














Model Engineering Lab 188.923 IT/ME VU, WS 2016/17	Assignment 2
Deadline: Upload (ZIP) in TUWEL until Sunday, November 20 th , 2016, 23:55 Assignment Review: Wednesday, November 23 rd , 2016	25 Points

Concrete Syntax

The goal of this assignment is to develop the concrete syntax of the *States Modeling Language (StatesML)* using Xtext and Sirius. In particular, you will develop two alternative concrete syntaxes for the state modeling concepts provided by StatesML: a textual concrete syntax and textual editor with Xtext in Part A of this assignment, and a graphical concrete syntax and graphical editor with Sirius in Part B of this assignment.

Assignment Resources

 ME_WS16_Lab2_Resources.zip

-  at.ac.tuwien.big.statesml (Modeling project with solution of Assignment 1)
-  at.ac.tuwien.big.statesml.edit (Edit project with solution of Assignment 1)
-  at.ac.tuwien.big.statesml.editor (Editor project with solution of Assignment 1)
-  at.ac.tuwien.big.statesml.statesystem (Xtext project for developing the textual syntax of StatesML)
-  at.ac.tuwien.big.statesml.statesystem.ide (Xtext project for the textual editor of StatesML)
-  at.ac.tuwien.big.statesml.statesystem.ui (Xtext project for the textual editor of StatesML)
-  at.ac.tuwien.big.statesml.xtext (Plugin for connecting the textual editor and the tree editor of StatesML)
-  at.ac.tuwien.big.statesml.xtext.ui (Plugin for connecting the textual editor and the tree editor of StatesML)
-  at.ac.tuwien.big.statesml.design (Sirius project for developing the graphical syntax of StatesML)
-  at.ac.tuwien.big.statesml.examples.xtext (Project with StatesML example models defined with the textual editor of StatesML)
-  at.ac.tuwien.big.statesml.examples.sirius (Project with StatesML example models defined with the graphical editor of StatesML)
-  lab2.pdf (this document)

Before starting this assignment, make sure that you have all necessary components installed in your Eclipse. A detailed installation guide can be found in the TUWEL course¹.

It is recommended that you read the complete assignment specification at least once. If there are any parts of the assignment specification or the provided resources that are ambiguous to you, don't hesitate to ask in the forum for clarification.

¹ Eclipse Setup Guide: <https://tuwel.tuwien.ac.at/mod/page/view.php?id=289051>

Part A: Textual Concrete Syntax

In the first part of this assignment, you have to develop the textual concrete syntax (grammar) for StatsML with Xtext. Additionally, you have to implement scoping support for the textual editor that is generated from the developed grammar.

The grammar has to be built upon the sample solution of the StatesML metamodel provided in the assignment resources (at.ac.tuwien.big.statesml/model/statesml.ecore).

Do not use the metamodel you have developed in Assignment 1 but the sample solution provided in the assignment resources!

A.1 Xtext Grammar for StatesML State Systems

Develop an Xtext grammar that covers all modeling concepts provided by StatesML for defining state systems. This comprises the modeling concepts state system, state, transition, selection divergence, selection convergence, edge, function call, parameter value, value specification, event, change expression, trigger, and attribute.

Hint: The representation “state systems class diagram” of the Ecore diagram file `statesml.aird` provided in the assignment resources at at.ac.tuwien.big.statesml/model/statesml.aird gives a good overview of StateML’s modeling concepts for defining state systems.

The Xtext grammar has to follow the example model that is depicted in Figure 1. This textual example models is also available in full length in the examples project: at.ac.tuwien.big.statesml/examples.xtext/simpleLineFollowing/LineFollowingBehavior.statesystem. A corresponding model serialized with XML is also provided in the file `LineFollowingBehavior.statesml` located in the same folder.

Figure 1: Example StatesML state system in textual concrete syntax

```
1 statesystem LineFollowingBehavior for SimpleLineFollowingRobot {
2   attributes {
3     timesReversed:Integer
4   }
5   events {
6     SurfaceColorIsRed: equals(s1=surfaceColor,s2="red")
7     SurfaceColorIsWhite: equals(s1=surfaceColor,s2="white")
8   }
9   states {
10    initial InitialStep
11    FollowingLine{
12      followLine()
13    }
14    Reversing{
15      pickDirection(direction="back")
16      shineLight(color="red",durationInMs=500,blinking=true)
17      incrementValue(value=timesReversed)
18    }
19    terminal TerminalStep
20  }
21  transitions {
22    FollowLine
23    Terminate triggered-by SurfaceColorIsWhite
24    Reverse triggered-by SurfaceColorIsRed
25    RepeatFollowLineAfterReversing
26  }
```

```

27  control {
28      convergence ConvergeToFollowLine
29      divergence DivergeAfterFollowLine
30  }
31  edges {
32      e1: InitialStep => FollowLine
33      FollowLine => ConvergeToFollowLine
34      ConvergeToFollowLine => FollowingLine
35      FollowingLine => DivergeAfterFollowLine
36      DivergeAfterFollowLine => Terminate
37      Terminate => TerminalStep
38      DivergeAfterFollowLine => Reverse
39      Reverse => Reversing
40      Reversing => RepeatFollowLineAfterReversing
41      RepeatFollowLineAfterReversing => ConvergeToFollowLine
42  }
43 }

```

To implement the Xtext grammar for StatesML state systems, perform the following steps:

1. Setting up your workspace

As mentioned in the beginning, the Xtext grammar has to be based on the sample solution of the StatesML metamodel. We provide this sample solution as well as skeleton projects for the Xtext grammar as importable Eclipse projects.

Import projects

To import the provided project in Eclipse select *File* → *Import* → *General/Existing Projects into Workspace* → *Select archive file* → *Browse*. Choose the downloaded archive *ME_WS16_Lab2_Resources.zip* and import the following projects:

1. at.ac.tuwien.big.statesml
2. at.ac.tuwien.big.statesml.edit
3. at.ac.tuwien.big.statesml.editor
4. at.ac.tuwien.big.statesml.statesystem
5. at.ac.tuwien.big.statesml.statesystem.ide
6. at.ac.tuwien.big.statesml.statesystem.ui
7. at.ac.tuwien.big.statesml.xtext
8. at.ac.tuwien.big.statesml.xtext.ui

Register the metamodel

Before you start implementing your grammar, you have to register the StatesML metamodel in your Eclipse instance, such that the Xtext editor with which you develop your grammar knows which metaclasses you want to use. You can register your metamodel by clicking right on *at.ac.tuwien.big.statesml/model/statesml.ecore* → *EPackages registration* → *Register EPackages into repository*. Clean your projects afterwards by selecting *Project* → *Clean...* *Clean all projects*.

After performing these steps, you should have eight projects in your workspace without any errors (there may be warnings about missing plugin.xml files, which will be generated later).

2. Developing your Xtext grammars

Define the Xtext grammar in the file `StateSystem.xtext` located in the project `at.ac.tuwien.big.statesml.statesystem` in the package `src/at.ac.tuwien.big.statesml`. Documentation about how to use Xtext can be found in the lecture slides and in the Xtext documentation².

Hint: Make use of the ID rule defined in the base grammar `org.eclipse.xtext.common.Terminals` to specify the names of different model elements, e.g., the *name* of state systems.

Hint: When providing cross-references to other model elements, make sure to use the rule `QualifiedName` already defined in the Xtext grammar `StateSystem.xtext`, e.g., `referenceToClass=[Class|QualifiedName]`.

3. Testing your editors

Generate Xtext artifacts

After you have finished developing your Xtext grammar in `StateSystem.xtext`, you can automatically generate the textual editor. The Xtext grammar project `at.ac.tuwien.big.statesml.statesystem` is accompanied by a so called model workflow file `GenerateStateSystem.mwe2`, which orchestrates the generation of the textual editor. By default, the workflow will generate all the necessary classes and stub classes in the grammar project and in the related ide-project and ui-project.

Note: In the workflow file you can specify whether the generator should create Xtend³ or Java stub classes for extension (flag `generateXtendStub`). For implementing scoping support you will have to extend one of these stubs (more information about the scoping support follows below). If you are not familiar with the syntax of Xtend, you can leave the setting as it is (we have changed the default setting). Otherwise you can change the flag to true and generate Xtend stubs.

To run the generation, right-click on the workflow file `GenerateStateSystem.mwe2` and select *Run As → MWE2 Workflow*. This will start the editor code generation. Check the output in the Console to see if the generation was successful. Please note that after each change in your grammar, you have to re-run the workflow to update the generated editor code.

If you start the editor code generator for the first time, the following message may appear:

```
*ATTENTION*
It is recommended to use the ANTLR 3 parser generator (BSD licence -
http://www.antlr.org/license.html). Do you agree to download it (size 1MB) from
'http://download.itemis.com/antlr-generator-3.2.0.jar'? (type 'y' or 'n' and hit enter)
```

Type *y* and download the jar file. It will be automatically integrated into your project.

Start a new Eclipse instance with your plugins

For starting the generated editor, we have already provided a launch configuration located in the project `at.ac.tuwien.big.statesml.statesystem`, which is called *"ME Lab 2 Runtime.launch"*. Run it by right-clicking on it and selecting *Run As → ME Lab 2 Runtime*. You can have a look at the configuration by right-clicking on that file and selecting *Run As → Run Configurations....*

² Xtext documentation: <http://www.eclipse.org/Xtext/documentation/>

³ Xtend: <http://www.eclipse.org/xtend/documentation/>

Start modeling

In the newly started Eclipse instance, you can create a new empty project (*File* → *New* → *Project* → *General/Project*) and create a new file (*File* → *New* → *File*) with the extension *.statesystem* in this project. If you are asked to add the Xtext nature to the project ('Do you want to convert 'projectname' to an Xtext project?') hit *Yes*. Right-click on the created **.statesystem* file and select *Open With* → *StateSystem Editor*. Now you can start modeling your state system to test the generated textual editor.

Hint: By pressing *Ctrl+Space* you activate content assist/auto completion, which provides you a list of keywords or elements that can be input at the next position.

Hint: When you have cross-references to other elements in your grammar, you can check which element is actually referenced in a model by using the linking feature. For this just hold *Ctrl* and click with the mouse on the cross-reference.

Hint: State systems created with the generated textual editor have to refer to elements defined in system units. E.g., the state system with the name "LineFollowingBehavior" that is defined in the example model depicted in Figure 1 refers to the system unit with the name "SimpleLineFollowingRobot". This system unit is defined in the model *LineFollowingRobotLibrary.statesml* provided in the project *at.ac.tuwien.big.statesml.examples.xtext/simpleLineFollowing/*.

For defining system units you can use the tree editor that was generated from the StatesML metamodel: Select *File* → *New* → *Other...* → *Example EMF Model Creation Wizards / StateML Model*, select the project where you define your example models, provide a name for the model file to be created, hit *Next*, select the model object "System Unit" on the next page and hit *Finish*. The model is then opened with the tree editor and you can define the system unit with its attributes and functions.

Alternatively, you can also use the before mentioned example model *LineFollowingRobotLibrary.statesml* for testing your grammar and textual editor.

Hint: Please note that after each change in your grammar, you have to re-run the workflow to update the generated editor code. Furthermore, you will also have to restart the Eclipse instance that you use for testing the generated editor. We also advise you to always clean your example project (*Project* → *Clean...* *Clean all projects*) and re-open your example model (**.statesystem*).

Check examples

In the newly started Eclipse instance, you can import some example models, which we have included in the project *at.ac.tuwien.big.statesml.examples.xtext* provided in the assignment resources. Just import the project in the workspace (see "Setting up your workspace").

LineFollowingBehavior.statesystem (folder <i>simpleLineFollowing</i>)	This file contains the model that is depicted in Figure 1. When opening this file with your Xtext editor (<i>StateSystem Editor</i>), the editor should show no errors and no warnings.
--	---

minimal.statesystem	This file provides a minimal state system that fulfills all OCL constraints defined in the metamodel of StatesML. You can use this file to conveniently produce your own test examples (copy the file and adapt/extend it).
----------------------------	---

Hint: Make sure that values for attributes and references are actually assigned to the model elements. For checking this, you can open *.statesystem files with the *Sample Reflective Ecore Model Editor* (right click on the *. statesystem file and select *Open With* → *Other...* → *Sample Reflective Ecore Model Editor*). Investigate the model elements in the tree and their attribute values and references in the *Properties* view (if the *Properties* view is not shown, right click on any model element and select *Show Properties View*).

A.2 Scoping Support for StatesML State Systems

In the grammar that you have developed, you should have defined some cross-references, e.g., a state system refers to a system unit and a function call refers to a function. When testing your editor, you will notice that whenever there is a cross-reference, the content assist (CTRL+Space) will provide all elements of the respective type that the editor can "find". Which elements the editor provides is defined in the scoping⁴ of the respective reference. By default, the editor searches through the whole class-path of the project and you will notice that if you have many models in your project, a lot of elements will show up as possible reference.

To restrict this behavior, Xtext provides a stub where developers can define their own scope. This stub can be found in the grammar project `at.ac.tuwien.big.statesml.statesystem` in the package `scoping` (`StateSystemScopeProvider.java`). Your task is to implement the following scoping behavior in addition to the already existing ones:

1) Scoping for parameters of function calls

A parameter value contained by a function call has to reference an *in* parameter of the called function, i.e., a parameter that is contained by the called function and that has the *direction* attribute set to `ParameterDirectionKind.IN`.

Example: A function call of the function named "equals" defined in the model `PrimitiveDataTypes.statesml` may only refer to the *in* parameters of this function, i.e., the parameters named "s1" and "s2" (see line 6 and line 7 of the example depicted in Figure 1). It is not allowed to refer to the *return* parameter named "result" of this function or any other parameter contained by any other function.

2) Scoping for attributes of attribute value specifications

An attribute value specification defined as part of a function call has to reference either an attribute that is contained by the state system containing the attribute value specification or an attribute that is contained by the system unit for which the containing state system is defined. Furthermore, the referenced attribute must have the same type as the parameter of the parameter value containing the attribute value specification.

Example: Any attribute value specification defined in the state system "LineFollowingBehavior" shown in Figure 1 has to reference either an attribute contained by this state system (i.e., the attribute "timesReversed") or an attribute contained by the system unit "SimpleLineFollowingRobot" (i.e., the attribute "surfaceColor"). If the attribute value specification is defined for the parameter "s1" of the function "equals", such as in line 6 of Figure 1, the attribute referenced by the attribute value specification has to be of the same type as the parameter "s1", i.e., of type "String". Thus, only the attribute "surfaceColor" may be referenced by this attribute value specification.

⁴ http://www.eclipse.org/Xtext/documentation/303_runtime_concepts.html#scoping

Hint: Xtext uses a so called polymorphic dispatcher to handle declarative scoping. This dispatcher uses the Reflection API to search for methods with a specific signature in the ScopeProvider class. How you can use this is explained in the lecture slides and in the Java documentation of the class `AbstractDeclarativeScopeProvider`⁵, which is the super class of your scoping stub `StateSystemScopeProvider.java`.

After you have implemented the scoping as defined above, start again a new instance of your Eclipse (use the provided launch configuration) and test your scoping with the content assist (Ctrl+Space) and linking feature (Ctrl+Left Click).

Check examples

In the newly started Eclipse instance you can check the remaining models provided by the examples project `at.ac.tuwien.big.statesml.examples.xtext` and see whether your scoping covers these cases. However, it should be noted that these tests do not cover all possible cases and additional testing of the implemented scoping from your side is required.

test1.statesystem (folder scoping) In this example, the event "SurfaceColorIsRed" defined in line 6 contains a function call for the function "equals". This function call defines a parameter value for the *return* parameter "result" of the function "equals". Since this is a *return* parameter and not an *in* parameter, the editor should show an error marker for this line. Similarly, also line 7 defines a function call for the function "equals", which defines a parameter value for the *in* parameter "direction". Because the *in* parameter "direction" is defined by the function "pickDirection" and not by the called function "equals", the editor should also show an error marker for this line. When you use the content assist in line 6 after "SurfaceColorIsRed:equals(", only the *in* parameters of the function "equals" should be offered, i.e., "s1" and "s2". After choosing "s1", the errors should disappear. The error for line 7 can be fixed in the same way.

test2.statesystem (folder scoping) In this example, the event "SurfaceColorIsRed" defined in line 6 contains a function call for the function "equals". The parameter value defined for the parameter "s1" defines an attribute value specification referring to the attribute "timesReversed" defined in line 3. Because the attribute "timesReversed" is of type "Integer" and the parameter "s1" is of type "String", the editor should show an error marker. When you use the content assist in line 6 after "SurfaceColorIsRed:equals(s1=", only the attribute "surfaceColor" should be offered. After choosing "surfaceColor", the error should disappear.

⁵ <http://download.eclipse.org/modeling/tmf/xtext/javadoc/2.3/org/eclipse/xtext/scoping/impl/AbstractDeclarativeScopeProvider.html>

Part B: Graphical Concrete Syntax

In the second part of this assignment, you have to develop a graphical concrete syntax and graphical editor for StateML state systems with *Sirius*. Like the textual concrete syntax, also the graphical concrete syntax has to be built upon the sample solution of the StatesML metamodel provided in the assignment resources.

Sirius Diagram Editor for StatesML State Systems

Develop a graphical concrete syntax and graphical (diagram) editor that allows to display and edit StatesML state systems. For this, develop *diagram mappings* and *element creation tools* as described in the following.

Mappings

For displaying StatesML state systems on diagrams, develop diagram node mappings and diagram edge mappings for the following StatesML concepts:

- State (distinguish between initial states, terminal states, and regular states)
- Transition (if a trigger is defined, the triggering event should be shown as part of a transition's label)
- Selection convergence
- Selection divergence
- Edge
- Function call

The diagram editor has to be able to display these elements following the *graphical concrete syntax* that is illustrated in Figure 2. Figure 2 shows the same example model as the one defined textually in Figure 1. The example model is available in the project `at.ac.tuwien.big.statesml.examples.sirius` in the model file `LineFollowingBehavior.statesml` (folder `simpleLineFollowing`).

The images for the mappings to be defined for the concepts Transition, Selection Convergence, and Selection Divergence are provided in `at.ac.tuwien.big.statesml.design/img`.

The used background and border colors are already defined in the provided viewpoint specification model `statesml.odesign` (located in `at.ac.tuwien.big.statesml.design/description`).

Creation Tools

For editing StatesML state systems, develop element creation tools for the following StatesML concepts (see also the tool palette shown in Figure 2):

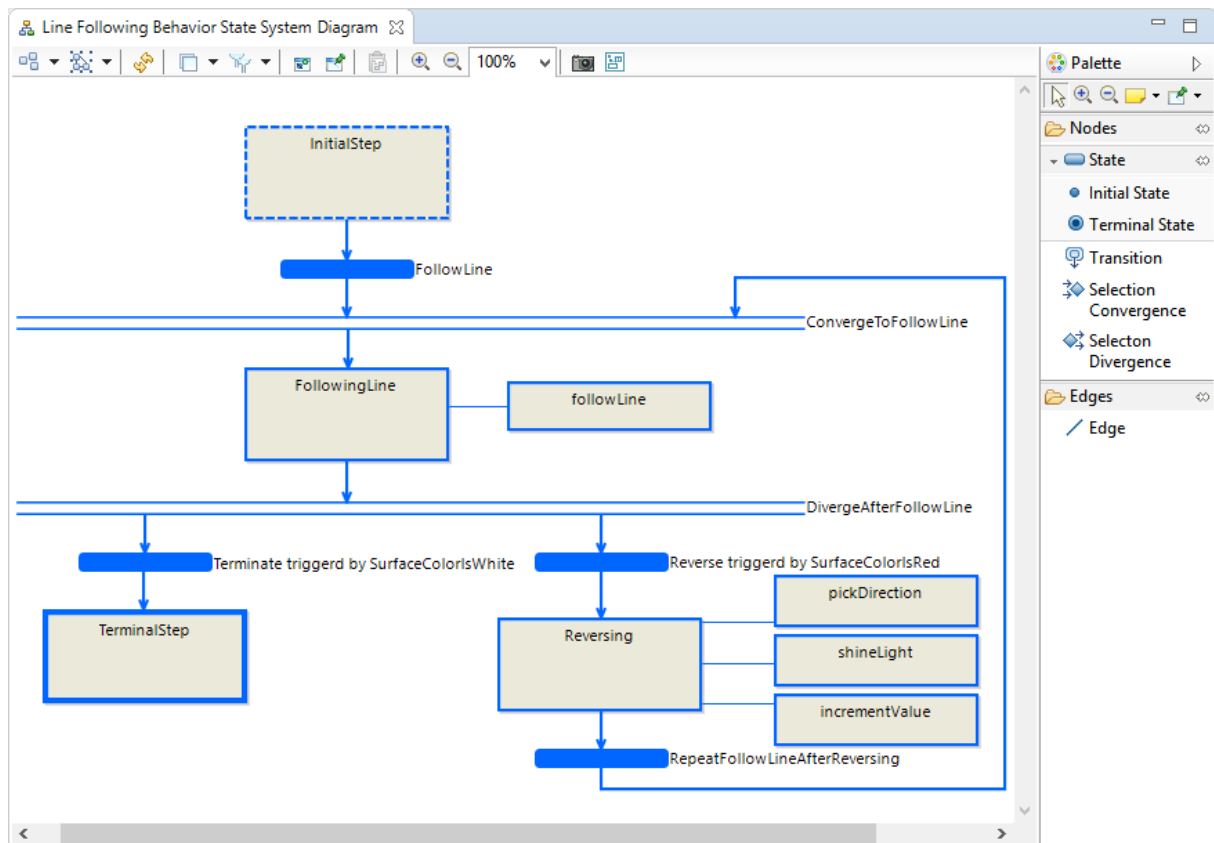
- State (distinguish between initial states, terminal states, and regular states)
- Transition
- Selection convergence
- Selection divergence
- Edge

You can reuse the icons for the element creation tools from the EMF UML plug-in `org.eclipse.uml2.uml.edit` using the following icon paths:

- Regular state: `/org.eclipse.uml2.uml.edit/icons/full/obj16/State.gif`
- Initial state: `/org.eclipse.uml2.uml.edit/icons/full/obj16/InitialNode.gif`

- Terminal state: /org.eclipse.uml2.uml.edit/icons/full/obj16/ActivityFinalNode.gif
- Transition: /org.eclipse.uml2.uml.edit/icons/full/obj16/Transition_external.gif
- Selection convergence: /org.eclipse.uml2.uml.edit/icons/full/obj16/MergeNode.gif
- Selection divergence: /org.eclipse.uml2.uml.edit/icons/full/obj16/DecisionNode.gif
- Edge: /org.eclipse.uml2.uml.edit/icons/full/obj16/Association.gif

Figure 2: Example StatesML state system in graphical concrete syntax



To implement the graphical concrete syntax and graphical (diagram) editor for StatesML, perform the following steps:

1. Setting up your workspace

Follow the instruction of Part A to setup your workspace and launch a new Eclipse instance. In summary, the following steps have to be performed for this:

- Your workspace has to contain the following projects (imported from the assignment resources):
 1. at.ac.tuwien.big.statesml
 2. at.ac.tuwien.big.statesml.edit
 3. at.ac.tuwien.big.statesml.editor
 4. at.ac.tuwien.big.statesml.statesystem
 5. at.ac.tuwien.big.statesml.statesystem.ide
 6. at.ac.tuwien.big.statesml.statesystem.ui
 7. at.ac.tuwien.big.statesml.xtext
 8. at.ac.tuwien.big.statesml.xtext.ui
- Launch a new Eclipse instance using the launch configuration "ME Lab 2 Runtime.launch" located in the project at.ac.tuwien.big.statesml.statesystem.

In the remainder of this document, the newly launched Eclipse instance is called “Runtime Eclipse Instance”.

Next, you have to import the following projects into the Runtime Eclipse Instance:

1. `at.ac.tuwien.big.statesml.design`
You will use this project, to develop the graphical concrete syntax and graphical editor of StatesML.
2. `at.ac.tuwien.big.statesml.examples.sirius`
You will use this project to test your graphical editor.

Lastly, switch into the *Sirius* perspective by selecting *Window* → *Perspective* → *Open Perspective* → *Other...* → *Sirius*. This perspective provides the *Model Explorer* and *Interpreter* views, which are useful for developing Sirius viewpoint specification models.

2. Developing your graphical editor

Define the graphical concrete syntax of StatesML with Sirius by extending the viewpoint specification model `statesml.odesign`, which is located in the project `at.ac.tuwien.big.statesml.design` in the folder `description`. Documentation about how to use Sirius can be found in the lecture slides and in the Sirius documentation⁶.

3. Testing your graphical editor

To test your graphical editor, open the diagram “Line Following Behavior State System Diagram” defined in the file `representations.aird` located in the project `at.ac.tuwien.big.statesml.examples.sirius` (you need to be in the *Sirius* perspective to be able to unfold the content of `representations.aird` in the *Model Explorer*). This diagram is mapped to the state system “LineFollowingBehavior” defined in the file `simpleLineFollowing/LineFollowingBehavior.statesml`. Thus, the diagram will be automatically updated with representations of model elements for which you have correctly defined node/edge mappings. Furthermore, the tool palette will show the element creation tools that you have correctly defined. Updates on your Sirius view point specification model `statesml.odesign` will be automatically reflected in the opened diagram.

For testing your element creation tools, you can use the diagram “Creation Tool Test State System Diagram” also defined in `representations.aird`. This diagram is mapped to the state system defined in the model `creationToolTest/CreationToolTestStateSystem.statesml`. If you add new elements to the diagram canvas by using the tool palette, this StatesML model has to be correctly updated. For instance, if you create a new initial state with the tool palette, a new state must be added to the model, which has the attribute `initial` set to `true` and the attribute `terminal` set to `false`.

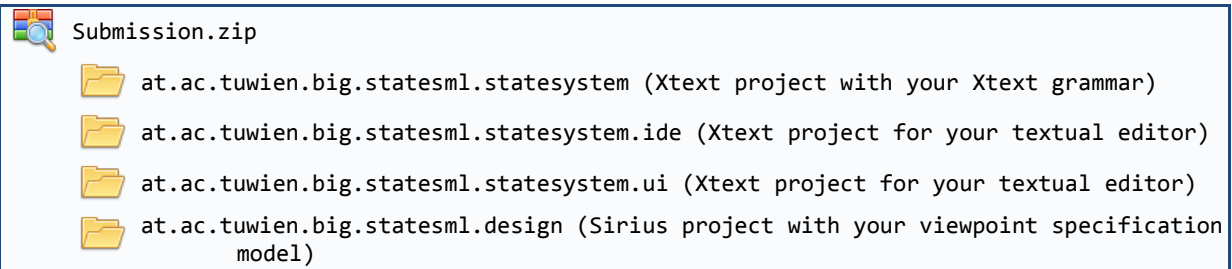
Hint: You can create additional StatesML models using the StatesML tree editor (*File* → *New* → *Other...* → *Example EMF Model Creation Wizards / StateML Model*). To create a diagram for a new StatesML model, expand the model in the *Model Explorer*, right-click on the State System element, select *New Representation* → *new State System Diagram*, enter a diagram name, and hit *OK*. Your graphical editor will be automatically opened for the example model.

⁶ Sirius documentation: <http://www.eclipse.org/sirius/doc/>
Sirius tutorials: <https://eclipse.org/sirius/getstarted.html>

Submission & Assignment Review

Upload the following components in TUWEL:

You have to upload one archive file, which contains the following projects:



For exporting these projects, select *File* → *Export* → *General/Archive File* and select the projects.

Make sure to include all resources that are necessary to start your editors!

Thus, if you make changes on any other project that was provided with the assignment resources or implement additional projects, make sure to include them in your submission.

At the assignment review you will have to present your solution, in particular, your Xtext grammar, the implemented scoping support, as well as your Sirius viewpoint specification model. You also have to show that you understand the theoretical concepts underlying the assignment.

All group members have to be present at the assignment review. The registration for the assignment review can be done in TUWEL. The assignment review consists of 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.