Group number: 5

Group name: Church

Names: Morten Egelund Rasmussen, Ole Ertmann Nielsen, Sten Anderson Rasmus Lohals Larsen

Emails: [stea@itu.dk](mailto:stea@itu.dk), [rlla@itu.dk](mailto:rlla@itu.dk)

# Table of Contents

1 Table of Contents 2

2 Example textual model 3

3 Meta-model 4

3.1 Class diagram 4

3.2 Partonomy 5

4 Static semantics 6

5 Xtext grammar 7

6 Backends 9

6.1 HTML 5 mobile web client 9

6.2 Windows Phone client 11

7 Test methods and artefacts 12

7.1 Test strategy 12

7.2 Metamodel test case examples 13

7.3 Grammar test case examples 14

7.4 Code generators test case examples 16

7.4.1 1. Overview of HTML5 mobile web client 16

7.4.2 Test of dynamic html generation 16

7.4.3 Test of static html generation 16

# Example textual model

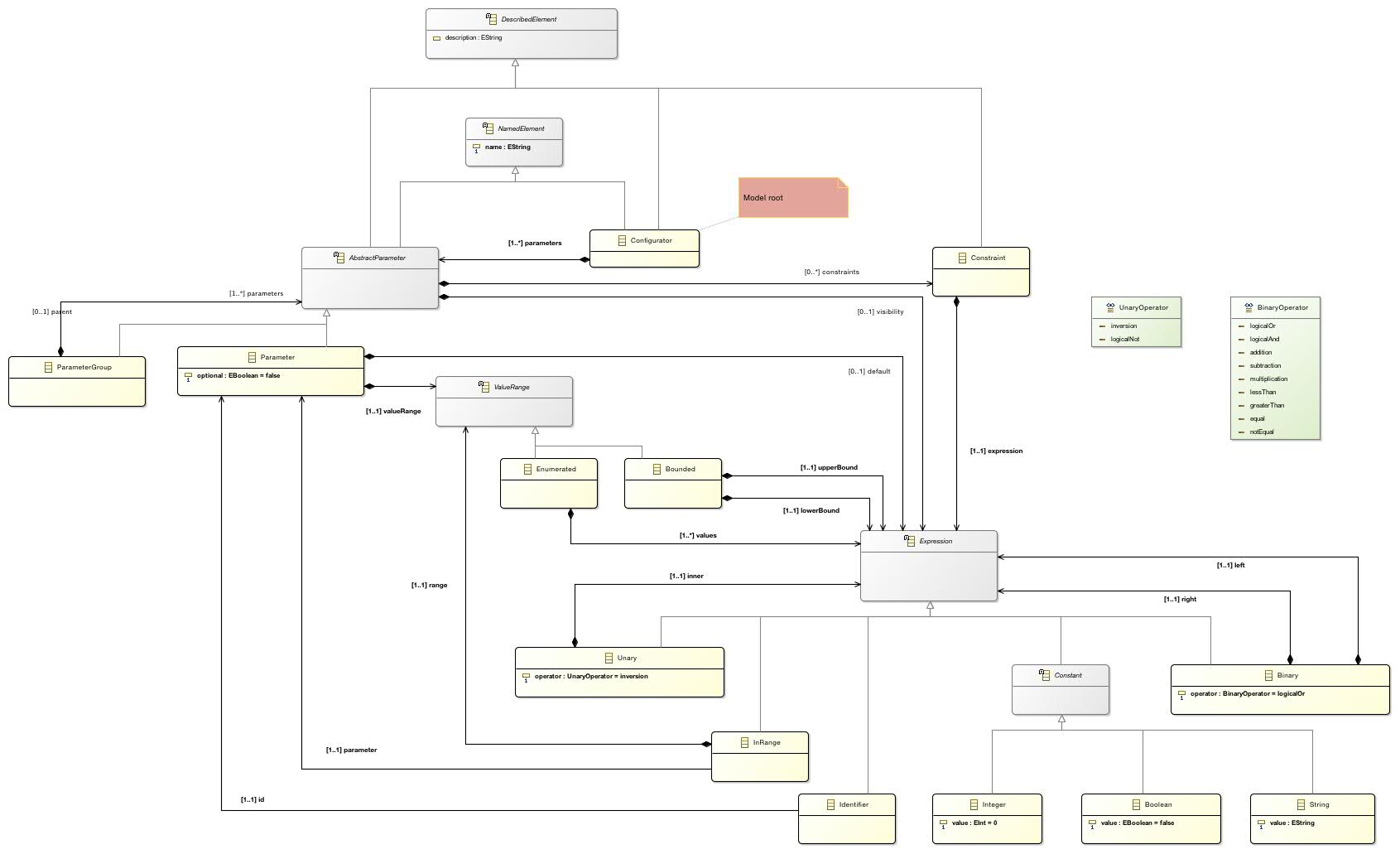
*Morten.*

*Tjek at vi kommer hele modellen rundt.*

No commentary expected

# Meta-model

## Class diagram



## Partonomy

*Sten*

# Static semantics

# Xtext grammar

**grammar** dk.itu.smdp2015.church.Configurator **with** org.eclipse.xtext.common.Terminals

**import** "http://itu.dk/smdp/configurator"

**import** "http://www.eclipse.org/emf/2002/Ecore" **as** ecore

Configurator:

'configurator' name=ID

(description=STRING)?

'{' parameters+=AbstractParameter ( ','? parameters+=AbstractParameter)\* '}';

AbstractParameter:

ParameterGroup | Parameter;

ParameterGroup:

'group' name=ID

(description=STRING)?

( ('visible-if' visibility=Expression)?

& ('constraints' '{' constraints+=Constraint ( ',' constraints+=Constraint)\* ','? '}' )?

& '{' parameters+=AbstractParameter ( ','? parameters+=AbstractParameter)\* ','? '}' );

Parameter:

'parameter' name=ID

(description=STRING)?

( ((optional?='optional')|'mandatory')?

& ('visible-if' visibility=Expression)?

& ('default-value' default=Expression)?

& ('constraints' '{' constraints+=Constraint ( ',' constraints+=Constraint)\* '}' )?

& 'values' valueRange=ValueRange );

ValueRange:

Enumerated | Bounded;

Enumerated **returns** *Enumerated*:

'(' values+=Expression ( ',' values+=Expression)\* ')';

Bounded **returns** *Bounded*:

'[' lowerBound=Expression ';' upperBound=Expression ']';

Constraint:

('description' description=STRING)?

expression=Expression;

Expression:

LogicalOr;

**enum** LogicalOrOperator **returns** *BinaryOperator*:

logicalOr = 'or' ;

LogicalOr **returns** *Expression*:

LogicalAnd ( {*Binary*.left=**current**} operator=LogicalOrOperator right=LogicalAnd )\*;

**enum** LogicalAndOperator **returns** *BinaryOperator*:

logicalAnd = 'and';

LogicalAnd **returns** *Expression*:

Equality ( {*Binary*.left=**current**} operator=LogicalAndOperator right=Equality )\*;

**enum** EqualityOperator **returns** *BinaryOperator*:

equal = '==' | notEqual = '!=';

Equality **returns** *Expression*:

Comparative ( {*Binary*.left=**current**} operator=EqualityOperator right=Comparative )\*;

**enum** ComparativeOperator **returns** *BinaryOperator*:

lessThan = '<' | greaterThan = '>' ;

Comparative **returns** *Expression*:

Additive ( {*Binary*.left=**current**} operator=ComparativeOperator right=Additive )\*;

**enum** AdditiveOperator **returns** *BinaryOperator*:

addition = '+' | subtraction = '-';

Additive **returns** *Expression*:

Multiplicative ( {*Binary*.left=**current**} operator=AdditiveOperator right=Multiplicative )\*;

**enum** MultiplicativeOperator **returns** *BinaryOperator*:

multiplication = '\*';

Multiplicative **returns** *Expression*:

Primitive ( {*Binary*.left=**current**} operator=MultiplicativeOperator right=Primitive )\*;

Primitive **returns** *Expression*:

Unary | InRange | Integer | Boolean | String0 | Identifier | '(' Expression ')';

**enum** UnaryOperator:

inversion = '-' | logicalNot = 'not';

Unary:

operator=UnaryOperator inner=Primitive;

Constant:

Integer | Boolean | String0;

InRange:

parameter=[*Parameter*] 'in' range=ValueRange;

Integer:

value=EInt;

Boolean:

value=EBoolean;

String0 **returns** *String*:

value=STRING;

Identifier:

id=[*Parameter*];

EInt **returns** *ecore::EInt*:

/\* '-'? \*/ INT;

EDouble **returns** *ecore::EDouble*:

/\* '-'? \*/ INT? '.' INT (('E'|'e') '-'? INT)?;

EBoolean **returns** *ecore::EBoolean*:

'true' | 'false';

# Backends

This section attempts to briefly give an understanding of the backends implemented.

The both walkthroughs is based on the same DSL sample:

|  |
| --- |
| DSL input sample |
| **configurator** car "A configurator for a car." {  **parameter** engine **values** ('TFSI 1.20', 'TFSI 1.4', 'TFSI 2.02') **constraints** { **description** "Big engines only available for sports model" variant == 'sport' **or** engine **in** ('TFSI 1.20', 'TFSI 1.4')},  **parameter** variant **values**('standard', 'sport','luxury') ,  **group** seats **visible-if** variant **in** ('standard') {**parameter** someSeatParam **values**("leather","cloth")}  } |

## HTML 5 mobile web client

The HTML client is build using HTML5, javascript and CSS. The code generated is purely html and javascript, so no compilation is taking place as these scripts are interpreted by a browser. We have used two popular javacript frameworks Jquery Mobile (JQM) and Knockout to build a single page web application, with a clearly defined user interface architecture. Jquery mobile enables mobile oriented user experiences using a simple declarative markup. Depending on the markup the framework applies javascript and CSS to give the application a native mobile look and feel. Knockout is a two way data binding javascript framework that uses the Model-View-ViewModel (MVVM) user interface architectual pattern to facilitate a clear separation of concerns between user interface logic and data model manipulation. This separation made it fairly straight forward to generate code from an instance of our Meta Model. Using the Knockout validation plugin, converting our validation expressions into javascript code was also straight forward, as this plugin enable custom validation rules, which is automatically applied by the framework.

The enduser can change configuration values, and validation constraints are applied as vales changes. When the user is finished configuring a ”submit” button serializes the model with current value state to JSON and thi scan then be save to a file, email or some service for consumption or further transformation.

|  |
| --- |
| Figure showing overview of html client architecture |
|  |

|  |
| --- |
| Sample template method for the HTML, which show the generation of the actual input element of a parameter with an enumerated ValueRange |
| **def** **dispatch** renderRangeInputElement(Enumerated **it**, Parameter param){  '''«**IF** rangeType == ExpressionType.*Boolean*»  «renderBooleanInputElement(param)»  «**ELSE**»  <select id="«param.name»-param" data-bind="options: «param.name».choices, selectedOptions: «param.name».value,optionsCaption:'Choose...'"></select>  «**ENDIF**»  '''  } |

|  |
| --- |
| Jquery mobile page sample showing the first page in the application. There |
| <**div id="main" data-role="page" data-add-back-btn="true"**>  <**div data-role="header"**>  <**h1**>   </**h1**>  <**button class="ui-btn-right ui-icon-check ui-btn-icon-right ui-btn" onclick="***submitconfiguration*();**"**>submit</**button**>  </**div**>  <**div role="main"**>  <**section class="description"**>  A configurator for a car  </**section**>  <**section class="validationSection" data-bind="css:{showValidationSummary: !isModelValid()}"**>  <**div class="validationSummary"**>  <**h4**>Validation summary</**h4**>  <**ul data-bind="foreach: currentErrors"**>  <**li**><**span data-bind="text: $data"**> </**span**> </**li**>  </**ul**>  </**div**>  </**section**>  <**ul data-role="listview"**>  <**li**>  <**label for="engine-param"**>engine:</**label**>   <**select id="engine-param" data-bind="options: engine.choices, selectedOptions: engine.value,optionsCaption:'Choose...'"**>      </**select**>  <**p class="validationMessage" data-bind="validationMessage: engine.value"**></**p**>  **[… *Cut out for brevity …*]**  <**li data-bind="visible: group\_seats().isVisible"**>  <**a href="#seats"**>seats  <**p class="validationMessage" data-bind="validationMessage: group\_seats"**></**p**></**a**>  </**li**>  </**ul**>  </**div**> </**div**> |

|  |
| --- |
| Knockout ViewModel-Model object sample.  Some elements are cut out for brevity. It shows the complete ’engine’ parameter data model, and start of nested group ’seats’. |
| ***App***.**ViewModel** = **ko**.validatedObservable({  **currentErrors**:**ko**.observableArray([]),  **isModelValid**: **ko**.deferredPureComputed(**function**()  {  **var** isValid= ***App***.**ViewModel**.**isValid**();  ***App***.**ViewModel**().**currentErrors**(***App***.**ViewModel**.**errors**());  **return** isValid;  },**this**),  **[… *Cut out for brevity …*]**  **engine**:  {  **choices**: [**'TFSI 1.2'**, **'TFSI 1.4'**, **'TFSI 2.02'**],  **value**: **ko**.observable()  .**extend**({  **validation**: {  validator: **function** (val, param) {  **if**(***App***.**ViewModel**==**null**)*//not initialized* **return true**;  *//Expression here:* **var** result =  (  **$**.inArray(**"TFSI 1.2"**, ***App***.**ViewModel**().**engine**.**value**()) > -1 ||  **$**.inArray(**"TFSI 1.4"**, ***App***.**ViewModel**().**engine**.**value**()) > -1  ) ||  **$**.inArray(**"sport"**, ***App***.**ViewModel**().**variant**.**value**()) > -1  **return** result;  },  **message**: **"Big engines only available for sports model"** }})  }  **group\_seats**: **ko**.observable({  **isVisible**:**ko**.deferredPureComputed(**function**(){  *//Example of evaluating value where value is pointing to enumeration where value will be in an array (select input element)* **return $**.inArray(**"standard"**, ***App***.**ViewModel**().**variant**.**value**()) > -1;  });  **[… *Cut out for brevity …*]**  })  }); |

|  |
| --- |
| Sample output of the configurator application.  This is the serialized data model in JSON format. Notice that only the selected values and global state of the configuration model is included in the output. |
| {  "currentErrors": [  "Big engines only available for sports model"  ],  "isModelValid": false,  "engine": {  "value": [  "TFSI 2.02"  ]  },  "variant": {  "value": [  "luxury"  ]  }  } |

## Windows Phone client

(Ole)

# Test methods and artefacts

## Test strategy

We have written tests covering the following parts of our project:

* Meta model: Tested through dynamic model instances.
* Parser: Testing grammar syntax.
* Constraints: Testing syntax that satisfies/violates the constraint in question.
* Code generators: Testing that different elements returns expected generated code.

We have written unit-tests for each part, which are all based on a known initial state / input (i.e. a test bench with a fixed input), and a confirmation that the tested element returns the expected output.

We have written unit tests which validates valid input, or (correctly) invalidates invalid input. Thus, we have both positive and negative test cases.

We have focused on making each unit test as small as possible, in order to give a detailed overview of the test results. This gives a clear indication for any possible test errors.

We are aware that unit tests cannot stand alone as a full test of the developed feature. An easy way extend the system tests would be to perform a compilation of the generated code (if the generated code needs to be compiled), subsequently performing an exploratory test of the final application which the user sees.

## Metamodel test case examples

**class** TestConstraints {

**def** **static** **dispatch** constraint(Configurator **it**) {

!parameters.empty && !name.empty

}

**def** **static** **dispatch** constraint(Parameter **it**) {

!name.empty

}

**def** **static** **dispatch** constraint(ParameterGroup **it**) {

!parameters.empty && !name.empty

}

**def** **static** **dispatch** constraint(Bounded **it**) {

**var** lBound = (lowerBound **as** dk.itu.smdp2015.church.model.configurator.Integer)

**var** uBound = (upperBound **as** dk.itu.smdp2015.church.model.configurator.Integer)

lBound.value < uBound.value

}

....

// Fallback

**def** **static** **dispatch** constraint(EObject **it**) {

**true**

}

}

## Grammar test case examples

**package** dk.itu.smdp2015.church.configurator.syntax.tests

// Imports removed

@RunWith(XtextRunner)

@InjectWith(ConfiguratorInjectorProvider)

**class** ConfiguratorGrammarTest {

@Inject **extension** ParseHelper<Configurator>

@Inject **extension** ValidationTestHelper

@Before

**def** **void** before() {

ConfiguratorPackage.*eINSTANCE*.eClass

}

@Test

**def** **void** testInvalidModelNoParameters() {

**var** model = '''configurator Empty'''.parse

model.assertError(ConfiguratorPackage.Literals.*CONFIGURATOR*, Diagnostic.*SYNTAX\_DIAGNOSTIC*, "mismatched input")

}

@Test

**def** **void** testValidBoundedRange() {

**var** model = '''configurator Bicycle "Bicycle configuration" { parameter wheel\_size values [16;24] }'''.parse

*assertEquals*("Bicycle configuration", model.description)

**var** param = model.parameters.get(0) **as** Parameter

*assertEquals*("wheel\_size", param.name)

**var** valueRange = param.valueRange **as** Bounded

*assertEquals*(16, (valueRange.lowerBound **as** IntegerImpl).value)

*assertEquals*(24, (valueRange.upperBound **as** IntegerImpl).value)

model.assertNoErrors

}

@Test

**def** **void** testInvalidBoundedRangeReverse() {

**var** model = '''configurator Bicycle { parameter wheel\_size values [24;16] }'''.parse

model.assertError(ConfiguratorPackage.Literals.*BOUNDED*, ConfiguratorValidator.*INVALID\_BOUND*, "Lower bound should be less than upper bound")

}

@Test

**def** **void** testInvalidBoundedRangeWrongTypes() {

**var** model = '''configurator Bicycle { parameter wheel\_size values [b;16] }'''.parse

model.assertError(ConfiguratorPackage.Literals.*BOUNDED*, ConfiguratorValidator.*WRONG\_TYPE*, "expected the same type")

}

@Test

**def** **void** testValidEnum() {

**var** model = '''configurator Car { parameter Variant values ("Standard", "Sport", "Luxury") }'''.parse

*assertNull*(model.description)

**var** param = model.parameters.get(0) **as** Parameter

*assertEquals*("Variant", param.name)

*assertFalse*(param.optional)

**var** enumerated = param.valueRange **as** Enumerated

*assertEquals*(3, enumerated.values.*length*)

*assertEquals*("Standard", (enumerated.values.get(0) **as** StringImpl).value)

*assertEquals*("Sport", (enumerated.values.get(1) **as** StringImpl).value)

*assertEquals*("Luxury", (enumerated.values.get(2) **as** StringImpl).value)

model.assertNoErrors

}

@Test

**def** **void** testInvalidEnumWrongElement() {

**var** model = '''configurator Car { parameter Variant values ("Standard", "Sport", Luxury) }'''.parse

model.assertError(ConfiguratorPackage.Literals.*ENUMERATED*, ConfiguratorValidator.*INVALID\_ENUMERATION*, "Enumerated item should be a constant")

}

// .....

}

## Code generators test case examples

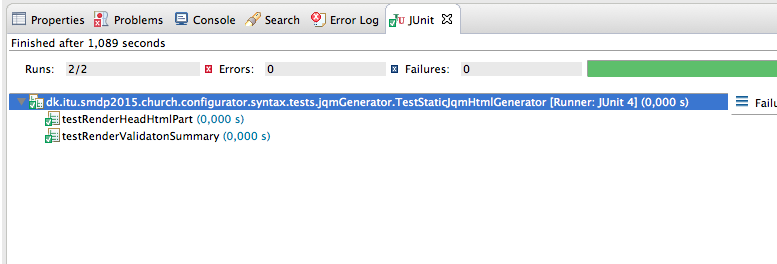
### 1. Overview of HTML5 mobile web client

Here we only show the jUnit tests for the html generator part. The test case are divided into a dymanic html generator and a static html generator.

First we show an overview of the passing tests as the present themselves in the Eclipse IDE.

### Test of dynamic html generation

### Test of static html generation



Then an explanation of how the testcode is built up.

|  |
| --- |
| Sample code showing some sample dynamic html test case. We make heavily use of Xtends ability to do chained method calls, increasing readability of the code.  All tests are build using the same pattern:  1. Arrange part  An input DSL string, just containing the essential part for test case at hand.  This test input are processed by adding boilerplate prefix (‘configuration someConfiguratorName’), and then parsed into an instance of our metamodel.  Then we select the part of the metamodel instance that are relevant for the test case (the parameter object in this case)  2. Act part  Here we call the .compileParameterLink which is the method being tested here.  3. Assert part  We test the returned string with an expected, string. Since it is html we cant compile it. The test case asserts not only that the semantics of the html is correct, but also that the generated is human readable and indented correctly. |
| @Test  **def** **void** testParameterLink\_RendersAsListItem(){    '{parameter test values (0;10)}'.addPrefix.parse.firstParam  .compileParameterLink  .assertHtmlWithExpectedOutput(  '''<li>  <label for="test-param" >test:</label>  <select id="test-param" data-bind="options: test.choices, selectedOptions: test.value,optionsCaption:'Choose...'"></select>  <p class="validationMessage" data-bind="validationMessage: test.value"></p>  </li>  ''')    } |

|  |
| --- |
| All boiler plate code are put in an abstract base class, and the the actual test class contains only a reference to class under test, and the test cases. |
| **class** TestDynamicJqmHtmlGenerator **extends** BaseTestJqmGenerator{  @Inject **extension** JqmHtmlGenerator //Sut    @Test  **def** **void** testConfigDescription\_rendersAsMainSection(){    'configurator app "main app description"{}'.parse  .renderAppDescription  .assertHtmlWithExpectedOutput(  '''<section class="main-description">  main app description  </section>  ''')    } |