# Assignment: Model-based Software Engineering

Thorsten Berger, Regina Hebig, Pierguiseppe Mallozzi, David Issa Mattos, Rebekka Wohlrab

March 31, 2017

Estimated time: 2 - 3 days of work. We suggest working in the teams.

## Introduction and objectives

In fall, you already have seen how robots, such as small turtle-bots can be controlled with specific languages that allow to define paths for their motion. When building industrial machines or robots, producers often create such user-friendly languages (often referred to as “domain-specific languages” or DSLs) to allow engineers, who are often not experts in software development, to use, control, and configure these machines or robots. Model-based software engineering provides the techniques to create such DSLs.

In this assignment, you will create such a user-friendly DSL. The objective of the task is to train your skills in:

- Creating the concrete syntax for a DSL using XText, and
- Creating the semantics for a DSL using template generation.

## Turtle-bot Mission DSL

The DSL that you will create allows to define missions for a turtle-bot (a small round robot that moves on wheels through the room).

The DSL should allow a user of the turtle-bot to specify waypoints in the area in which the turtle-bot should move, including:

- The types of waypoints that can be found within the area (e.g. GasStations, PetrolStations, Airport, etc.). Note, when writing your generator you will realize that you cannot specify the type of the waypoint for the simulator. Please, support nonetheless waypoint types, since it allows users of your DSL to analyze whether missions conform to their requirements.
- The concrete waypoints in the area, including their types (a waypoint can have multiple types, e.g. can be an Airport and a GasStation) and the position (x and y position relative to the area, e.g. x=0; y=0 is on corner of the area and x=length; y=width is the opposite corner)
- Note that the area defined by the simulator is 12*12. Your DSL will allow to redefine the area for real turtle-bots, but this will have no impact on the simulator.

Furthermore, the DSL shall allow to define the starting point of the bot, as one of the waypoints, and missions for the bot. Missions have a name (that acts as identifier) and a row of commands that can be are of three types:

- “Line” movement, specifies a sequence of waypoints that should be approached by the turtle-bot in the given order.
- Free-order (you can also call them “shortest-path”) movement, specifies a set of waypoints that should be approached by the robots, but the order in which they are approached is not specified. In these cases, the turtle-bot should chose the shortest path through the waypoints.
- “Return to start” movement, provides no further waypoints, but specifies that the turtle-bot shout return to its defined starting point.

To define the semantic of the DSL, you will have to translate these specifications to movement commands for the turtle-bot. Therefore, you should use ROS[1] to send simple movement commands to the turtle-bot. This includes some challenges:

1. You have to decide what movement commands should be send. That implies that you have to come up with a generic strategy to translate the abstract mission commands from your DSL. For example:
    a. The command "line" gives you target points that the robot should approach in the specified order. To translate this to movement commands, you have to figure out where the robot will be when the command when the command is called. Furthermore, you might want to consider other waypoints in the system, to make sure that the turtle-bot does not crash into them.
    b. For the command "free-order" (or "shortest-path") you need to run (in addition to the calculations for a normal line command) a short approximation of the shortest path to navigate from the current position of the robot to each of the specified waypoints[2].
2. The ROS implementation can be used from C++ and Python (there is also a Java version, however, unfortunately still not stable). We propose you to work with the Python interface. This means that you have to compile your DSL to the Python code that sends the commands to ROS.

***Note:*** *To simplify the task, we assume an ideal environment with no surprises, a precise movement of the robot (i.e. the robot does not deviate with time, you can conclude from its former movements on its current position), and that the model includes a precise and complete specification of the environment.*
*Furthermore, we assume that no elements are positioned at in between the natural numbered positions, i.e. a waypoint cannot be at x=2.5 or y=3.3.*
*The final assumption is that all missions end with a "return to start" command. Advanced students can enforce that with the help of an inbuilt OCL constraints. However, you can also just assume that this condition holds.*
*These assumptions will allow you build the DSL without having to read and interpret sensor data of the turtle-bot.*

## Input

As input for your task, you get the following elements:

- The metamodel for your DSL (see Figure 1), including generated editor code (e.g. the code that allows to store models conform to this metamodel)
- A basic XText grammar and editor generated based on the metamodel. Note that the syntax provided by this textual editor is by no means user-friendly. Your task is it to adapt this grammar.
- A plugin frame that you will fill to generate python code for the currently opened instance of your DSL.
- An example instance of the DSL that your resulting grammar should be able to parse and compile.

[1] To refresh your knowledge about ROS, this is a link to ROS tutorials: http://wiki.ros.org/ROS/Tutorials
[2] You might recognize this as a variant of the traveling salesman problem.

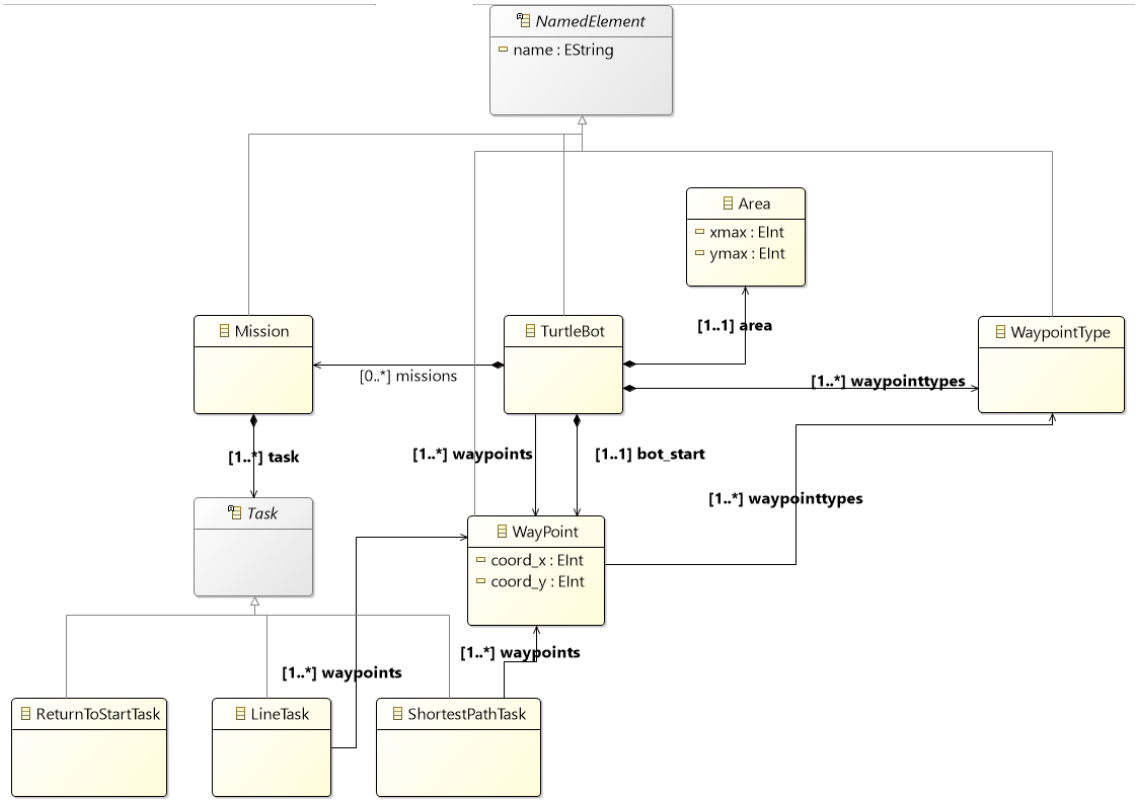


*Figure 1 DSL Metamodel*

## Procedure

1. Make yourself familiar with the metamodel, e.g. by using simple tree editor that we generated for the DSL.
2. Use the generated Xtext grammar as a starting point for creating a user-friendly grammar that can parse the given example.
3. Write code that navigates and reads the models created with your textual editor. This is where you can place the logic that decides about the movement commands.
4. Use string templates to compile the model into Python code that sends the commands to ROS.
5. Use the turtlebot_simulator[3] to test whether you create the correct ROS calls in the correct order

## Reporting format

Report on your results, by submitting your sources together with

- A short description of your Xtext grammar (explaining why it allows to parse the given example)
- A short description of your approach towards code generation
- A short manual, describing how to run your DSL
- 3 showcases of scripts (or models) specified in your DSL
- A screencast (or short video) that shows how one of the showcases is executed in the turtlebot_simulator

[3] http://wiki.ros.org/turtlebot_simulator