CSE – 621
LEJOS PROJECT – PHASE II
Java Code Generation from Statechart Diagram
Instructor: Dr. Eric Rapos

Bhisma Adhikari, Nick DiGennaro

March 2020

1 Description of Source and Target Meta-Models

The primary goal of this phase of the project was to generate compilable java code from a statechart diagram. In other words, the task was to transform a given textual representation of a statechart diagram into a valid java file.

1.1 Source Meta-Model

The source file for the transformation is, any valid state-machine/statechart diagram [2] file generated using the StarUML software. State-machine diagrams are UML standard diagrams used for the behavioral modeling of systems. The textual representation of a startchart diagram generated by StarUML is a json [1] file, saved with an extension '.mdj', for example SampleStatechart.mdj. Json is a file format (or a string format) used commonly for data-exchange. In this format, data or objects are stored as key-value pairs, where the key is always a string, while the value can either be a string or a nested json object.

The top level view of the json string of a statechart.mdj file has the following attributes::

- _type
- _id
- name
- ownedElements

Among these attributes the values of all the attributes except 'ownedElements' are of string type. OwnedElements, on the other hand, is a list of one or more OwnedElements, each of which is a nested json object. The following are some typical attributes found in an 'ownedElement':

- _type
- _id
- _parent
- name
- defaultDiagram
- ownedViews

A typical tree structure showing only some major attributes in a stateflow.mdj file is shown below:

- _type
- _id
- name
- ownedElements
  - _type
  - _id
  - _parent
    - $ref
  - name
  - defaultDiagram
  - ownedViews
    - _type
    - _id
    - _parent
    - model
    - subViews
      - _type
      - _id
      - _parent
      - visible
      - font
      - left
      - top
      - height
    - font
    - top

        * width
        * height
        * stereotypeLabel
        * nameLabel
        * namespaceLabel
        * propertyLabel
        * regions

The 'regions' attribute contains list of 'region' objects. The 'region' object is organized as follows:

- regions
    - _type
    - _id
    - _parent
        * $ref
    - vertices
        * _type
        * _id
        * _parent
            · $ref
        * name
    - transitions
        * _type
        * _id
        * _parent
            · $ref
        * source
            · $ref
        * target
            · $ref
        * guard
        * triggers
            · _type
            · _id
            · _parent
            · name

While generating an equivalent java code from the startchart diagram, we are concerned only with the information regarding States (Vertices, in StarUML's terminology) and Transitions. Both of these information are located in the "region" section of the statechart diagram as shown in the json tree structure above. Transitions contain information about triggers and guards, if any.

1.2 Target Meta-Model

The target file for the transformation is a single valid java file (CSE621.java), that can be compiled and run on the LeJOS EV3 hardware. The file must represent the same information as the given input statechart diagram file.

To be considered a valid output, the generated file should be valid java code capable of compiling on the LEJOS EV3 hardware. Therefore, there are actually no restrictions on how the code inside the generated java program is organized. However, for the sake of human-readability, we further constrict the structure of the target output (This is not required, but is nonetheless good to have). The target file shall have the following organization:

- package $< PACKAGE\_NAME >$
- import statements
- class definition of class CSE621
  - This class shall contain the definition of the Main() method.
    - The Main() method shall contain an infinite loop which continuously does the following tasks:
      - Get robot's current state.
      - Check if any of the triggers (relevant to the current state of the robot) occurs, and make the necessary transition.
- class definition of class Robot
  - This class shall model the robot.
  - The constructor shall initialize all actuators and sensors of the robot
  - The class shall define public methods for
    - setting Robot's state. The state can be one of the values defined in enum State (see below). This method should also perform necessary actions which are to be done upon entering that particular state.
    - getting sensor values – color and distance from obstacle.
- enum definition of Color with the following values
  - RED
  - GREEN
  - BLUE
  - YELLOW
  - UNKNOWN
- enum definition of State with the following values
  - IDLE

- FORWARD
- BACKWARD
- ROTATE_LEFT
- ROTATE_RIGHT

2 Transformation Rules

In this section, we will explain how a transformation is made from a statechart file to java source code. To start, the implementation follows an object-oriented approach (rather than a functional approach). This means the transformation rules will not always result in a function/method name. For example, the Trigger constructor initiates a trigger type when instantiating a Trigger object. This means the transformation is handled when a trigger is recognized and initiated instead of adding a robot command. We now enlist each of these transformation rules written either explicitly/implicitly in uml2lejos.java in terms of how each phase of the transformation was achieved. When referring to a transformation rule, we will use the notation class_name.method_name. This does not necessarily mean that the method is a static method.

(a) **Transformation: Statechart file to Vertex**

- This transformation rule is defined in CodeGenerator.setVertices().
- This rule uses jackson library's method ObjectMapper.readValue() to convert a Statechart.mdj file to nested java map objects.
- This rule uses Util.getValueByKey() to extract all vertices from the nested java map objects obtained from the previous step.
- This rule uses the constructor Vertex.vertex() to create Vertex objects from the java map objects obtained from the previous step.

(b) **Transformation: Statechart file to Transition**

- This transformation rule is defined in CodeGenerator.setTransitions().
- This rule uses jackson library's method ObjectMapper.readValue() to convert a Statechart.mdj file to nested java map objects.
- This rule uses Util.getValueByKey() to extract all transitions from the nested java map objects obtained from the previous step.
- This rule uses the constructor Transition.transition() to create Transition objects from the java map objects obtained from the previous step.

(c) **Transformation Rule : CodeGenerator.generateCode()**

- This transformation rule calls the following three transformation rules, appends the returned code snippets from each of the called rules into a single long string and then writes it in the file CSE621.java.

i. **Transformation Rule : CodeGenerator.generateImports()**
   - This rule generates the package and import statements.

ii. **Transformation Rule : CodeGenerator.generateClassCSE621()**
   - This rule generates the class definition for the class CSE621.
   - This rule uses the transitions, vertices and related information made available by the transformation rules CodeGenerator.setVertices() and CodeGenerator.setTransitions().
   - The generated class definition includes the definition of the program's entry point i.e. the "public static void Main(String[] args)" function. This function contains an infinite while loop with contents as specified in the section "Target Meta-Model".
   - This rule uses CodeGenerator.transitionToStmt() to transform Transition objects to equivalent java statements. CodeGenerator.transitionToStmt() checks all relevant triggers and guards while generating their respective code snippets.

iii. **Transformation Rule : CodeGenerator.generateOtherClasses()**
   - This rule generates the class definition of class Robot and enums Color and State as specified in the section "Target Meta-Model".

3 Remarks

1. When the sign "==" appears in distance triggers, the equivalent generated code contains "<" (if robot's current state is FORWARD), and ">" (if robot's current state is BACKWARD) rather than "==". This is done on purpose to enhance reliability of the LeJOS robot.

2. Sections of the code which are required in the generated java file but are not directly generated from the given statechart diagram are hard-coded. This includes the import statements, package statements and the definitions of class Robot, enum Color, and enum State. These sections of code can be found in the text files, code-imports.txt and code-other-classes.txt. The text files are read by uml2lejos.java during the transformation to create the hard-coded sections. Both of these files are included in the zipped folder for submission.

3. The output CSE621.java file is a valid java file that is well-formatted by the use of proper style guidelines. Thus, the generated code is readable to a human programmer.

4 Demonstration that the generated code compiles and uploads to the EV3 brick

The following screenshots confirm that the generate code compiles and uploads to the EV3 brick.

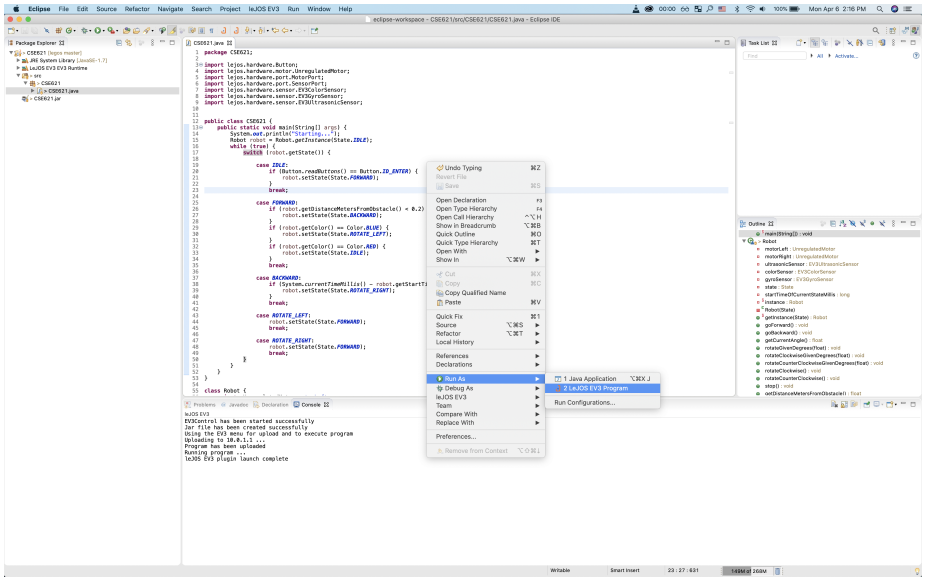

Figure 1: Running the generated code as LEJOS EV3 program in the EV3 brick

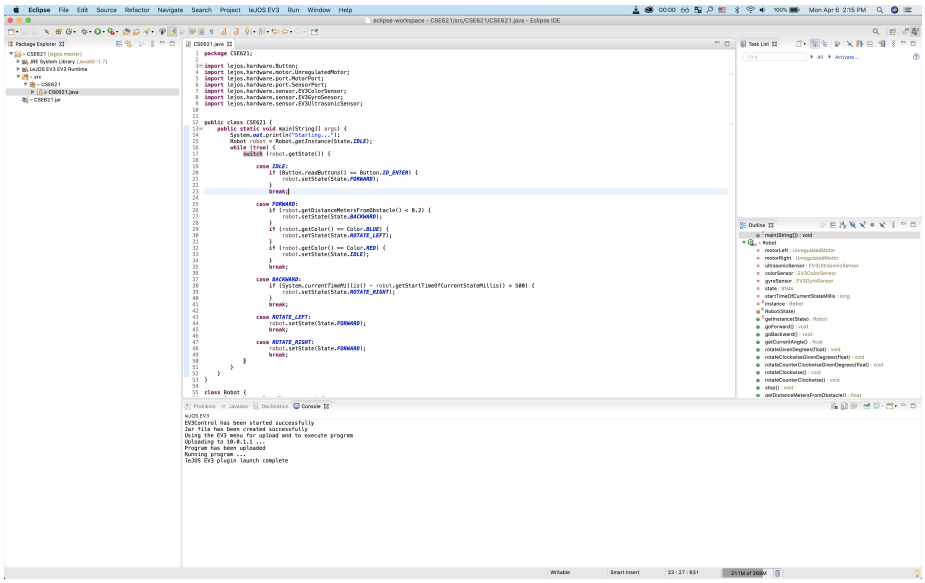

Figure 2: As evident from the console output, the generated code compiled and uploaded successfully in the EV3 brick

References

[1] Introducing json. `https://www.json.org/json-en.html`. Accessed: 03-25-2020.

[2] State machine diagrams. `https://www.uml-diagrams.org/state-machine-diagrams.html`. Accessed: 03-25-2020.