3

SYSML-UML Like Modeling Environment Based on Google Blockly Customization

Arun Babu Puthuparambil[1], Francesco Brancati[2], Andrea Bondavalli[3,4] and Andrea Ceccarelli[3,4]

[1]Robert Bosch Center for Cyber Physical Systems, Indian Institute of Science, Bangalore, India
[2]Resiltech s.r.l., Pontedera (PI), Italy
[3]Department of Mathematics and Informatics, University of Florence, Florence, Italy
[4]CINI-Consorzio Interuniversitario Nazionale per l'Informatica-University of Florence, Florence, Italy

3.1 Introduction

In industries, it is often observed that system designers may not be CS/OO/SysML experts and often required lot of training and support to use the modeling tools. Ideally, designers should spend all their effort on modeling and nothing else. However, existing modeling tools have lot of issues related to installation and plug-ins.

The use of Google Blockly was envisaged for use of modeling and simulation of systems. Blockly is a visual programming library, used to model/program using interlocked blocks (Figure 3.1). Each of the blocks also support traditional input widgets such as labels, images, textbox, checkbox, combo box, etc. It can be configured in such a way that only compatible blocks can be connected together (i.e. can be made "valid by design"). Blockly supports code and XML generation, and requires only a modern web browser which can be run on any device or operating system.

However, Blockly was not readily useable for modeling SysML/UML like models. A lot of changes and customizations were made in Blockly to make it more suitable for such type of modeling.

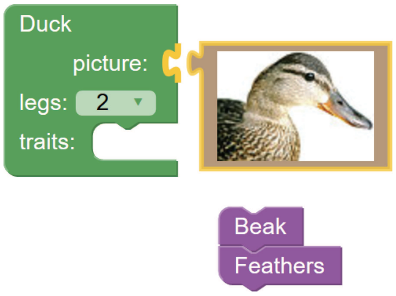


Figure 3.1 Various types of blocks in Blockly.[1]

3.1.1 Goal

To create a tool to create object diagrams based on a UML/SysML profile, which is simple, intuitive, fast, and reduce cognitive complexity. Also, the tool must support rapid modeling and code-generation. On a whole, the goal was to design a tool to model, validate, query, and support simulation.

3.1.2 Blockly Customization

Below is the list of customization performed on Blockly to make it more suitable to create SysML/UML like models. (i) support constraints; (ii) support behaviors; (iii) support links; (iv) support viewpoints; (v) support intuitive maximize, collapse, and semi-collapse; (vi) support requirements management; (vii) guide user to select compatible blocks; (viii) Blockly to PlantUML conversion; (ix) Blockly to Python code generation; (x) Blockly to graph conversion and graph querying; (xi) support sequence diagrams; (xii) custom minification of JavaScript for faster loading; and (xiii) support cardinality and singleton blocks.

3.1.3 Model Transformation

A SysML/UML profile can be given as an input to the tool, which will be converted to an intermediately format in PlantUML. As PlantUML is a simple textual language,[2] conversion to PlantUML makes it easier to debug model transformation. This also makes possible to edit and add any extra features/statements in the converted PlantUML by hand/tool if the earlier format did not support certain features.

[1]https://blockly-games.appspot.com/puzzle
[2]http://plantuml.com/PlantUML_Language_Reference_Guide.pdf

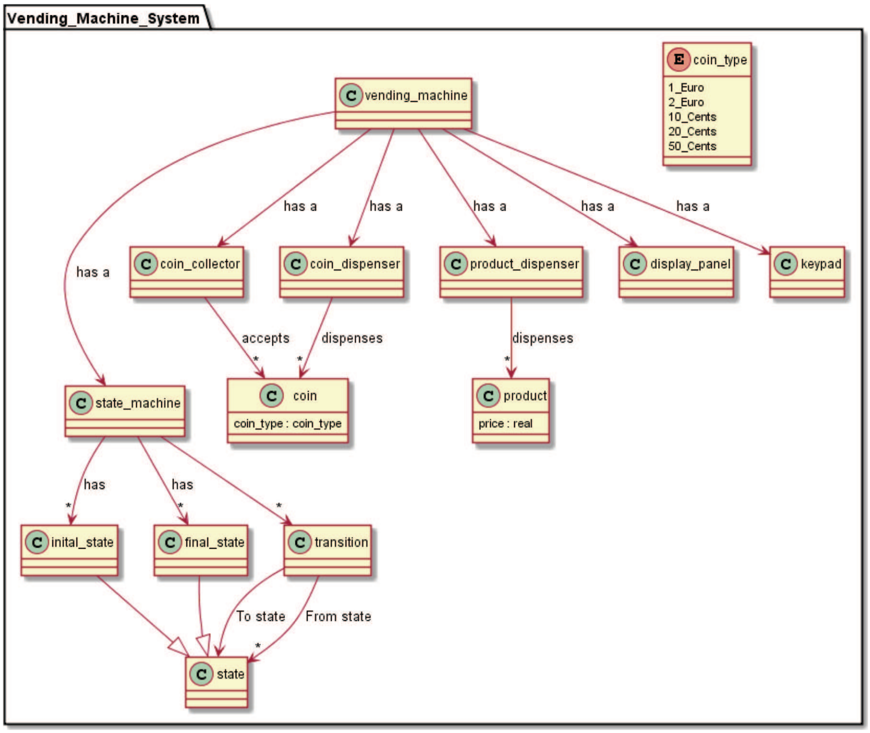


Figure 3.2 An example of a vending machine profile in PlantUML.

Figures 3.2 and 3.3 show a simple example of a vending machine system. The profile in PlantUML is given as input and the tool transforms it into interconnectable blocks.

3.1.4 Requirements Management

In the tool, each block can satisfy a set of requirements and a requirement can be satisfied by a set of blocks (Figure 3.4).

3.1.5 MDE Flow

The model-driven engineering (MDE) flow with the tool is shown in Figure 3.5. First, a profile expert provides a domain specific profile in SysML/UML. This profile restricts what a designer can design and which blocks are compatible with each other. The profile is then converted automatically to PlantUML and is imported into the Blockly format.

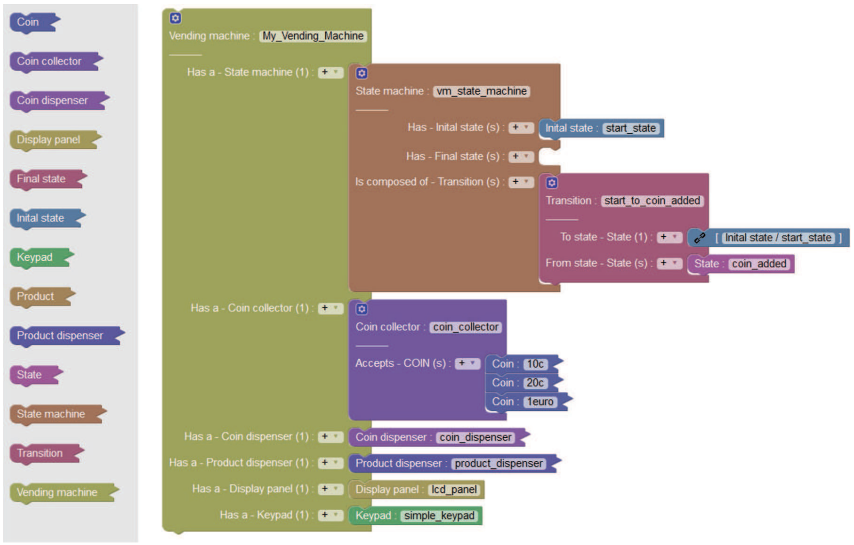

Coin
Coin collector
Coin dispenser
Display panel
Final state
Inital state
Keypad
Product
Product dispenser
State
State machine
Transition
Vending machine
Vending machine : My_Vending_Machine
Has a - State machine (1) :
State machine : vm_state_machine
Has - Inital state (s) :
Inital state : start_state
Has - Final state (s) :
Is composed of - Transition (s) :
Transition : start_to_coin_added
To state - State (1) :
[Inital state / start_state]
From state - State (s) :
State : coin_added
Has a - Coin collector (1) :
Coin collector : coin_collector
Accepts - COIN (s) :
Coin : 10c
Coin : 20c
Coin : 1euro
Has a - Coin dispenser (1) :
Coin dispenser : coin_dispenser
Has a - Product dispenser (1) :
Product dispenser : product_dispenser
Has a - Display panel (1) :
Display panel : lcd_panel
Has a - Keypad (1) :
Keypad : simple_keypad

Figure 3.3 An exampleof a vending machine model under construction.

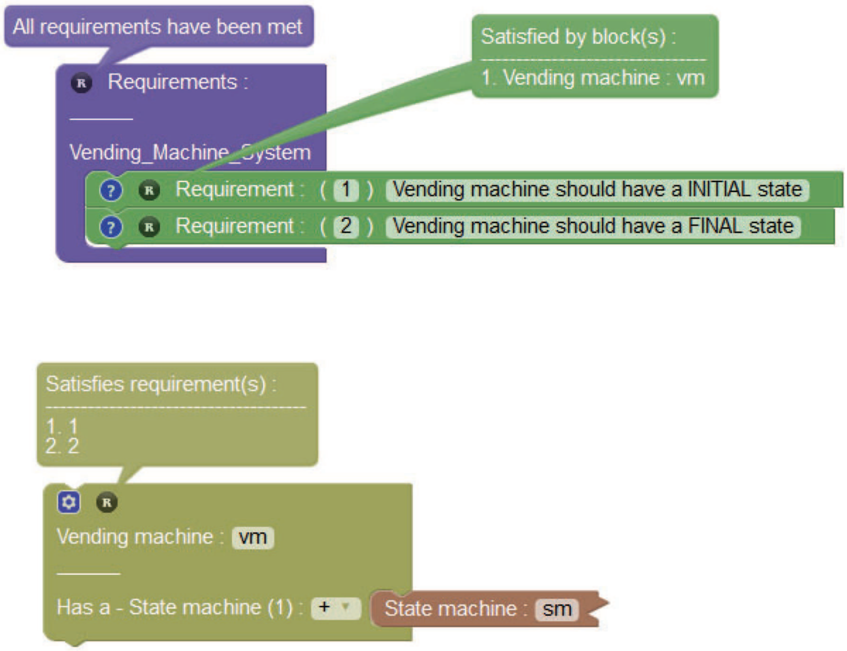

All requirements have been met
Requirements :
Vending_Machine_System
Requirement : (1) Vending machine should have a INITIAL state
Requirement : (2) Vending machine should have a FINAL state
Satisfied by block(s) :
1. Vending machine : vm
Satisfies requirement(s) :
1. 1
2. 2
Vending machine : vm
Has a - State machine (1) :
State machine : sm

Figure 3.4 An example of requirements management.

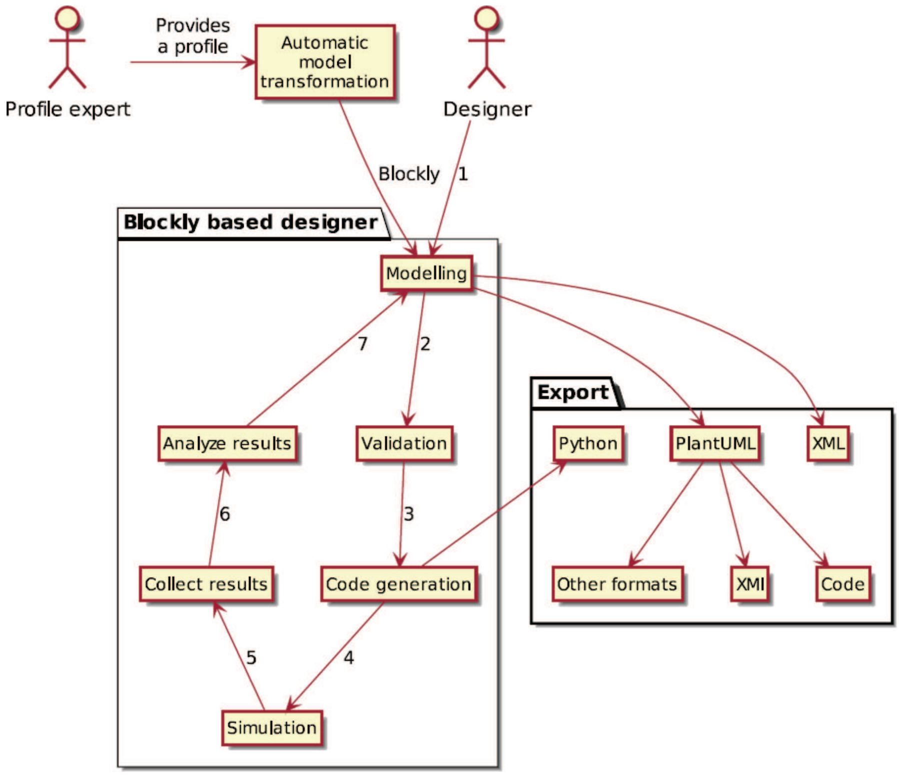


Figure 3.5 MDE flow.

The designer now uses the tool to model and validate the design. After validation code can be generated in Python programming language, which can be used for simulation/testing. After testing the results can be analyzed and changes can be made in the model if necessary. This cycle continues till the model is refined as necessary.

3.1.6 Guiding and Warning Users

The tool guides the designer in two ways: (i) Suggestions for the list of compatible blocks (Figure 3.6); and (ii) Using the type-Indicator plugin[3] (Figure 3.7).

[3]https://github.com/SPE-Systemhaus/blockly-type-indicator/wiki/Type-Indicator

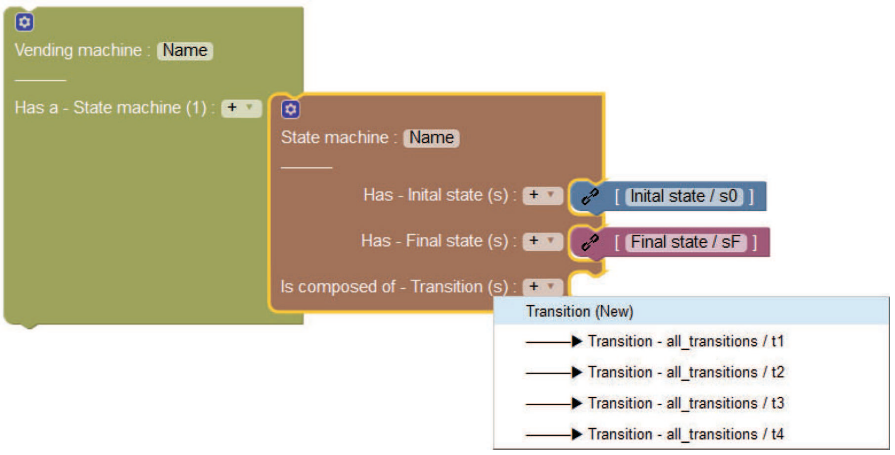

Vending machine : Name
Has a - State machine (1) : +
State machine : Name
Has - Inital state (s) : +
[Inital state / s0]
Has - Final state (s) : +
[Final state / sF]
Is composed of - Transition (s) : +
Transition (New)
Transition - all_transitions / t1
Transition - all_transitions / t2
Transition - all_transitions / t3
Transition - all_transitions / t4

Figure 3.6 An Example of guiding users with compatible blocks (for Transitions).

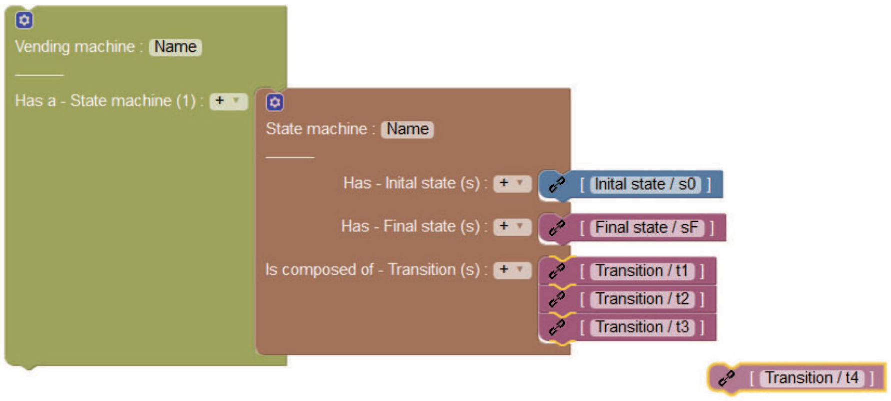
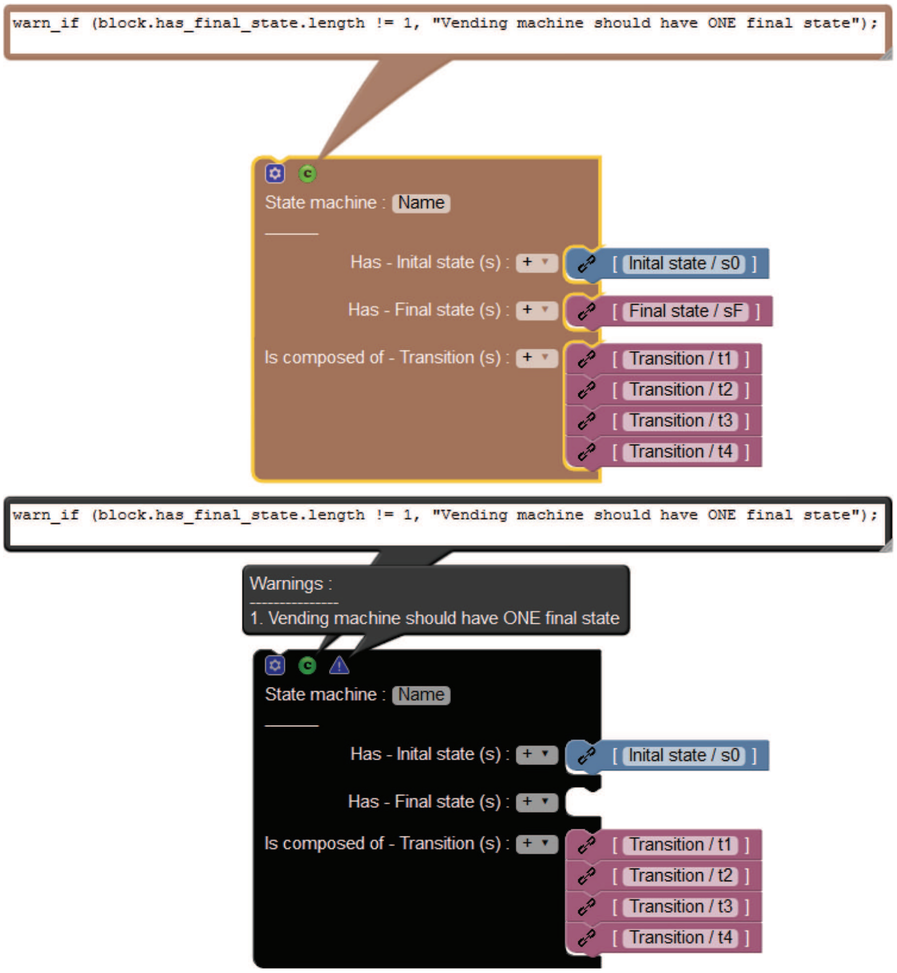


Figure 3.7 An example of type indicator plugin (Shows which blocks are compatible with the current selected block "Transition/t4" with yellow color).

Constraints make a model more precise, hence design time constraints are supported to warn designers when they make mistakes. These constraints are written in JavaScript and are evaluated at every *on change* event of a block (Figure 3.8).



```
warn_if (block.has_final_state.length != 1, "Vending machine should have ONE final state");
```

```
warn_if (block.has_final_state.length != 1, "Vending machine should have ONE final state");
```

```
Warnings :
---------------
1. Vending machine should have ONE final state
```

Figure 3.8 An example of constraints.

3.1.7 Modular Design and Viewpoints

Meaningful groups can be formed to modularize design and links can be used instead of lines to connect two blocks which are away from each other. Use of groups and links avoid the spaghetti diagrams in large models (see Figure 3.9).

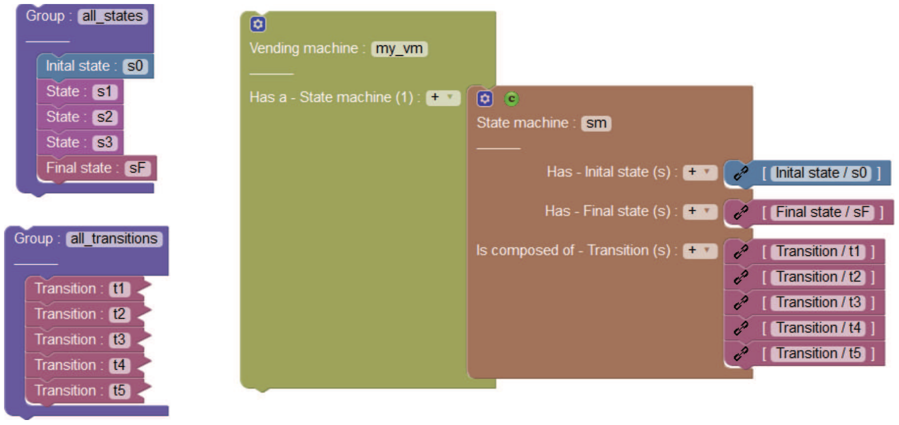


Figure 3.9 An example of groups and links.

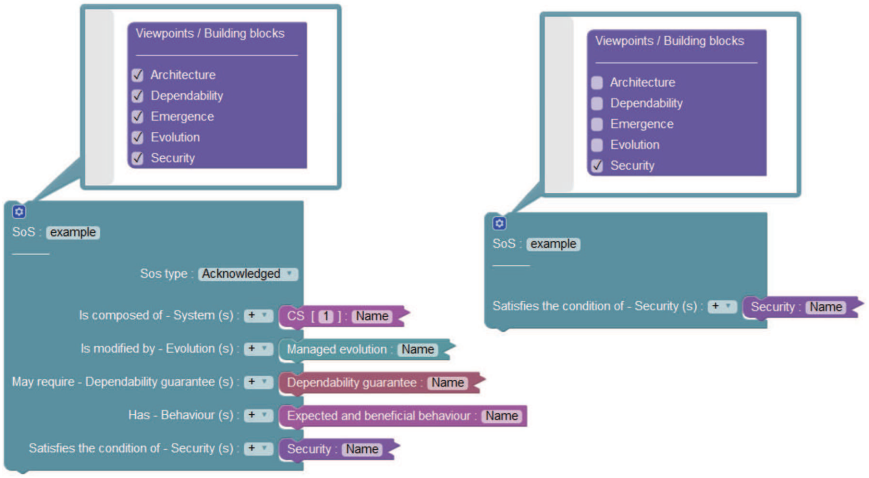

Viewpoints / Building blocks
✓ Architecture
✓ Dependability
✓ Emergence
✓ Evolution
✓ Security
SoS : example
Sos type : Acknowledged
Is composed of - System (s) :
CS [1] : Name
Is modified by - Evolution (s) :
Managed evolution : Name
May require - Dependability guarantee (s) :
Dependability guarantee : Name
Has - Behaviour (s) :
Expected and beneficial behaviour : Name
Satisfies the condition of - Security (s) :
Security : Name
Viewpoints / Building blocks
Architecture
Dependability
Emergence
Evolution
✓ Security
SoS : example
Satisfies the condition of - Security (s) :
Security : Name

Figure 3.10 Enabling and disabling viewpoints in model.

Viewpoints are used in profile to reduce cognitive complexity for the designers. Viewpoints allow users to focus on one aspect of the model, e.g.: Architecture/Communication etc. Usually viewpoints do not exist in isolation; various viewpoints have relationships between each other. Figure 3.10 is an example of viewpoints in the tool; the viewpoints can be enabled/disabled.

3.1.8 Model Querying

On large models, it is important to query for blocks satisfying certain conditions. Thus, support for model querying in JavaScript was provided in the tool. The user provides a filter function, which is checked with all blocks. If the condition is satisfied, then it is highlighted, else it is not. Figure 3.11 is an example of the query “return true;” query, i.e., does not apply any filter and show all blocks. In Figure 3.12, instead, a filter is applied: it selects all blocks of type “RUMI” (return block.of_type == ‘RUMI’).

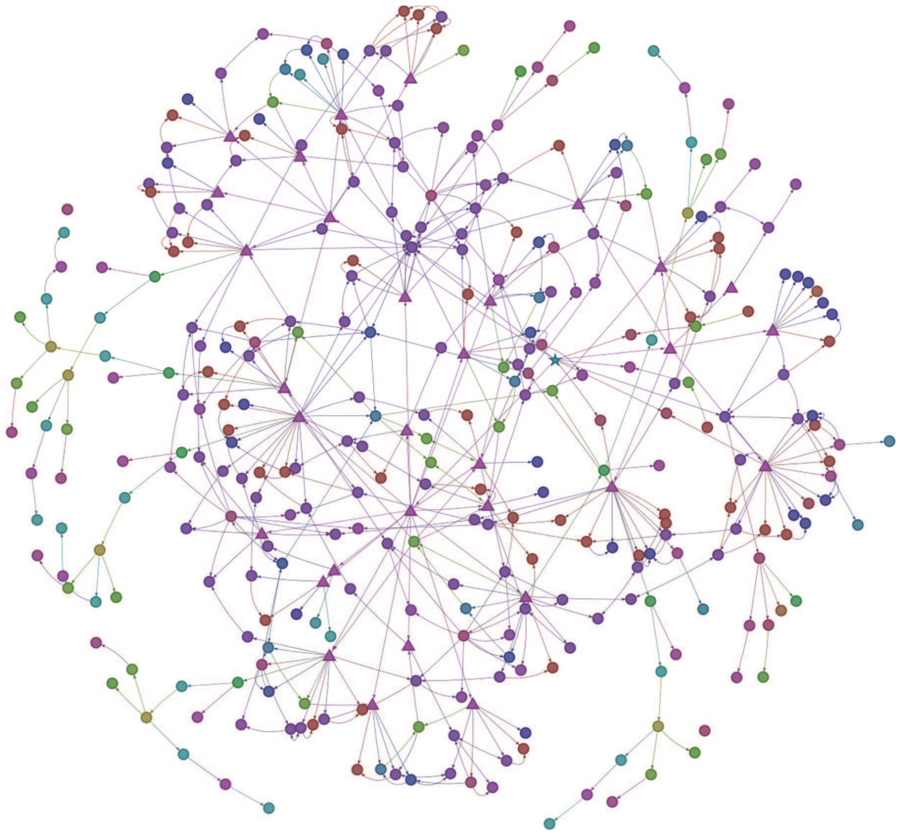


Figure 3.11 Model query without any filter (return true;).

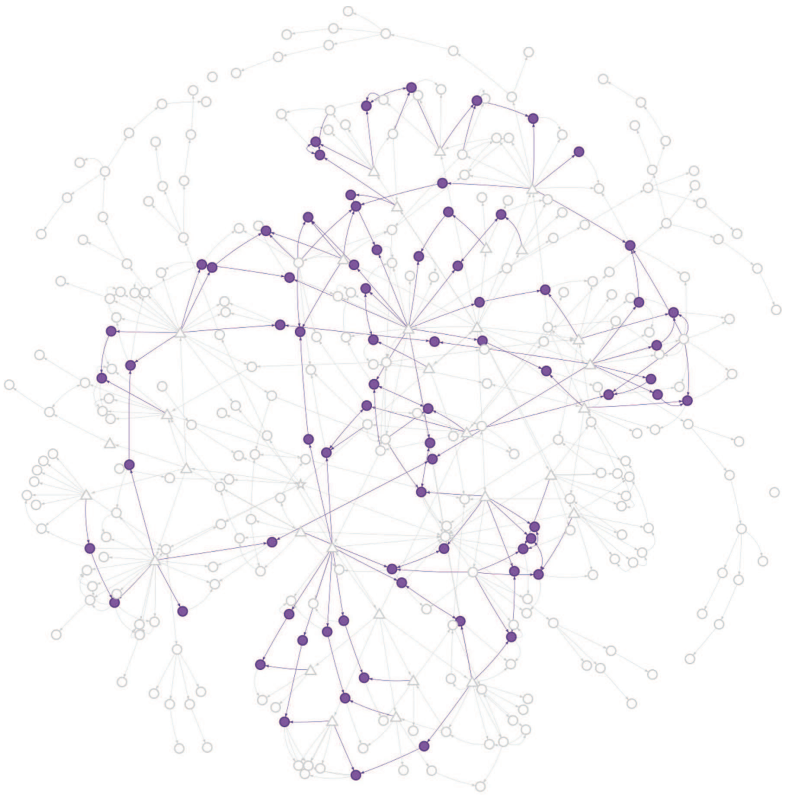


Figure 3.12 Example of model query to select all blocks of type “RUMI” (return block.of_type == ‘RUMI’).

3.1.9 Code Generation and Export to PlantUML

From the model, code can be generated to Python automatically. Python was chosen as it is one of the simplest object oriented programming language. However, other programming languages can easily be supported in Blockly.[4]

Also, as the model is available in .xml format and PlantUML format, custom code[5] and other programming languages can be supported in future.

Blockly models can be exported to PlantUML (Figure 3.13). The PlantUML version of model consist of two type of diagrams (i) the whole

[4]https://developers.google.com/blockly/guides/configure/web/code-generators

[5]https://developers.google.com/blockly/guides/create-custom-blocks/generating-code

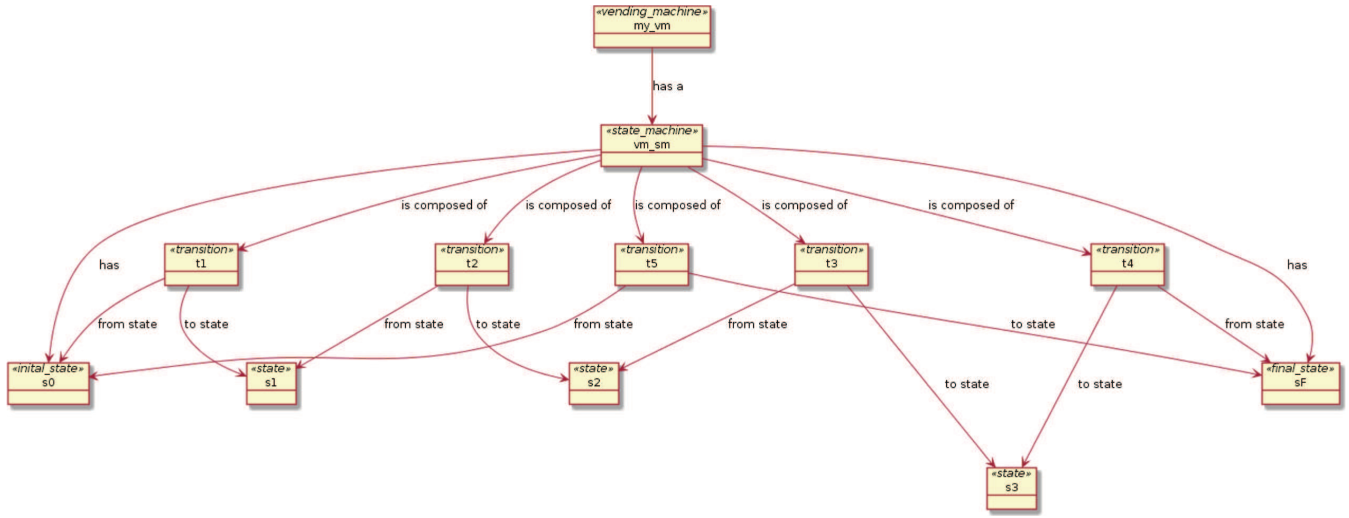


Figure 3.13 The subset of example model of “Vending machine” exported to PlantUML.

model without viewpoints; and (ii) model divided into separate files with grouped as viewpoints. These PlantUML models can be used for further refinement or be used with other tools.[6]

3.1.10 Simulation

Simulation of scenarios is supported using sequence diagrams and simulation related blocks custom code in Python. Domain specific sequence diagrams blocks are supportedto make design easier and error-free. As opposed to traditional generic sequence diagrams, these domain specific blocks are non-ambiguous and it allows correct code generation. Figure 3.14 shows an example of a sequence diagram containing domain specific blocks. Each sequence diagram can consist of sub-sequence, which in turn can consist of simple blocks such as: if, while, parallel, etc., and may also contain custom domain specific blocks.

The sequence diagram drawn using blocks can also be automatically converted to classical sequence diagram view as shown in Figure 3.15.

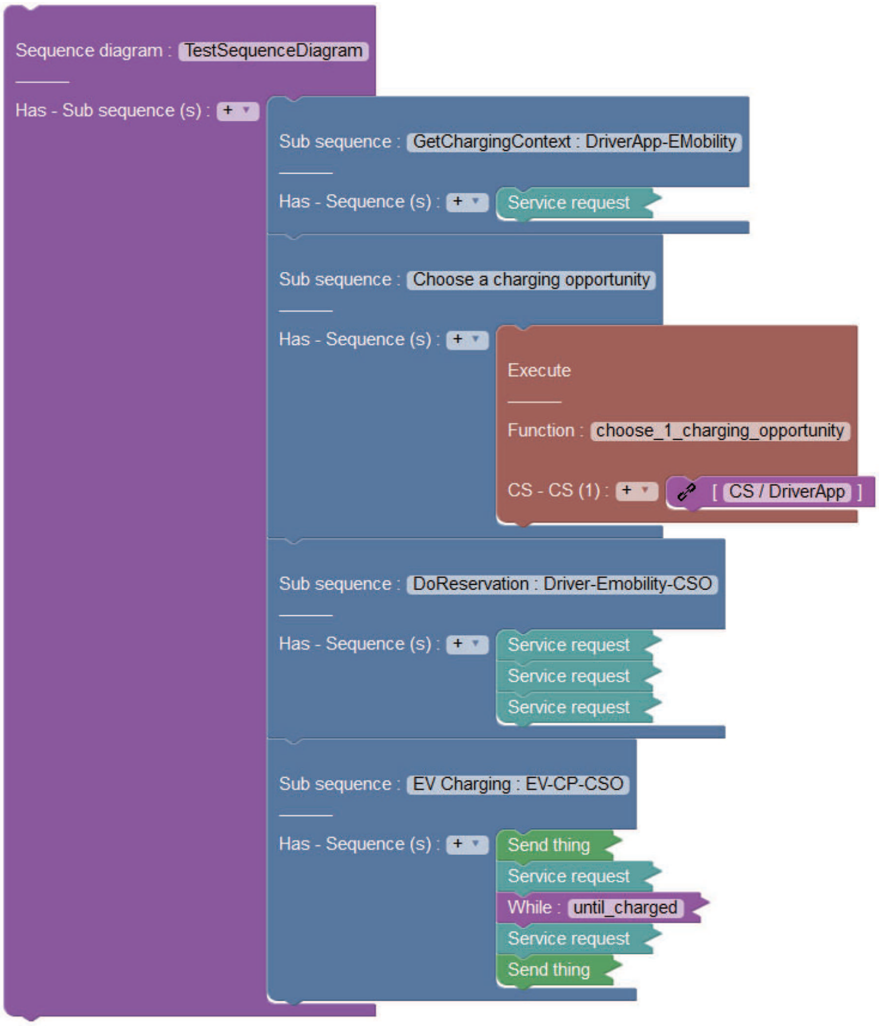
Custom code to be run before starting and after ending simulation can also be added using the simulation related blocks (Figure 3.16). These blocks can be used in initializing variables before simulation, pre-processing of data before simulation, and post-processing of results after simulation.

3.1.11 Conclusion and Future Work

This chapter has introduced an intuitive and simple semi-formal tool to be used to model, validate, query and simulate systems based on a SysML/UML profile. There is always scope to improve upon the approaches proposed in the chapter especially to make semi-formal methods popular among non-experts.

Some of the possible future research areas are: (i) Blocks with images or blocks shaped as images could make a great feature to make the design more intuitive (Figure 3.17); (ii) Currently, a transformation from Eclipse/Papyrus to PlantUML is available and can be readily used by the tool, however many more transformations can be written to PlantUML; and (iii) More programming languages support could be added in future.

[6]http://plantuml.com/running



```
Sequence diagram : TestSequenceDiagram
Has - Sub sequence (s) :
    Sub sequence : GetChargingContext : DriverApp-EMobility
    Has - Sequence (s) :
        Service request
    Sub sequence : Choose a charging opportunity
    Has - Sequence (s) :
        Execute
        Function : choose_1_charging_opportunity
        CS - CS (1) :
            [ CS / DriverApp ]
    Sub sequence : DoReservation : Driver-Emobility-CSO
    Has - Sequence (s) :
        Service request
        Service request
        Service request
    Sub sequence : EV Charging : EV-CP-CSO
    Has - Sequence (s) :
        Send thing
        Service request
        While : until_charged
        Service request
        Send thing
```

Figure 3.14 Example sequence diagram in Blockly.

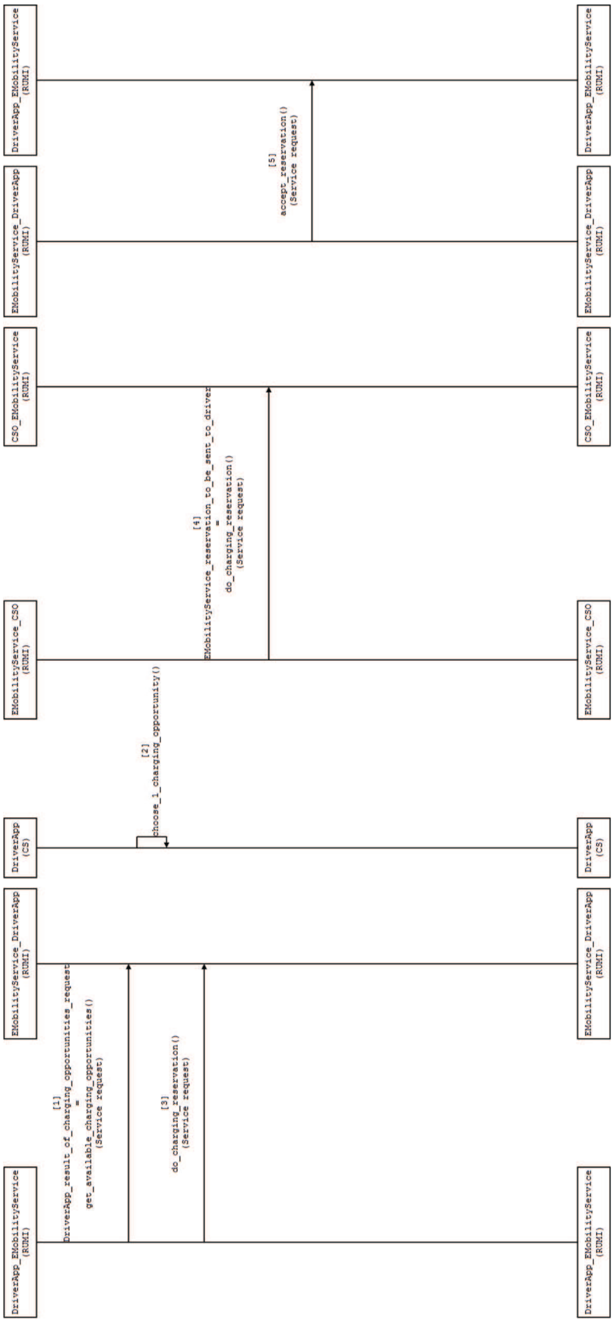
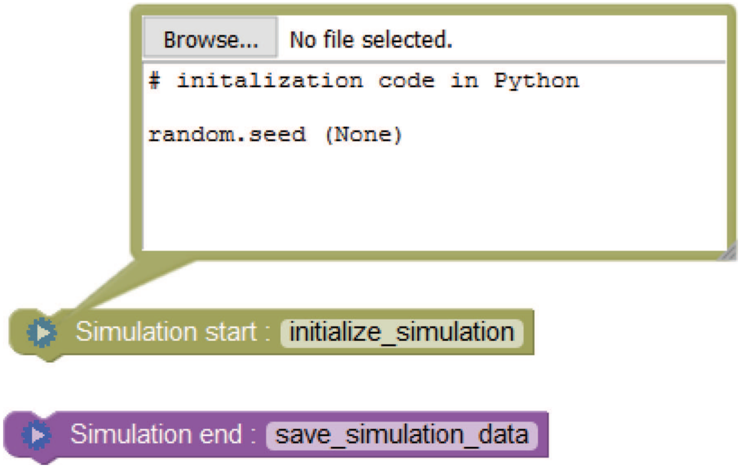

DriverApp_EMobilityService (RUMI)
EMobilityService_DriverApp (RUMI)
DriverApp (CS)
EMobilityService_CSO (RUMI)
CSO_EMobilityService (RUMI)
EMobilityService_DriverApp (RUMI)
DriverApp_EMobilityService (RUMI)
[1] DriverApp_result_of_charging_opportunities_request = get_available_charging_opportunities() (Service request)
[2] choose_1_charging_opportunity()
[3] do_charging_reservation() (Service request)
[4] EMobilityService_reservation_to_be_sent_to_driver = do_charging_reservation() (Service request)
[5] accept_reservation() (Service request)

Figure 3.15 Classical view of sequence diagram (subset).


Browse...
No file selected.
Simulation start : initialize_simulation
Simulation end : save_simulation_data

```
# initalization code in Python

random.seed (None)
```

Figure 3.16 Blocks to support custom simulation initialization and code to execute when simulation ends.

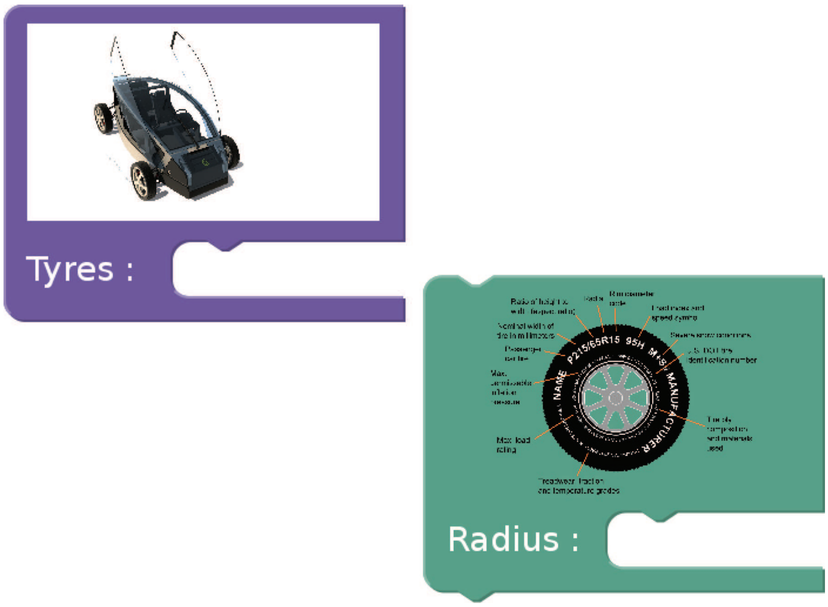


Figure 3.17 Blocks with images.