CSE 621: Fundamentals of Software Engineering
Project Phase 2 – Code Generation
Due Date: 11:59pm on Monday, April 6, 2020

Description

In this phase you will define and implement a model transformation to generate code for your robot based on behavior defined in StarUML statecharts.

Details

1. You will need to define the source and targets for your Code Generation. Recall that code generation is a type of model transformation; you need to understand the source meta-model (statecharts in StarUML) and your target meta-model (LeJOS code) well enough to begin to map between them. As such, you will need to document the general formats for the constrained set of statechars (see below), and the possible code implementations for the required behavior (see below).
2. Once you have an understanding of the source and target, you will need to define rules that will map from your source to your target for each element. These rules can be written in any format you see fit, as long as they are clear and exhaustive.
3. Given your source and target meta-models, along with a set of rules to implement your transformation, implement a model-transformation engine capable of transforming any input model into a java source file that could be compiled and run on the LeJOS firmware. You can implement this transformation in any language you see fit.

Source Models

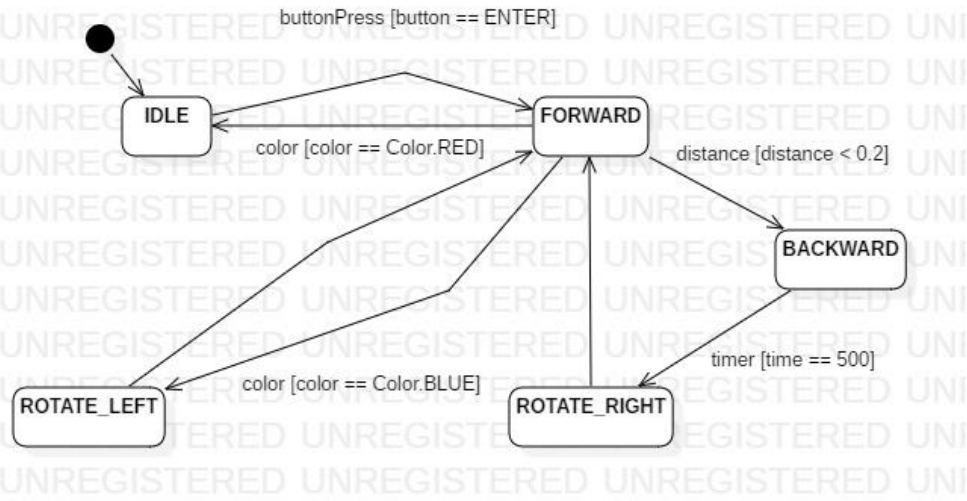
The input source files for this project will be statecharts written in StarUML. Specifically, the statecharts will be constrained to a limited number of states, triggers, and guards, which will be enumerated below. Your solution will need to handle all elements in the defined universe, and can safely ignore any other possible inputs.

The following are the elements of statecharts that your solution must address:

- States
  - IDLE – a state that will cause the robot to do nothing; stop moving and wait
  - FORWARD – move forward at a constant speed until the state is left
  - BACKWARD – move backward at a constant speed until the state is left
  - ROTATE_LEFT – rotate left 90 degrees
  - ROTATE_RIGHT – rotate right 90 degrees
- Triggers
  - buttonPress – a button is pressed on the EV3 brick
  - distance – a specific distance event occurs
  - timer – a set amount of time elapses
  - color – the color sensor detects a specific color
  - [blank] – a transition with no trigger occurs immediately when the state is resolved
- Guards
  - Guards for buttonPress trigger
    - button == ENTER – the enter button was the button pressed

        - button == ESCAPE – the escape button was the button pressed
        - button == UP – the up button was the button pressed
        - button == DOWN – the down button was the button pressed
        - button == LEFT – the left button was the button pressed
        - button == RIGHT – the right button was the button pressed
    - Guards for distance trigger
        - distance == X – distance is exactly X
        - distance < X – distance is less than X
        - distance > X – distance is greater than X
    - Guards for timer trigger
        - time == X – X ms have elapsed
    - Guards for color trigger
        - color == Color.X – the color X (constants of the color class, e.g. Color.BLUE) was detected by the color sensor

Example: below is an example statechart that conforms to this set of constraints:



The .mdj file for this statechart is attached to this phase of the project for you to explore.

Model File Format: You will need to extract the information you need (states, transitions, triggers, guards, etc.) from the textual representation of the file, since your transformation engine will use this as an input. You will need to open the .mdj file in a text editor to examine its contents to define your meta-model for statecharts (the textual representation of them, at least). Hint: you will only be concerned with the content of the "regions" section of the file as it contains all of this information. Your engine will essentially need to parse the file to extract relevant information.

Target Text

The target of this transformation will be a valid LeJOS file that will compile and run on the EV3 brick; however the specific types of control structures will be limited by the statechart style of execution. You will need to be able to define a template for all possible statecharts to be implemented in LeJOS Java. You will need to consider what it means to be "in a state" and "for a trigger to occur".

Essentially, you will have to understand how to represent states, transitions, triggers, and guards in LeJOS Java when defining your target meta-model.

Hints:

- you will likely need to take advantage of infinite looping, and only exit when a trigger event has been observed (recall `while(true){…}` will be helpful here)
- you will need to keep track of the current state in some way
- you are only required to check sensor values for valid transitions from the state you are in; e.g. in the example above, if the robot is moving forward, the code only needs to check for distance < 0.2 and the color sensor to detect blue; any other events are irrelevant in that state

Rules

With a solid understanding of source and target formats, define a set of rules that will map general inputs to outputs. For example, for every state in the input model, what code needs to be generated? What code is generated regardless of the states (e.g. instantiating motors and sensors, etc.)? Define the rules in any format you want, as long as the format is consistent and it is clear what your intent is; these rules are used by you to implement your transformation.

Implementing the Transformation

In whatever language or environment you are most comfortable, implement the rules you created to map from input statecharts to java source code.

Your tool/script/process/etc. should be named uml2lejos, and it should take as input one .mdj file and produce as output one .java file. The .java file you generate should be able to be inserted into an existing LeJOS EV3 Eclipse project that is part of a package named CSE621, and it should contain a class named CSE 621. The output of your generation should work in this project without any manual editing following generation.

Demonstrating Success

You will need to demonstrate that your tool works by generating the java code for the provided sample statechart. You can feel free to create your own statecharts and try to generate other code as well (either more or less complex), but the benchmark of success will be the provided model.

What to Submit

- Descriptions of your source and target meta-models; these can be in any format that **clearly describes the formats and demonstrates your understanding** of both input and output.
- The rules you defined to map from source to target models. Again these can be in any clear format, **as long as your intent is clear**.
- The source code for your implementation of the code generation process; submit your uml2lejos script/program/etc.
- The generated CSE621.java file that represents the SimpleStatechart.mdj that would run your robot based on the defined behavior.
- Demonstration that the generated code compiles and uploads to the EV3 Brick (screenshot or other confirmation, as appropriate).
- (Optional) Include a video of your robot demonstrating the behaviors

Grade Breakdown

There will be a total of **100 points** available, broken down as follows:

Category	High Quality	Medium Quality	Low Quality	No Points
Source Meta-Model Documentation	A clear definition of the source meta-model is provided that addresses the conceptual representation of statecharts as well as the textual representations in the .mdj file. **15 Points**	The defined meta-model of statecharts is mostly complete but some important elements or aspects are missing. Alternatively, the description is confusing. **12 Points**	There are significant aspects of StarUML statecharts missing from the source meta-model definition that lead to behavior not being represented in the solution system. **8 Points**	No documentation explaining the source meta-model has been provided. **0 Points**
Target Meta-Model Documentation	A clear definition of the target meta-model is provided that addresses the conceptual representation of statecharts within a java based source code environment (representing states, transitions, etc.). **15 Points**	The defined meta-model of java code is mostly complete but some important elements or aspects of state-based behavior are missing. Alternatively, the description is confusing. **12 Points**	There are significant aspects of StarUML statecharts missing from the target meta-model definition that lead to behavior not being represented in the solution system. **8 Points**	No documentation explaining the target meta-model has been provided. **0 Points**
Rule Definitions	The defined rules accurately address how to convert from StarUML statecharts to java representations without loss of information. The rules are clear and easy to understand. **10 Points**	There are some key rules missing that would cause for some statechart behavior to not be converted from input to output. Alternatively the rules are not well defined or documented. **8 Points**	Significant rules are missing to ensure that statechart behaviors are represented in the resulting generated java code. **6 Points**	No rule definitions have been provided. **0 Points**
Implementation of Code Generation – Technical Soundness	The implementation of code generation demonstrates technical soundness in the methods used to map from inputs to outputs; The rules defined previously are clearly represented in the implementation. **25 Points**	The approach demonstrates effectiveness, but is not very robust; it will work for several examples but does not generalize well to all possible input models. **20 Points**	The approach will work for the provided example, but the logic and soundness do not hold for a more general code generation problem. More attention needs to be paid to the source and target meta-models in the implementation. **15 Points**	The code provided does not implement a model transformation in any way. **0 Points**
Implementation of Code Generation – Correctness	The implementation of code generation shows evidence that it will generate valid outputs for any given input. **20 Points**	The code generation process is well designed but there are errors with the approach that lead to incorrect outputs. **12 Points**		The submitted approach does not implement code generation in this context. **0 Points**
Implementation of Code Generation – Well Documented Code	The submitted code is well documented following CSE Department Style Guidelines for the chosen language (or the most similar guideline available) **5 Points**	[redacted]		The submitted program does not meet the CSE Department Style Guidelines. **0 Points**
Demonstration of Effectiveness	The resulting CSE621.java code file is a valid LeJOS file and correctly compiles and loads to the EV3 brick without issue. **10 Points**	The generated CSE621.java file appears to be correct, but there were errors in attempting to compile and load to the EV3 brick. **8 Points**	Resulting code was submitted, but there is no indication it could be compiled or run on the EV3 brick. **6 Points**	No resulting source code for the example model was submitted. **0 Points**