Model Engineering Lab 188.923 IT/ME VU, WS 2018/19	**Assignment 1**
Deadline: Upload (ZIP) in TUWEL until Sunday, November 4th, 2018, 23:55 Assignment Review: Wednesday, November 21st, 2018	25 Points

It is recommended that you read the complete specification at least once before starting with your implementation. If there are any parts of the specification that are ambiguous to you, don't hesitate to ask in the TUWEL forum for clarification.

Lab Overview

Lab 1: In Lab 1, your goal is to develop the *abstract syntax* of the so-called *Rover Modeling Language (RoverML)* using *Ecore* and *OCL*. RoverML allows the modeling of rovers or other moving robots, and their programs which control their behavior.

Lab 2: In Lab 2, you will develop the *concrete syntax* of RoverML as well as *editor support* for RoverML models using *Xtext* and *Sirius*.

Lab 3: In Lab 3, you will develop model-to-model transformation for RoverML models using *ATL* and *Henshin*.

Lab 4: In Lab 4, you will develop a model-to-code transformation that generates code from RoverML models using *Xtend*. The generated code will provide a simulation of the modeled rover and its program.

Lab 1: Metamodeling

The goal of this assignment is to develop the abstract syntax of the *Rover Modeling Language (RoverML).* Therefore, you have to develop a metamodel for RoverML with Ecore, as well as OCL constraints defining additional well-formedness rules for RoverML models.

Part A: Development of the Metamodel

Modeling Language Description

RoverML is a modeling language for modeling rovers and the programs they execute. A rover model defines the rover's topology, i.e., the individual components of a rover, and the rover's program, i.e., the commands which define its comportment and actions. Rovers are used in education (e.g. teaching kids programming), research (e.g. Mars rovers) and industrial applications (e.g. automated rovers in a warehouse transporting goods).

The concepts provided by the RoverML for modeling rovers and their programs are described using the examples depicted in the figures below.

Figure 1 shows an example for a small Rover, the Ozobot, which is used in education. The Ozobot can be controlled by a simple block-based programming language. The user can create a program, upload it to the bot and let the bot execute it. The bot has different actuators and sensors, e.g. it has a motor, lights, a proximity sensor, etc.

Figure 2 shows an example for a rover program and a rover modeled with RoverML. Figure 3 shows another example.

Table 1 describes the different Rover modeling concepts in detail

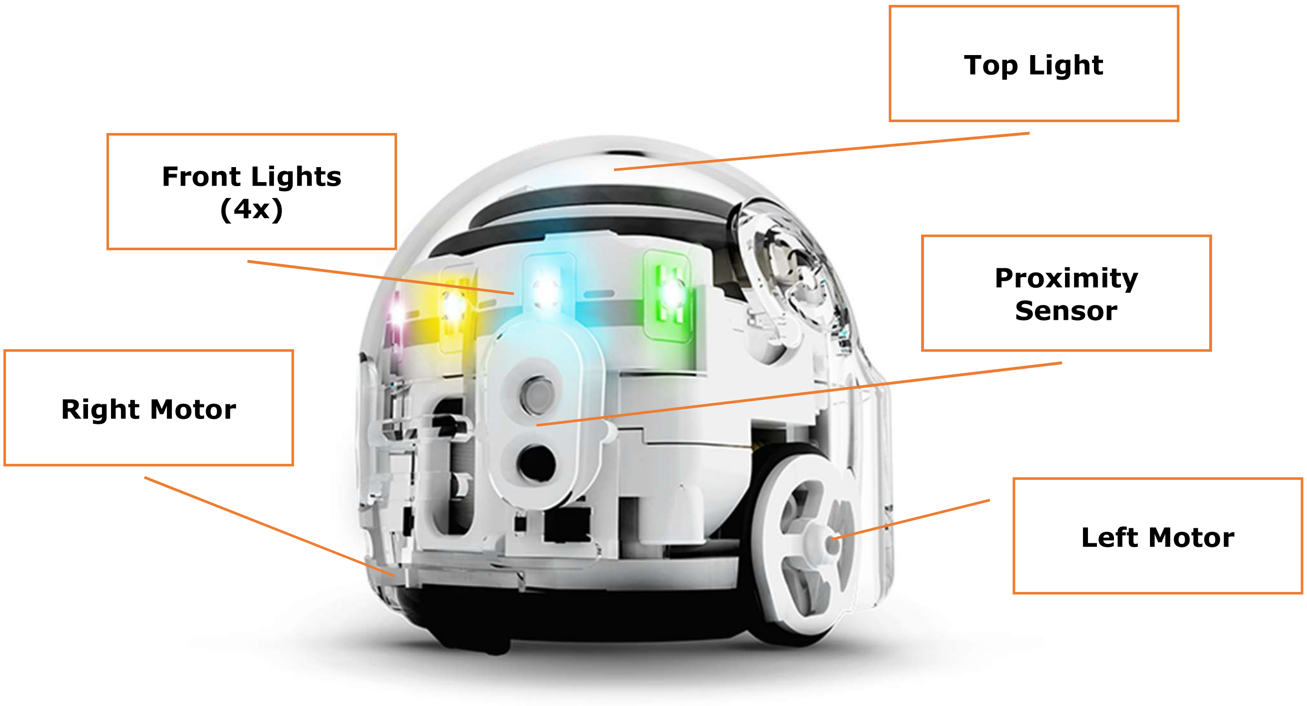


Figure 1: Ozobot

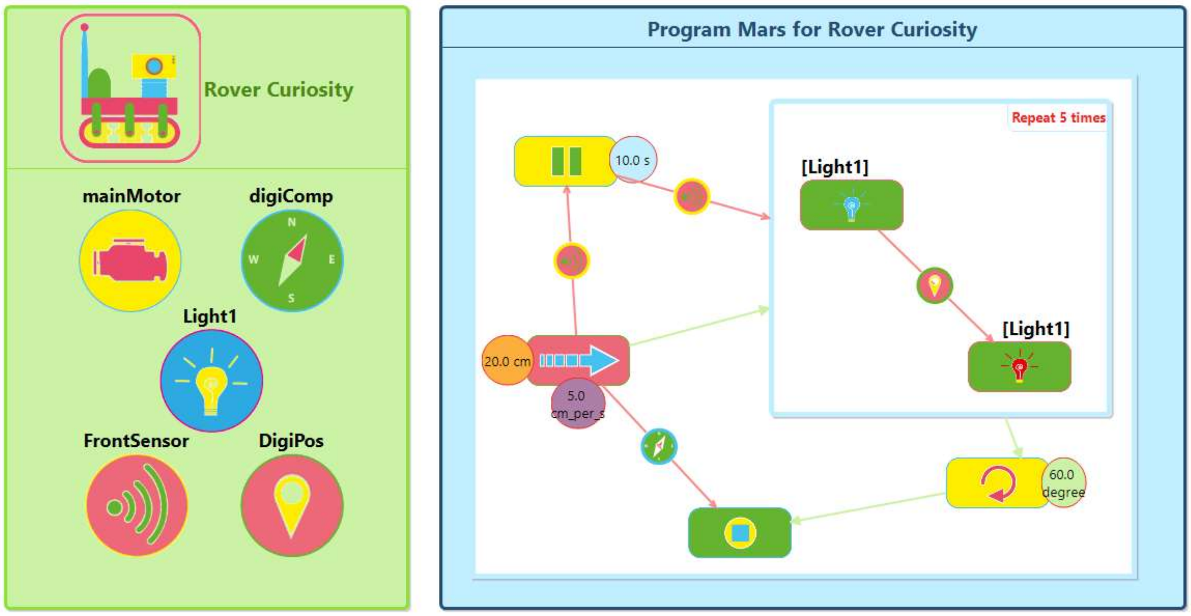


Figure 2: A rover program

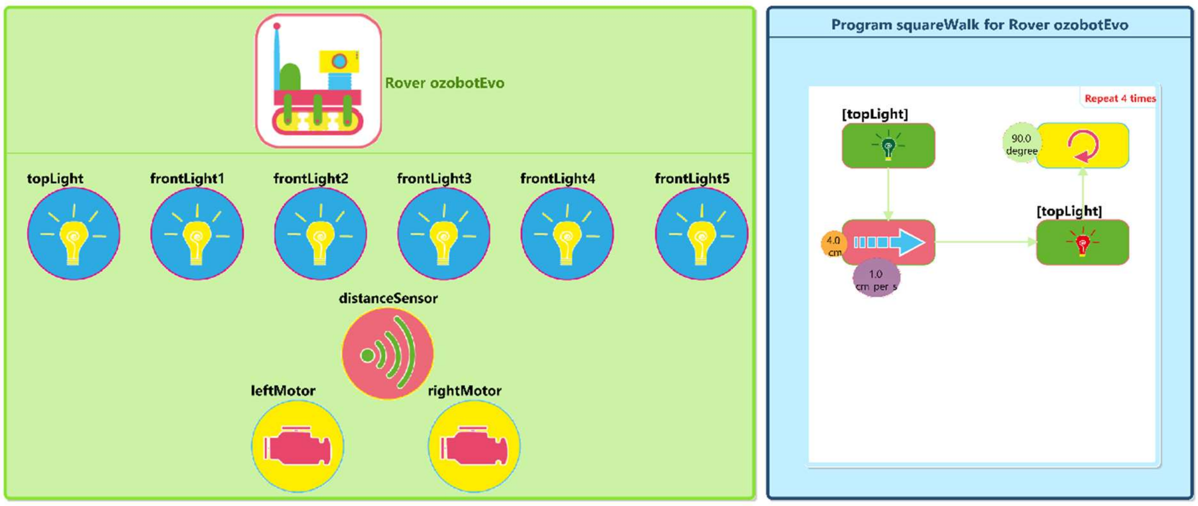


Figure 3: Another rover program

Syntax	Description
	A *rover* and its *program* are contained in a *system*. In such a system, multiple *rovers* and *rover programs* can exist. Example: The model in Figure 2 defines a rover and a program which references this rover.
	Rovers have a name and consist of various *components*. Example: The Rover in Figure 2 is named Curiosity and has five components: mainMotor, Light1, digiCom, digiPos and frontSensor.
	Components of a rover have a unique name. There are two types of components: *sensors* and *actuators*. Sensors may be a GPS, a compass or a distance sensor. Possible actuators are motors and lights. All sensors report their last sensed value. The GPS measures the current position (in a 2D plane). The distance sensor measures the (remaining) distance to an object. The compass tells which direction (as an angle) the rover is currently facing. Example: In Figure 2, digiComp, digiPos and frontSensor are sensors. The sensor digiComp is a compass.
	A *motor* is a specific type of actuator. Each Rover needs to have at least one motor. Example: In Figure 2, mainMotor is a motor of the rover Curiosity.
	A *light* is a specific type of actuator. Lights are an optional component. Example: In Figure 2, Light1 is a light of the rover Curiosity.
🟩	A *rover program* is a set of *commands* executed by a rover. The rover which executes the program must be specified. A rover program has a *name*. Example: The rover program Mars in Figure 2 is executed by the rover Curiosity.
☐	A Rover program is composed of exactly one block. A *block* contains *commands* and *transitions*. A specific type of block is a *repeat* block. Example: The rover program Mars in Figure 2 is composed of one main block which contains 5 commands.
	A *command* specifies an action of the rover. There are several different commands: *SetLightColor*, *Wait*, *Move*, *Rotate*, *Terminate* or *Repeat*. Some commands use *quantities* or reference

	components. Every command of a block except one (start command) must have at least one incoming transition. Example: In the main Block in Figure 2, the commands Move, Wait, Repeat, Rotate and Terminate are represented.
→	A *transition* connects two commands with each other. A transition always has one source command and one target command. A transition can also be *triggered*. If two or more transitions have the same source command, at most one of them can be a regular transition, the others must be triggered. Example: In Figure 2, the command Move and Repeat are linked through a transition.
💡	*SetLightColor* is a specific command. It specifies one or more lights that should be set to a specific *color*. There are five possible colors: none (i.e. the light should be turned off), red, green, blue and yellow. Example: In Figure 2, the repeat block contains 2 SetLightColor commands, for turning the light blue and red.
➡	*Move* is a specific command. It tells the rover to move a certain *length* with a certain *velocity*. If no length is set, the rover moves indefinitely with the given velocity. If the rover is moving indefinitely, the outgoing transition of the move command must be a triggered one. Example: In Figure 2, the command Move has a length of 20 cm and a velocity of 5 cm per s.
↻	*Rotate* is a specific command. It tells the rover to rotate to an *angle*. This command cannot be used twice in a row. Example: In Figure 2, the command rotate has an angle of 60 degrees.
⏸	*Wait* is a specific command. It tells the rover to wait for the next input for a specified amount of *time*. Example: In Figure 2, the command wait has an incoming transition from the command Move and a duration of 10 s.
⏹	*Terminate* is a specific command that terminates the program of the rover. It can only be defined once per block. This command is not allowed to have any outgoing transitions. Example: This is the last command in the main block from the mars program in Figure 2.
🔄	*Repeat* is both a *command* and a *block*. Repeat has a count, i.e. the number of times the commands contained are repeated. The commands must be repeated at least once. If the count is set to "**-1**", the block is repeated indefinitely. In this case, the outgoing transition of this command needs to be a triggered one.

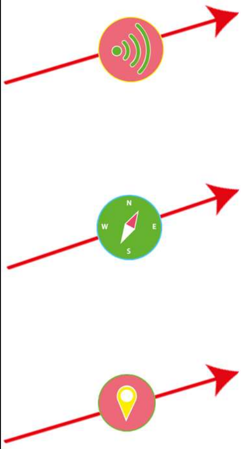
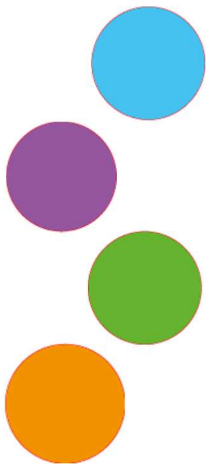
	Example: In Figure 2, the main block contains a Repeat command which contains 2 SetLightColor commands and is repeated 5 times.
	A *triggered transition* is a type of transition. It is used to determine when the transition should fire based on sensor values. Each type of triggered transition is related to an available sensor. There are three different types of triggers: a *distance sensor trigger,* a *compass trigger* and a *GPS trigger.* A triggered transition references the *quantity* to which it is compared. It also states which *operator* is used for comparation: smaller, greater, equal or unequal. The comparison value should fit the type of the triggered transition, i.e. a distance sensor trigger uses *length* as comparison value. The sensor should fit the type of the triggered transition, i.e. a distance sensor trigger may only reference a distance sensor. Example: In Figure 2, the rover program has 4 triggered transitions: two distance sensor triggers, a compass trigger and a GPS trigger.
	A *quantity* can either be a single Quantity or a Position. A single quantity has a value. There are different kinds of single quantities: *time*, *velocity*, *length, angle.* Time has a specified unit of time: nanoseconds, milliseconds, seconds, minutes or hours. A time quantity cannot be negative. Velocity has a specified unit: millimeters per second or centimeters per second. Velocity cannot be zero and must either be positive (moving forward) or negative (moving backward). Angle has a specified unit: radians or degrees. Length has a specified unit: millimeters, centimeters or meters. Position: Refers to a (GPS) position. A position consists of two *lengths*, one defines the X position and the other one defines the Y position. Example: In Figure 2, the command Move has a Velocity and a Length while the command Wait has a Time.

Table 1: RoverML concepts

Task Description
Develop the metamodel for RoverML with EMF's metamodeling language Ecore and define 8 OCL constraints for ensuring the well-formedness of RoverML models.

- Make sure that you specified the metamodel as precisely as possible, i.e., the metamodel must contain all described language concepts.
- Use the "Sample Ecore Model Editor", the "Ecore Editor", or the "Sirius Diagram Editing Editor" to create the metamodel.
- It is not required to define operations for the metaclasses.
- Constraints regarding the well-formedness of RoverML models, which cannot be ensured by the metamodel only, should be defined as OCL invariants. You have to implement 8 OCL constraints. Make sure that you address well-formedness rules defined in the modeling language description above – do not invent additional well-formedness rules.
- Duplicate OCL constraints do only count for one of the eight required constraints. An OCL constraint is duplicate if the very same OCL constraint is defined for several context metaclasses. An example would be to define for several metaclasses an OCL constraint that checks that the value of an attribute is not negative.
- Use the "OCLinEcore Editor" to add the OCL constraints to your metamodel.

IMPORTANT: Do NOT translate the example models depicted in Figure 2 and Figure 3 to Ecore, but develop a modeling language (i.e., a metamodel) to define the language concepts as described above that allow to define the example models.

Part B: Metamodel Testing

Task Description

In Part A you developed the metamodel for RoverML using Ecore and defined 8 OCL constraints. They have to be tested in this part of the lab.

- Use the functionalities of EMF to generate a tree-based model editor.
- Model the example rover depicted in Figures 2 and 3 with the help of this editor. Assume attribute values that are not explicitly given in this document (e.g., names of components, values, etc.).
- For each defined OCL constraint create one small example model which fulfills the constraint and one small example model which does not fulfill the constraint. This means that you have to prepare 16 small example models for testing the 8 required constraints. The invalid example models should only violate the tested constraints; all other constraints should be fulfilled.
- Put the test models into an own project "roverml.examples". You can create this project by selecting *File → New → Other... → General / Project.*

Submission & Assignment Review

At the assignment review, you will have to present your metamodel for RoverML, the additionally defined OCL constraints, the generated modeling editor for RoverML, as well as the example models for testing the metamodel and the OCL constraints. The metamodel has to contain all described concepts and it must be possible to model the example rovers and rover programs depicted in Figure 2 and Figure 3. Furthermore, the correctness of the defined OCL constraints must be shown using the prepared example models. You also have to show that you understand the theoretical concepts underlying the assignment.

Upload the following components in TUWEL:

- Entire EMF projects exported as archive file, containing:
    - Ecore modeling project "roverml" including the Ecore model (roverml.ecore), the Ecore diagram (roverml.aird), the generator model (roverml.genmodel), and the generated model code
    - Generated edit project "roverml.edit"
    - Generated editor project "roverml.editor"
    - Project "roverml.examples" containing the example RoverML models:
        - Model of the example rover depicted in Figures 2 and 3 (RoverExample.roverml).
        - Models for testing the OCL constraints (constraint[1...8]valid.xmi, constraint[1...8]invalid.xmi)

All group members must be present at the assignment review. The registration for the assignment review can be done in TUWEL. The assignment review is divided into two parts:

- **Submission and group evaluation:** 20 out of 25 points can be reached.
- **Individual evaluation:** Every group member is interviewed and evaluated separately. The remaining 5 points can be reached. If a group member does not succeed in the individual evaluation, the points reached in the group evaluation are also revoked for this student, which results in a negative grade for the entire course.

Notes

Setting up Eclipse

For the lab part, we provide a description of **how to set up an Eclipse version** that contains all necessary plug-ins for all assignments. This Eclipse setup guide is available in TUWEL: https://tuwel.tuwien.ac.at/mod/page/view.php?id=388945

Metamodeling with Ecore

Once you have installed Eclipse and all necessary plug-ins, you can **create Ecore models**. Therefore, select *File* ➔ *New* ➔ *Other* ➔ *Eclipse Modeling Framework* ➔ *Ecore Modeling Project*, click *Next >,* enter the *Project name* "roverml" on the next page and confirm it with *Next >*, enter the *Main package name* "roverml" and complete the wizard by clicking *Finish*. The created project "roverml" contains already some auto-generated files, which can be used to define an Ecore model (roverml.ecore, roverml.aird).

The **Ecore diagram** (.aird file) provides a graphical view on the metamodel defined in the Ecore model (.ecore file). For opening the Ecore diagram file make sure that you are in the *Model* perspective (it should be selected in the upper-right corner of Eclipse; if, for instance, the *Resource* or *Java* perspective is opened, select *Window* ➔ *Perspective* ➔ *Open Perspective* ➔ *Other…* ➔ *Modeling*). In the *Modeling* perspective, you can open the .aird file by double clicking on it and expand it until the diagram "roverml" is visible. By double clicking, you can open this diagram and start defining your metamodel by dragging metaclasses onto the canvas (Palette *Classifier* ➔ *Class*).

You can also modify the **Ecore model** (.ecore file) directly with a **tree editor** (right click on the .ecore file and select *Open with* ➔ *Sample Ecore Model Editor* or *Ecore Editor*). However, the graphical view in the Ecore diagram will not be automatically updated. To add elements created in the tree editor to the diagram, expand the .ecore file in the *Model Explorer* and drag and drop the elements onto the diagram canvas.

You can decide, whether you create the metamodel with the graphical editor in the concrete syntax or with the tree-based editor in the tree-based abstract syntax. If you decide to use the tree editor, generate a graphical view after creating the metamodel and layout it such that it is readable.

Testing the metamodel

To **test the metamodel** you can right click on the created .ecore file and select *EPackages registration* ➔ *Register EPackages into repository*. Now, open the .ecore file in the tree-based editor, right click on the metaclass you want to test and select *Create Dynamic Instance*. Specify a location and name for the new model (e.g., *test.xmi*). To determine whether the whole model is correct or not right-click on the root element of the model and select *Validate.*

Creating a model editor

To create a tree-based editor from your metamodel, open the **Generator model** (.genmodel file), right click on the root element and select *Generate All*. As a result, three additional projects with the postfix *edit*, *editor*, and *test* are created.

To **test your editor**, open the *MANIFEST.MF* file of the previously generated *editor* project. In the *Overview* of the *MANIFEST.MF* click *Launch an Eclipse application*. In the new opened Eclipse instance, you can create a RoverML model by selecting *File* ➔ *New* ➔ *Other* ➔ *Example EMF Model Creation Wizards* ➔ *RoverML Model* (depending on the settings in your .genmodel file).

For more information about EMF visit http://www.eclipse.org/emf.

OCL in Ecore

You can add OCL constraints to your metamodel by opening the .ecore file with the OCLinEcore editor by right clicking on your .ecore file and selecting *Open With* ➔ *OCLinEcore Editor*.

The *OCL Console* is very helpful for defining and testing *OCL constraints*. To make use of the *OCL Console* open a test model (e.g., *test.xmi*) with the *Sample Reflective Ecore Model Editor*, right click on any model element and select *OCL* ➔ *Show Xtext OCL Console*.

Further information about OCLinEcore can be found in the help contents of the Eclipse IDE (*Help* ➔ *Help Contents* ➔ *OCL Documentation*).