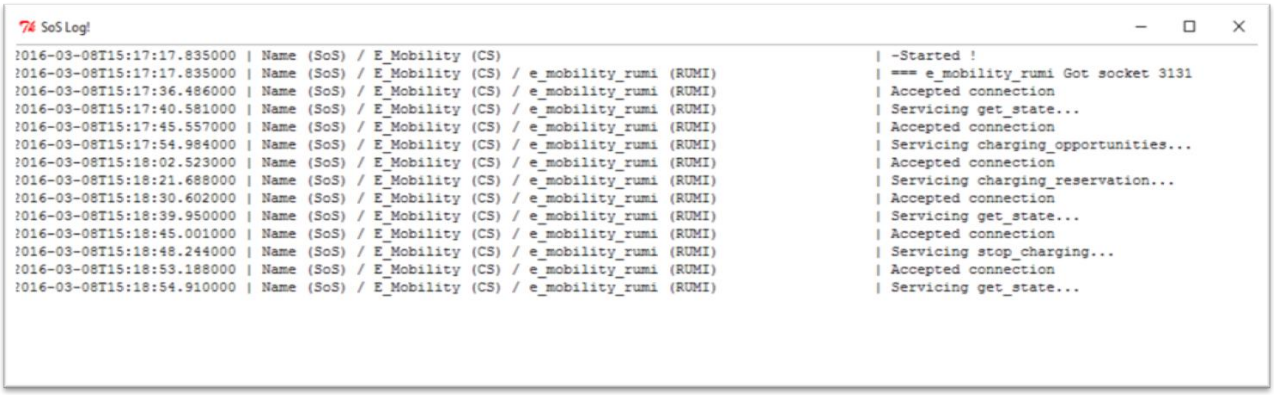

Figure 54 – Example log.

4 BLOCKLY TO PLANTUML

The model created by the user can be converted to PlantUML by clicking on Download PlantUML or view PlantUML. The example presented below shows the sample model in Blockly and the corresponding model in PlantUML.

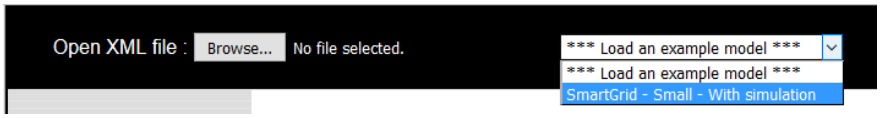

Figure 55 – Loading the sample model in the supporting facility tool

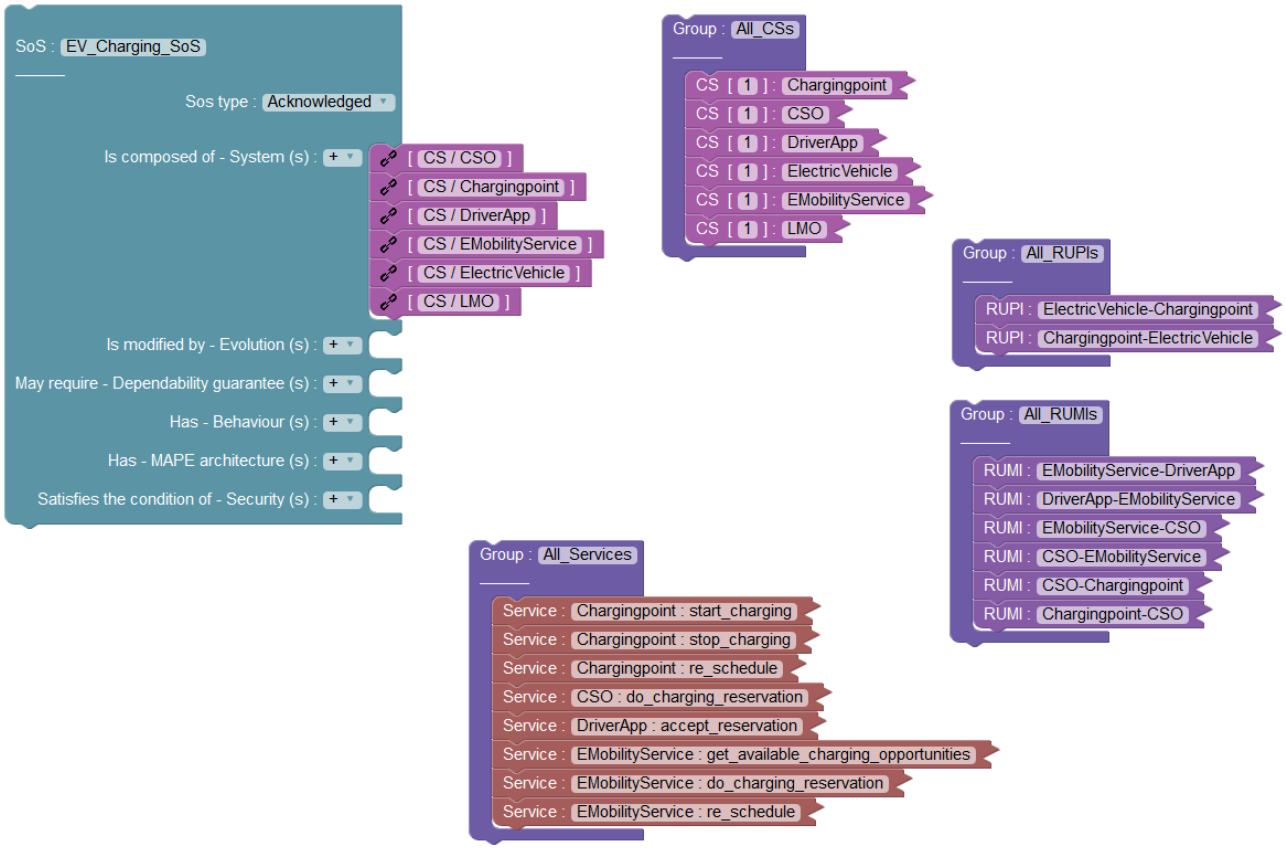

Figure 56 - Sample model after loading

The view PlantUML generates several diagrams, such as: full model without viewpoints, architecture viewpoint, communication viewpoint, interaction between architecture and communication viewpoint diagram.

The full model is large and may require zooming in order to visualize the attributes of blocks. **(Note that some of the diagrams are large and may not be readable, the purpose of these diagrams is just to show that they PlantUML can be generated from Blockly online).**

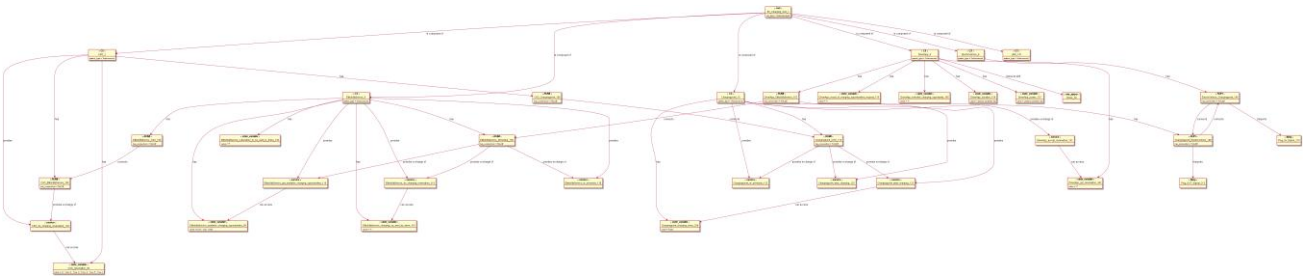

Figure 57 - Full model in PlantUML

Hence for large models it is preferable to view the PlantUML model per viewpoint, as shown below:

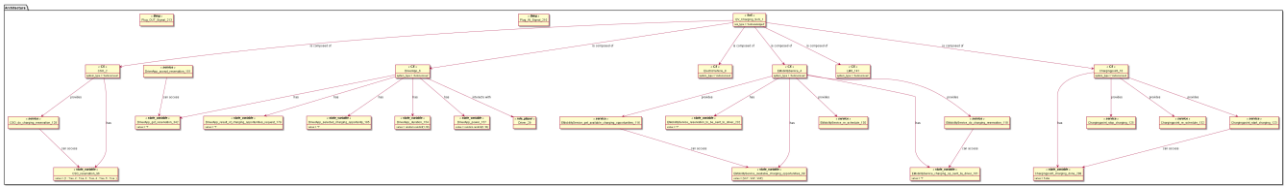

Figure 58 - Architecture viewpoint

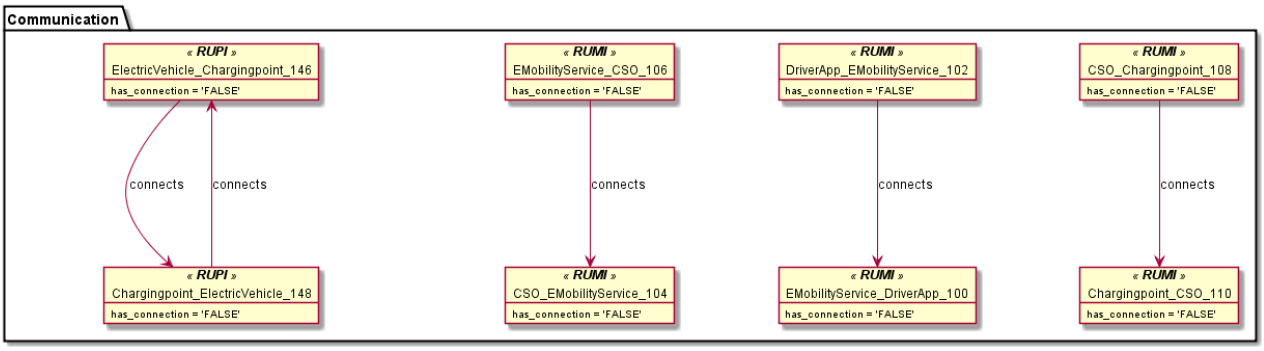

Figure 59 - Communication viewpoint

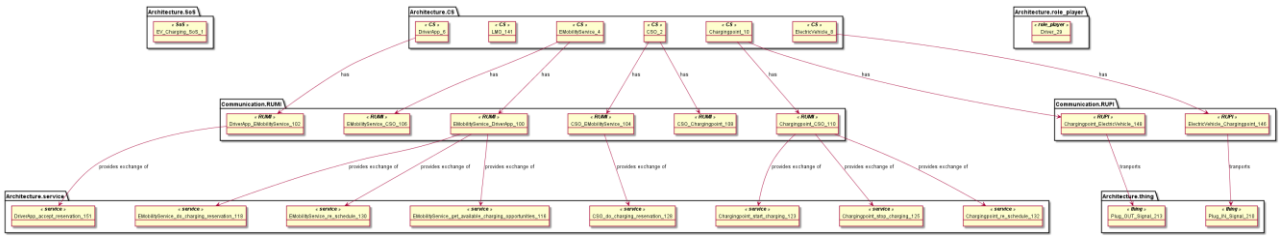

Figure 60 - Interaction between architecture and communication and viewpoint (To simplify the diagram, state-variables have been removed manually, as it does not have any interactions with blocks in communication viewpoint)

5 REFERENCES

[1] Google Blockly Homepage : https://developers.google.com/blockly/
[2] Project AMADEOS. Deliverable D2.2 AMADEOS conceptual model.
[3] Project AMADEOS. Deliverable D2.3 AMADEOS conceptual model - revised.