INF5120 - Assignment 2

Building a DSL Workbench for your selected Domain Specific Language – both as a Graphical DSL with Sirius and as a Lexical DSL with Xtext

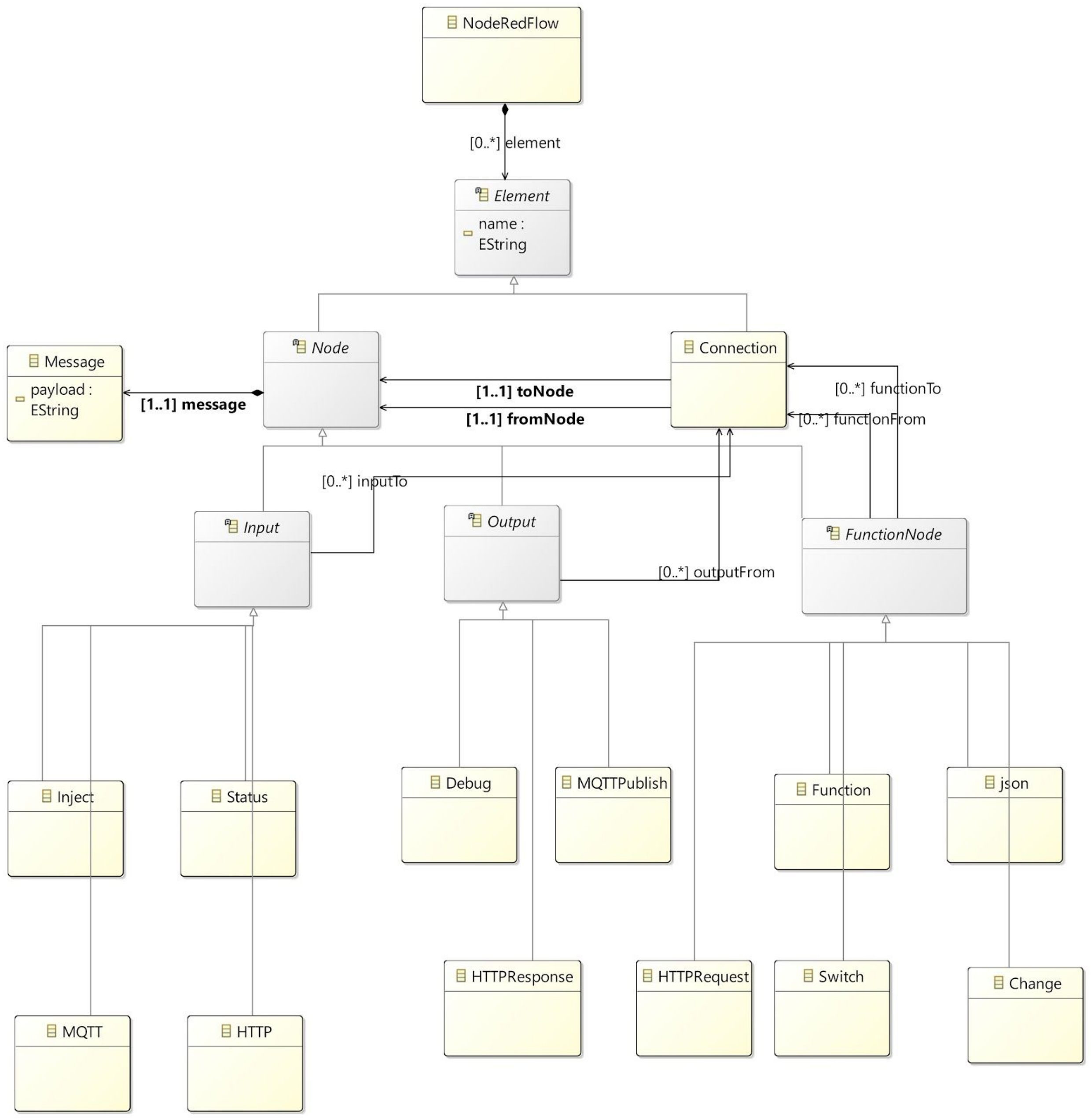
Adnan Alisa - adnanali@ifi.uio.no
An Lam - an.n.lam@hiof.no
Lucas Paruch - lucasp@ifi.uio.no
Tanusan Rajmohan - tanusanr@ifi.uio.no

University of Oslo

Spring 2019

Part 1

The Domain Specific Language (DSL) we have chosen to use for our graphical example is Node-RED. We started by creating a metamodel as shown below, as well as generating a graphical model and some examples as you can see in the following pages.



Clarification:

Metamodel:

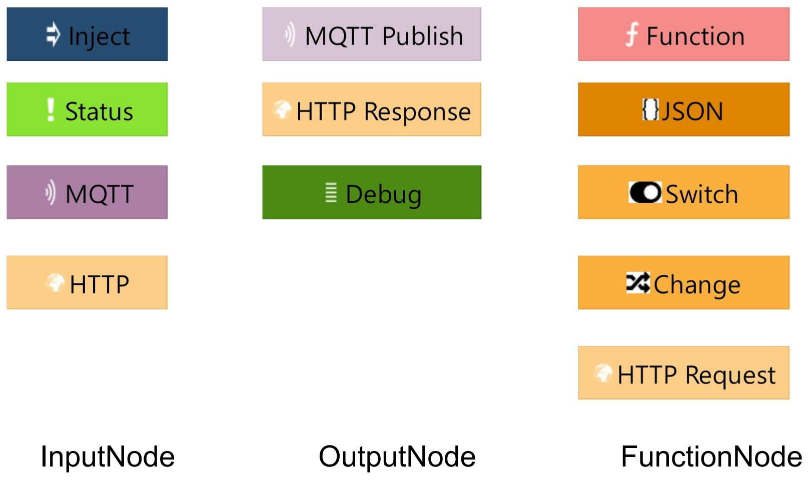
Node-RED consists of two types of fundamental elements: Node and connection. A flow is a set of Nodes and connections between the Nodes. A user can create a Node in Node-RED, but the Node must be a specific type of node (in our case either input-, output- or functionNode). Therefore we have made the Node an abstract class and divided its types into three subclasses. Furthermore, a user cannot create general input-, output-, and functionNodes, it must be of a specific kind. Therefore, we have also made these three subclasses abstract as well. When creating a Node instance it must be of a specific type and a kind e.g. HTTP Node. Each node needs to be classified with its own symbols and color.

Node:

We divided Node into three types:

1) **InputNode:** This type can only have connections going out from the node, and no connections coming in from the node (as with all inputNodes)
2) **OutputNode:** This type can only have connections coming in towards the node, and no connections coming out from it.
3) **FunctionNode:** This type can have an arbitrary number of both incoming and outgoing connections.

Instead of including all of the nodes from Node-Red into the meta-model, we have chosen only a subset of the nodes that represent the main principles of the domain-specific language.

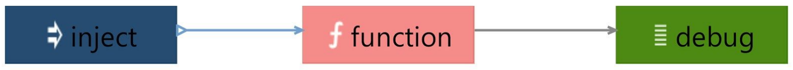


Connection:

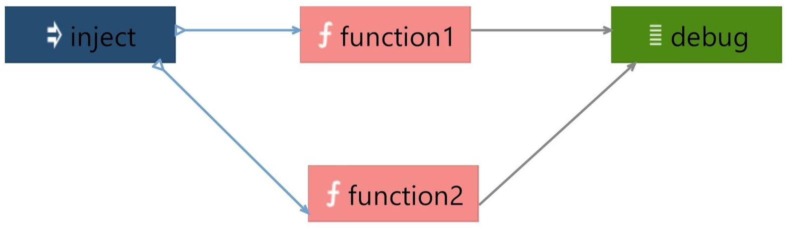
We have defined a connection as a relation between Nodes. Where a connection is a relation going from one Node to another Node. All the connections are 1-to-1 and not 1-to-many and many-to-1. However, a node may have an arbitrary number of connections (in and out). We have created an illustration based on the metamodel (see below).

Examples:

- A flow with three types of nodes:

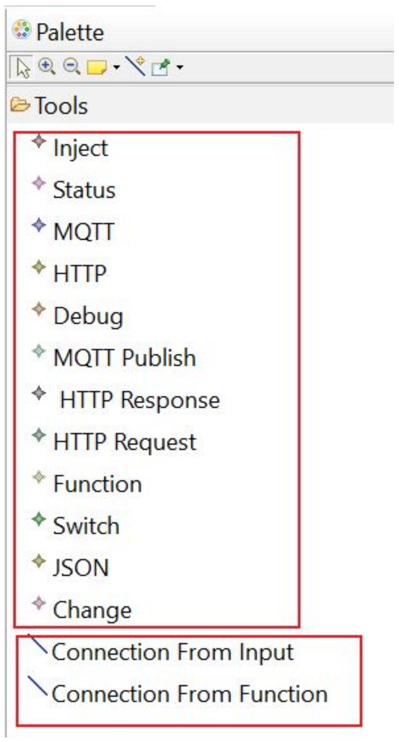


- One node can have multiple in and out connections:



Graphical palette

Here you can see the palette that was created for our graphical model. This tool supports the creation of all the type of Nodes that we have in the metamodel as well as the connection between the nodes.



Part 2

In this part, we still use Node-RED as our example of the DSL, and below you can see the Lexical Node-RED DSL with Xtext. We first created the grammar of the lexical language for Node-RED and then created examples to illustrate and show how our language works in accordance with the graphical models.

Grammar

The description of the elements in the grammar for the lexical language is similar to the graphical element descriptions. In addition, we have properties and function.

The functions, in this case, represent the operations to be performed on the input. They are corresponding to the subclass nodes in the graphical meta-model. Each Node in Node-RED has a set of properties. Looking at the Node-RED documentation we found that properties are passed to the node constructor when an instance of a Node is created[1].

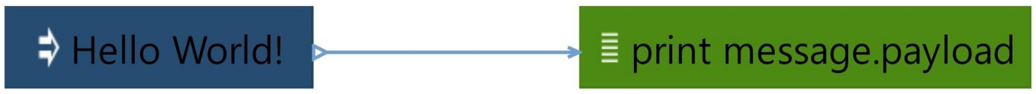
```
grammar org.xtext.example.mydsl.Oblig2 with org.eclipse.xtext.common.Terminals
generate domainmodel "http://www.xtext.org/example/mydsl/Oblig2"

Domainmodel:
	(element+=Type)*;
Type:
	 DataType | Flow | Function;
Function:
	'function' name=ID;
Property:
	(many?='many')? name=ID ':' type=[DataType];
InputNode:
	'inputNode' name=ID '{'
	'outputType' outputType=[DataType]
	function+=Function
	(properties+=Property)*
	'}';
OutputNode:
	'outputNode' name=ID '{'
	'inputType' inputType=[DataType]
	function+=Function
	(properties+=Property)*
	'}';
 FunctionNode:
```

[1] https://nodered.org/docs/creating-nodes/properties

```
        'functionNode' name=ID '{'
        'inputType' inputType=[DataType]
        'outputType' outputType=[DataType]
        function+=Function
        (properties+=Property)*
        '}';
DataType:
        'datatype' name=ID;
Node:
        InputNode | OutputNode | FunctionNode;
FromNode:
        InputNode | FunctionNode;
ToNode:
        OutputNode | FunctionNode;
Flow:
        'flow' name=ID '{'
        (nodes+=Node)*
        (connection+=Connection)*
        '}';
Connection:
        'connection' name=ID '{'
        'fromNode' fromNode=[FromNode]
        'toNode' toNode=[ToNode]
        '}';
```

Example 1: Two nodes and one connection

The graphical model:

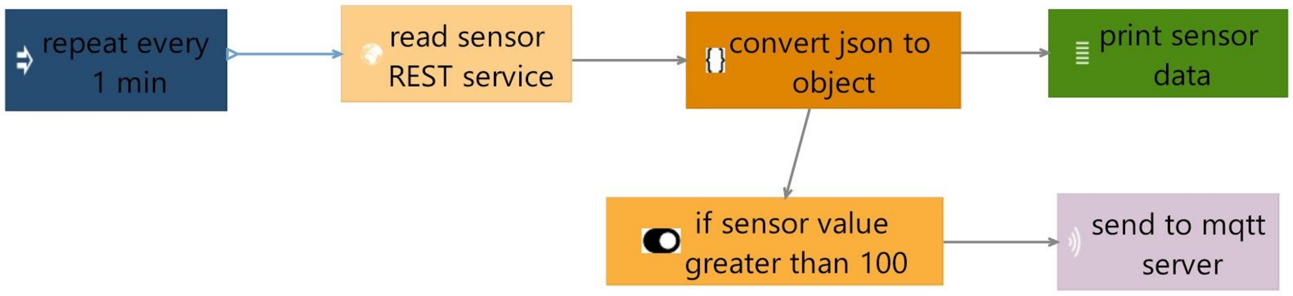


The corresponding lexical model:

```
datatype String

flow Oblig2 {
        inputNode HelloWorld {
                outputType String
                function Inject
                message: String
        }
        outputNode PrintMessagePayload {
                inputType String
                function Debug
        }

        connection connection1{
                fromNode HelloWorld
                toNode PrintMsg
        }
}
```

Example 2: More Complex

The graphical model:



The lexical model:

```
datatype String
datatype Interval
datatype Json
datatype SensorNode

flow Oblig2 {
        inputNode RepeatEvery1MinInput {
                outputType Interval
                function Inject
                interval: Interval
        }

        functionNode ReadSensorRESTService {
                inputType Interval
                outputType Json
                function HTTPRequest
        }

        functionNode ConvertJsonToObject {
```

```
		inputType Json
		outputType SensorNode
		function json
	}

	outputNode PrintSensorData {
		inputType SensorNode
		function Debug
	}

	functionNode IfSensorValueGreaterThan100 {
		inputType SensorNode
		outputType SensorNode
		function Switch
	}

	outputNode SendToMqttServer {
		inputType SensorNode
		function MQTTPublish
	}

	connection connection1 {
		fromNode RepeatEvery1MinInput
		toNode ReadSensorRESTService
	}

	connection connection2 {
		fromNode ReadSensorRESTService
		toNode ConvertJsonToObject
	}

	connection connection3 {
		fromNode ConvertJsonToObject
		toNode PrintSensorData
	}

	connection connection4 {
		fromNode ConvertJsonToObject
		toNode IfSensorValueGreaterThan100
	}

	connection connection5 {
		fromNode IfSensorValueGreaterThan100
		toNode SendToMqttServer
	}
}
```

Part 3

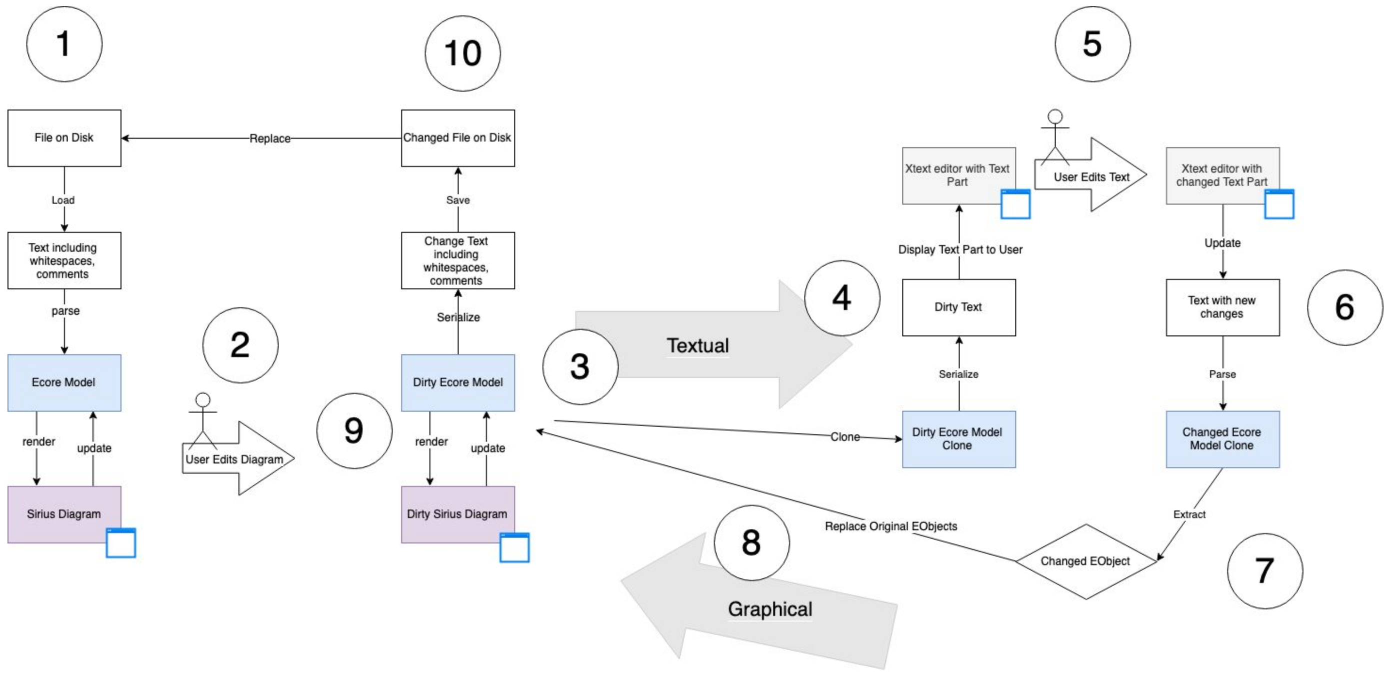
Model transformation:

Model transformation is the process of transforming an input model(s) (source model) to an output model(s) (target model). The transformation is based on the meta models of the input and output models, so the instance of the input meta model is transformed into the instance of the output model. Model transformation use mapping rules to describe what the elements in the input model should be transformed to in the output model, and when all elements have mapping rules the output model can be automatically generated from the input model.

Model traversal with Java API in EMF:
EMF makes it possible to generate Java API related to arbitrary metamodels. Mappings can be defined in a Java program by using the APIs. We can then run the Java program on a input model to transform it into a output model.

(See the next page for example and detailed step-by-step.)

Sirius and Xtext:



We will be presenting an example of going from graphical to lexical model and vice-versa, by using Sirius and Xtext. By using both frameworks it's possible to go back and forth, as illustrated in the figure above with a detailed step-by-step.[2, 3]

1. The user opens a Sirius diagram.
2. The user changes the diagram, e.g. they add some connections or remove some elements.
3. Sirius changes the underlying model without saving it. At this point, the contents of the file and the Ecore model differ.
4. Xtext serializes the dirty Ecore model clone and displays text to the user.
5. The user edits the Xtext file.
6. The changes from the user are updated and parsed into a Ecore model.
7. We compare the changes from the previous Ecore file to the changed Ecore file. We extract out the changed EObjects.
8. Changed EObjects are replaced with the EObjects in the "original" Ecore model.
9. Sirius renders the Ecore model with the updated EObjects to display the diagram.
10. The user saves the diagram

[2] https://www.obeodesigner.com/resource/white-paper/WhitePaper_XtextSirius_EN.pdf
[3] https://www.nikostotz.de/blog/xtext-editors-within-sirius-diagrams/