

SUBMISSION OF WRITTEN WORK

Class code:

Name of course:

Course manager:

Course e-portfolio:

Thesis or project title:

Supervisor:

Full Name:

Birthdate (dd/mm-yyyy):

E-mail:

- | | | |
|----------|-------|--------------|
| 1. _____ | _____ | _____@itu.dk |
| 2. _____ | _____ | _____@itu.dk |
| 3. _____ | _____ | _____@itu.dk |
| 4. _____ | _____ | _____@itu.dk |
| 5. _____ | _____ | _____@itu.dk |
| 6. _____ | _____ | _____@itu.dk |
| 7. _____ | _____ | _____@itu.dk |

Model Driven Development Project

A Configurator Project

Jorgensen, Anders B.
abrj

Helvind, Jakob B.
jbah

Hartvig, Martin R.
mrha

Kjerri, Rune M.
rmkj

May 18, 2015

1 THE TEXTUAL MODEL

Listing 1: Concrete syntax of a CarFactory

```
carFactory {
2   BMW {
      has
4     carType ["SUV", "Pickup", "Minivan", "Supercar", "Sedan", "Stationcar",
              "Microcar"],
        enginType ["Diesel", "Gas", "Electric", "Hybrid"],
6     doors ["5-Door", "4-Door", "2-Door"],
        seatType ["Racing Seats", "Standard Seats", "Hardwood Seats"],
8     seatHeat ["Seat Heat": "No Seat Heat"],
        wheel [14-28],
10    numberOfSeat [2,4,5,7,8],
        color
            ["DarkSalmon", "FloralWhite", "MidnightBlue", "OliveDrab", "RosyBrown", "LemonChiffon",
             "DimGray"]
12
14
16    Constrained by
        if carType = "Supercar"
18    then (seatHeat can "Seat Heat") && (doors can "2-Door")

20    if seatHeat = "Seat Heat"
        then seatType can "Standard Seats"
22
24    if (wheel > 24) && (carType = ("SUV" | "Pickup"))
        then enginType can "Diesel", "Gas"

26    if enginType = ("Electric" | "Hybrid")
        then carType can "Sedan", "Stationcar", "Microcar"
28
30    if doors = "5-Door"
        then carType can "SUV", "Minivan", "Sedan", "Stationcar"

32    if (seatType = "Hardwood Seats") && (color = "LemonChiffon")
        then (enginType can "Gas") && (seatHeat can "No Seat Heat")
34
36    if numberOfSeat > 5
        then carType can "Stationcar", "Minivan"

38    if color = "DarkSalmon"
        then carType can "Minivan", "Microcar"
40  }
}
```

2 THE VIEWS OF THE META-MODEL

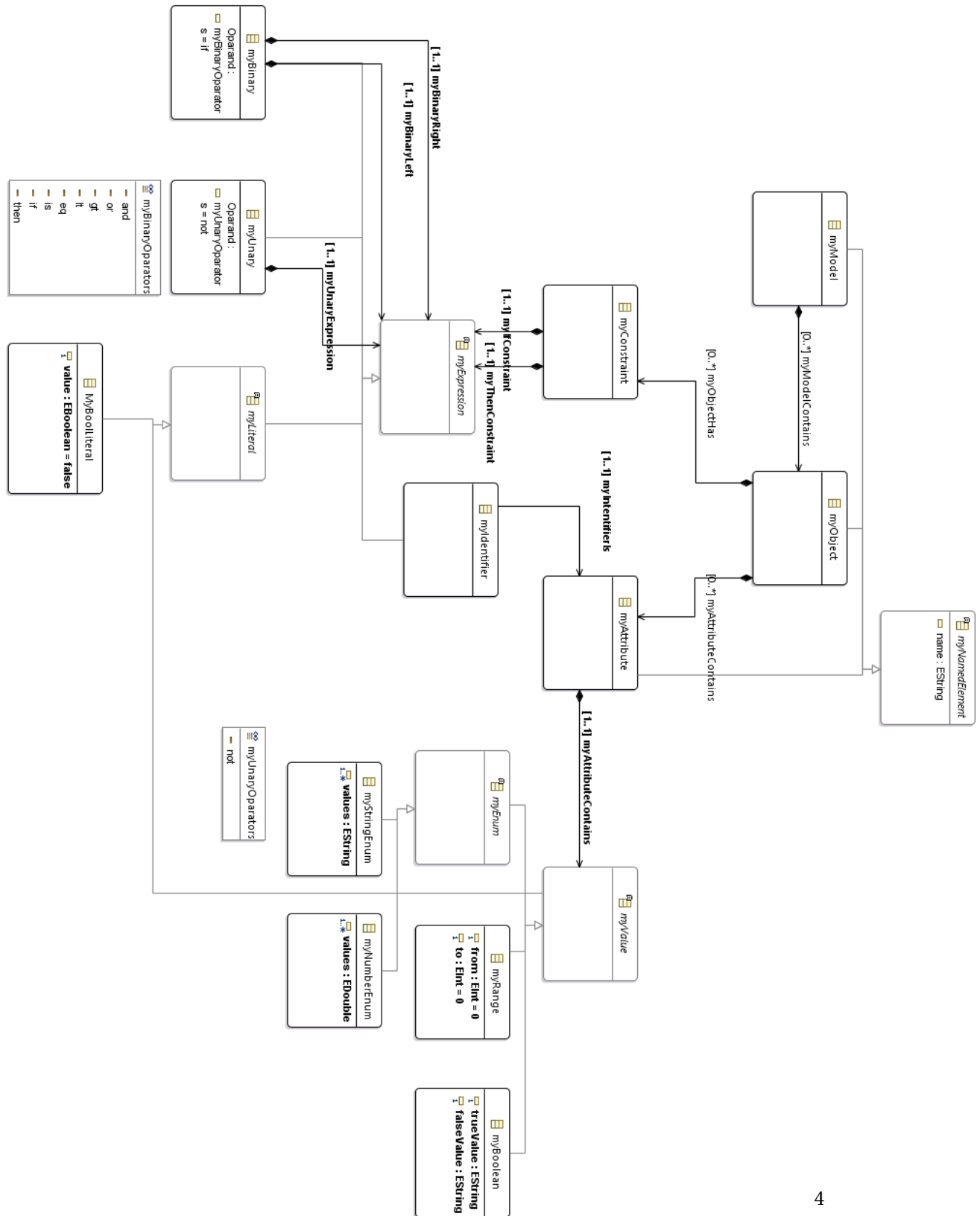


Figure 2.1: View of the whole meta-model

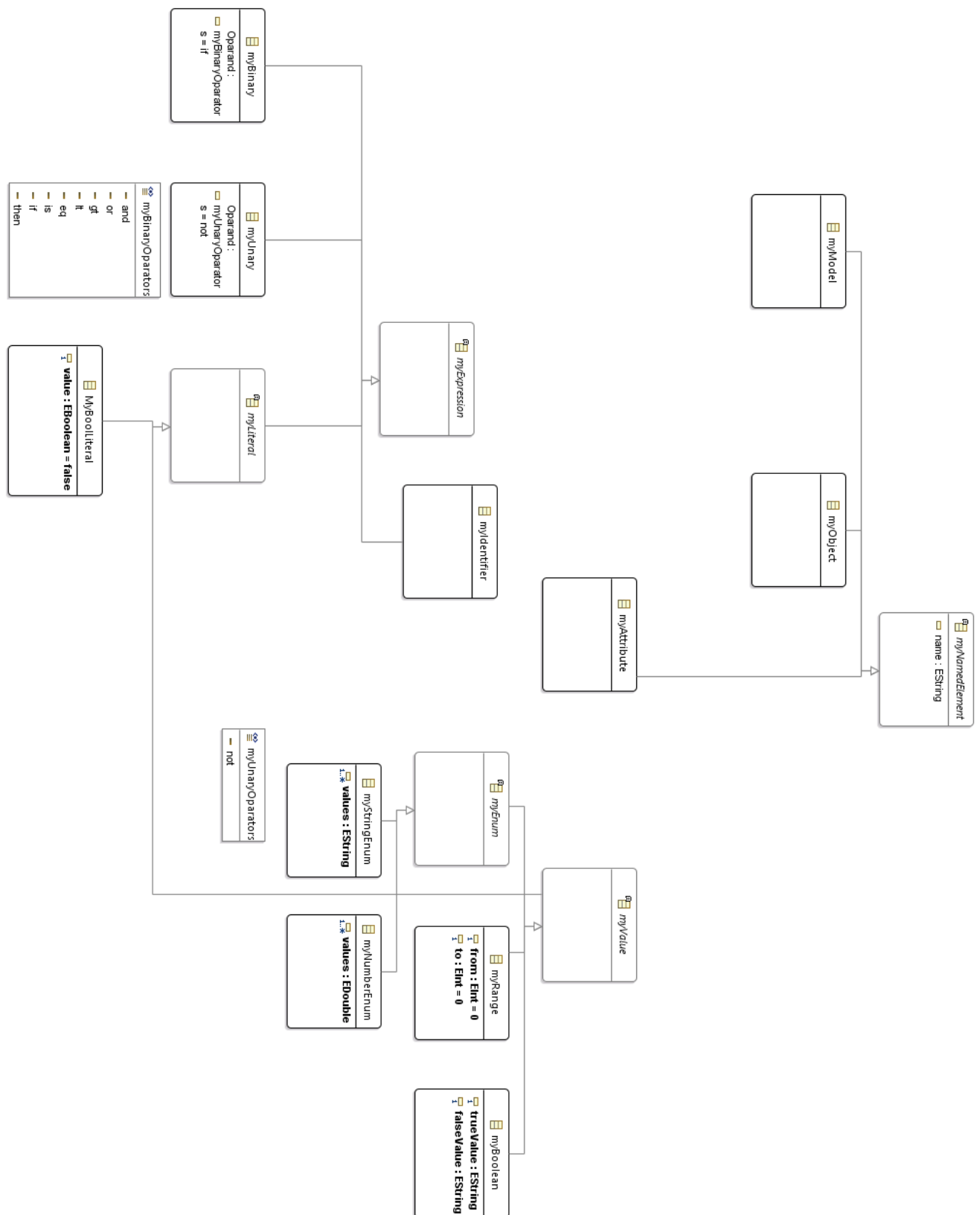


Figure 2.2: View of the taxonomy

3 THE STATIC SEMANTICS

Listing 2: Static Constraints

```
class SmdpDslValidator extends AbstractSmdpDslValidator {
2  @Check
  def boolean constraint(myObject it) {
4    try {
      // Attribute name must be unique
6    if (!(it.myAttributeContains.forall[ attributeName |
      myAttributeContains.filter[name.equalsIgnoreCase(attributeName.name)].size
      == 1])){
      error("A attribute should have a unique name", null);
8    }
    return true;
10   } catch (Exception e) {
    e.printStackTrace();
12   return false;
  }
14 }

16 @Check
  def boolean constraint(myConstraint it){
18   // Checks the then part of
    try {
20     if (!myValuesCheck(myThenConstraint as myBinary, null)) {
      error("Constraint contains illigal values", myThenConstraint, null)
22     }

24     if (!myValuesCheck(myIfConstraint as myBinary, null)) {
      error("If statements contain a invalid value", myIfConstraint, null);
26     }
    return true;
28   } catch (Exception e) {
    e.printStackTrace();
30   return false;
  }
32 }

34 @Check
  def boolean constraint(myNumberEnum it) {
36   try {
38     if (values.length == 0) {
      error("All number enum must have a size of at least 1", it, null)
40     }
    return true;
42   } catch (Exception e) {
    e.printStackTrace();
  }
```



```

44     return false;
45     }
46 }
@Check
48 def boolean constraint(myRange it) {
49     try {
50         if (from > to) {
51             error("The start value in a range cannot be larger than the end value", it,
52                 null);
53         }
54         return true;
55     } catch (Exception e) {
56         e.printStackTrace();
57         return false;
58     }
59 }
@Check
60 def boolean constraint(myBoolean it) {
61     try {
62         if (trueValue == falseValue) {
63             error("The values for boolean can't be the same", it, null);
64         }
65
66         if (trueValue == "" || falseValue == ""){
67             error("Boolean must be assigned a value",it, null);
68         }
69
70         return true;
71     } catch (Exception e) {
72         e.printStackTrace();
73         return false;
74     }
75 }
@Check
76 def boolean constraint (myStringEnum it) {
77     try {
78         if (values.length == 0 || values.exists[v | v.equalsIgnoreCase(")]) {
79             error("String enum must contain a value",it, null)
80         }
81         return true
82     } catch (Exception e) {
83         e.printStackTrace();
84         return false;
85     }
86 }
87
88
89
90 /* Helper method to go throuh the expression tree, to check if all values
    are valid

```

```

    * We don't take the operand into account, there for it will be possible to
      set values that are outside a range scope
92  * if '<' or '>' is used.
    */
94  def boolean myValuesCheck(myBinary it, myIdentifier attribute){
      var boolean leftCorrect = false;
96      var boolean rightCorrect = false;
      var myIdentifier att;
98      // If left is a identifier, get the attribute
      if (it.myBinaryLeft instanceof myIdentifier) {
100         att = it.myBinaryLeft as myIdentifier;
         leftCorrect = true;
102     } else {
         att = attribute;
104     }

106     // If both left and right are binaries, then both sides must be true
      if (it.myBinaryLeft instanceof myBinary && it.myBinaryRight instanceof
          myBinary) {
108         return myValuesCheck(it.myBinaryLeft as myBinary, att) &&
            myValuesCheck(it.myBinaryRight as myBinary, att)
      }

110     // If the right is a binary, then go one depth deeper.
112     if (it.myBinaryRight instanceof myBinary){
        return myValuesCheck(it.myBinaryRight as myBinary, att)
114     }

    //region Check the values in the left side
116     if (it.myBinaryLeft instanceof myStringEnum) {
        // If the left is a string, then the value must either be a StringEnum or
          a Boolean
118         val attributeValue = att.myIdentifierIs.myAttributeContains;

120         if (attributeValue instanceof myStringEnum) {
            leftCorrect = myStringEnumValueCheck(attributeValue as myStringEnum,
                myBinaryLeft as myStringEnum)
122         }

124         if (attributeValue instanceof myBoolean){
            leftCorrect = myBooleanValueCheck(attributeValue as myBoolean,
                myBinaryLeft as myStringEnum)
126         }

128         if (it.operand != myBinaryOperators.OR) {
            error("Operand cannot be other than '|'",it,null)
130             leftCorrect = false;
        }
132     }

```

```

134     if (it.myBinaryLeft instanceof myNumberEnum) {
135         val attributeValue = att.myIntenfifierIs.myAttributeContains;
136         // If the left is a number, then the value must either be a NumberEnum or
            a Range
137         if (attributeValue instanceof myNumberEnum){
138             leftCorrect = myNumberEnumValueCheck(attributeValue as myNumberEnum,
                myBinaryLeft as myNumberEnum)
139         }
140
141         if (attributeValue instanceof myRange){
142             leftCorrect = myRangeValueCheck(attributeValue as myRange, myBinaryLeft
                as myNumberEnum)
143         }
144     }
145     //endregion
146
147     //region Check the values in the right side
148     if (it.myBinaryRight instanceof myStringEnum) {
149         val attributeValue = att.myIntenfifierIs.myAttributeContains;
150
151         if (attributeValue instanceof myStringEnum) {
152             rightCorrect = myStringEnumValueCheck(attributeValue as myStringEnum,
                myBinaryRight as myStringEnum)
153         }
154
155         if (attributeValue instanceof myBoolean){
156             rightCorrect = myBooleanValueCheck(attributeValue as myBoolean,
                myBinaryRight as myStringEnum)
157         }
158     }
159
160     if (it.myBinaryRight instanceof myNumberEnum) {
161         val attributeValue = att.myIntenfifierIs.myAttributeContains;
162         if (attributeValue instanceof myNumberEnum){
163             rightCorrect = myNumberEnumValueCheck(attributeValue as myNumberEnum,
                myBinaryRight as myNumberEnum)
164         }
165
166         if (attributeValue instanceof myRange){
167             rightCorrect = myRangeValueCheck(attributeValue as myRange, myBinaryRight
                as myNumberEnum)
168         }
169     }
170     //endregion
171
172     // Check that both right, and left side is true
173     if (leftCorrect && rightCorrect) {
174         return true
175     }

```

```

176     return false;
177 }
178
179 def boolean myStringEnumValueCheck(myStringEnum it, myStringEnum
    expectedValue){
180     val res = it.values.containsAll(expectedValue.values)
181     return res;
182 }
183
184 def boolean myNumberEnumValueCheck(myNumberEnum it, myNumberEnum
    expectedValue){
185     val res = it.values.containsAll(expectedValue.values)
186     return res;
187 }
188
189 def boolean myBooleanValueCheck(myBoolean it, myLiteral expectedValue){
190     // Apparently the expected value gets mapped to a string enum.
191     val value = expectedValue as myStringEnum;
192     val res = it.trueValue.equalsIgnoreCase(value.values.get(0)) ||
        it.falseValue.equalsIgnoreCase(value.values.get(0));
193     return res;
194 }
195
196 def boolean myRangeValueCheck(myRange it, myLiteral expectedValue){
197     if (expectedValue instanceof myNumberEnum) {
198         val res = expectedValue.values.forall[v | it.from <= v && v <= it.to]
199         return res;
200     }
201     if (expectedValue instanceof myRange){
202         val res = it.from <= expectedValue.from && expectedValue.to <= it.to;
203         return res;
204     }
205     return false;
206 }
}

```

4 THE XTEXT GRAMMAR

Listing 3: Xtext Grammar from SmdpDsl.xtext

```

myModel:
2   name=EString
   ('{' myModelContains+=myObject ( "," myModelContains+=myObject)* '}' )?
4   ;

6 myValue returns myValue:
   myBoolean | myRange | myNumberEnum | myStringEnum;

```

```

8
10 EString returns ecore::EString:
    STRING | ID;
12
13 myObject:
14     name=EString
15     '{'
16     ('has' myAttributeContains+=myAttribute ( ","
        myAttributeContains+=myAttribute)* )?
17     ('Constrained by' myObjectHas+=myConstraint (myObjectHas+=myConstraint)* )?
18     '}' ;

20 myConstraint returns myConstraint:
    'if' myIfConstraint=myBinary 'then' myThenConstraint=myBinary
21 ;

24 myAttribute returns myAttribute:
    name=EString
25 '['
26     myAttributeContains=myValue
27     ']' ;

30
31 myBinary returns myExpression:
32     myUnary ({myBinary.myBinaryLeft=current} Oparand=myBinaryOperators
        myBinaryRight=myUnary)*
33 ;
34 // ( and ) is with for fixing left recursion
35 myPrimary returns myExpression:
36     myBoolean | myRange | myNumberEnum | myIdentifier | '(' myBinary ')' |
        myStringEnum
37 ;

38
39 myUnary returns myExpression:
40     {myUnary} (Oparand=myUnaryOperators
        myUnaryExpression=myPrimary) | myPrimary
41 ;

42
44 myBoolean returns myBoolean:
    trueValue=STRING ':'
45     falseValue=STRING
46 ;

48
49 myIdentifier returns myIdentifier:
50     myIntentifierIs=[myAttribute|ID];

52 myRange returns myRange:
    from=INT '-' to=INT

```

```

54     ;

56     myStringEnum returns myStringEnum:
    values+=STRING ( "," values+=STRING)* ;

58

    myNumberEnum returns myNumberEnum:
60     values+=EDouble ( "," values+=EDouble)*;

62     enum myBinaryOperators returns myBinaryOperators:
        and = '&&' | or = '|' | gt = '<' | lt = '>' | eq = '=' | is = 'can'
64     ;

66     enum myUnaryOperators returns myUnaryOperators:
        not = 'not';

68

    EBoolean returns ecore::EBoolean:
70     'true' | 'false';

72     EInt returns ecore::EInt:
        '-'? INT;

74

    EDouble returns ecore::EDouble:
76     ('-'? INT? '.' INT (('E'|'e') '-'? INT)? ) | EInt;

```

5 THE BACK-ENDS

We have implemented two code generators, one in html + JavaScript and one in Java. Our HTML consist of a dropdown for each attribute, containing all possible values. After selecting a value for each attribute, the JavaScript checks if it is a valid assignment. The Java code guides the user through each attribute, shows the possible values, and then checks the assignment in the end. Both saves a valid assignment to a txt file.

The architecture of both solutions is similar; in each solution, constraints is converted into a set of language specific, valid if statements. With help from a couple of generated helper methods, invalid assignments is removed from the possible values. The assignment is then validated by going through each attribute to check if there are still any values left and if the selected value is still available.

The actual conversion of constraints from our domain specific language to either JavaScript or Java is done with a recursive function *generateIfConstraintString*, the function traverses the binary tree and builds a *if statement*.

Listing 4: part of generateIfConstraintString method from JavaCodeGenerator.xtend

```

def String generateIfConstraintString(myBinary it, myIdentifier attribute,
    myBinaryOperators parentOperand){
2   var myIdentifier att;

```

```

var myBinaryOperators pOpe;
4 // If left is a identifier, get the attribute
  if (it.myBinaryLeft instanceof myIdentifier) {
6     att = it.myBinaryLeft as myIdentifier;
    pOpe = it.oparand;
8
  } else {
10     att = attribute;
    pOpe = parentOperand;
12  }

14 // If both left and right are binaries, then both sides must be true
  if (it.myBinaryLeft instanceof myBinary && it.myBinaryRight instanceof
    myBinary) {
16     return "(" + generateIfConstraintString(it.myBinaryLeft as myBinary, att,
        pOpe) + " " + convertOperand(oparand) + " " +
        generateIfConstraintString(it.myBinaryRight as myBinary, att, pOpe)
        + ")"
    }

18
  if (it.myBinaryLeft instanceof myIdentifier && it.myBinaryRight instanceof
    myBinary) {
20     return generateIfConstraintString(it.myBinaryRight as myBinary, att, pOpe)
    }

```

This function uses two helper methods; the first, *ConvertAttributeName*, retrieves the value the user has selected for a given attribute, the other, *convertOperand*, converts our operands from our DSL specific operand type (can, has i.e.) to the language specific equivalent. For JavaScript, *ConvertAttributeName* looks like this:

Listing 5: ConvertAttributeName method from JavaScriptCodeGenerator.xtend

```

def String ConvertAttributeName(String name, myValue type) {
2   if (type instanceof myRange || type instanceof myNumberEnum) {
    return "parseInt(document.querySelector(\"#" + name + "\").value)";
4   }
    return "document.querySelector(\"#" + name + "\").value"
6   }

```

It is using a HTML5 dom selector to get the selected value and convert it to a double if it is expected. The equivalent in the Java, selects the value from a HashMap.

Listing 6: ConvertAttributeName method from JavaCodeGenerator.xtend

```

def String ConvertAttributeName(String name, myValue type) {
2   if (type instanceof myRange || type instanceof myNumberEnum) {
    return "Double.parseDouble(ChosenValues.get(\"" + name + "\"))";
4   }
    return "ChosenValues.get(\"" + name + "\")"
6   }

```

The *then* part of the *if-statement* is converted into code that removes invalid values in the recursive method *generateThenConstraintString*. This is again using *convertOperand* to convert the operand, the generated code is using a hardcoded function to remove the values at runtime.

Listing 7: part of generateThenConstraintString method from JavaCodeGenerator.xtend

```

def String generateThenConstraintString(myBinary it, myIdentifier attribute){
2   if (it.myBinaryLeft instanceof myBinary && it.myBinaryRight instanceof
    myBinary){
    return generateThenConstraintString(it.myBinaryLeft as myBinary, null) +
        generateThenConstraintString(it.myBinaryRight as myBinary, null);
4   }

6   var myIdentifier att;
    // If left is a identifier, get the attribute
8   if (it.myBinaryLeft instanceof myIdentifier) {
    att = it.myBinaryLeft as myIdentifier;
10  } else {
    att = attribute;
12  }

14  if (it.myBinaryLeft instanceof myIdentifier && it.myBinaryRight instanceof
    myValue) {
    var StringBuilder sb = new StringBuilder();
16    if (it.myBinaryRight instanceof myStringEnum) {
    for(v: (it.myBinaryRight as myStringEnum).values) {
18      sb.append("add(\""+ v +"\");");
    }
20    return "removeNonPossibleValuesFromAttribute(\""+
        att.myIdentifierIs.name + "\", new ArrayList<String>(){\" +
        sb.toString + "}}, \"" + it.oparand + "\");"
    }
22  if (it.myBinaryRight instanceof myNumberEnum) {

```

6 THE TEST METHODS AND ARTEFACTS

The overall goal for the test phase, have been to make tests covering all possible paths of the implementation, also known as *path coverage*. This would include testing all possible outcomes in a path.

This means that the number of tests needed to archive full code coverage is determined by the number of possible paths, in each of the steps from the meta-model until the user frontend. Each path should be tested for both positive and negative testcases.

To archive this goal we wish to construct a series of unit-tests within our project using JUnit and Xtend. These tests have been split into three files: *SmdpDslParserTest.xtend*, *Smdp*

pDslGeneratorTest.xtend and *SmdpDslValidator.xtend*. Each file represents test coverage for one of the three major blocks used in the process from DSL to generated code.

The file *SmdpDslParserTest.xtend* contains all tests of the parser, where each test is related to one or more production(s) in the grammar in *SmdpDsl.xtext*, testing how the types from the concrete syntax is inferred from the model and how the parser behaves when given an unexpected input.

SmdpDslValidator.xtend tests our constraints found in *SmdpDslValidator.xtend*, which is used to validate our DSL. Again, this is testing both intended and unintended input.

The *SmdpDslGeneratorTest.xtend* file should tests two different aspects of the code. The first is testing that the generated code has the expected layout and syntax and the second being testing that generated code is functioning.

In all three files we want to use the same snippets of concrete syntax written in *SmdpDsl* to act as test caseses.

The above section outlines how the testing part of the project could and should have been made. We have not been able to achieve this fully. One reason being, that we had problems with the way we wanted to test functionality of the generator. A fully implementation of this testing strategy would have lead to a more robust system and it would imply a system with a higher guarantee of expected behavior.

Furthermore, all tests have been created towards the end of the project, which isn't optimal in terms of using the test results. Because of this lateness, we have found errors in the project which have not been possible to correct this late in the project.

Below is listed a handfull of testcases, showing how we try to get full code coverage. First is four test cases showing how the parser handles different inputs for our *model* and *object* productions. After this a negative test in the parser, suppose to fail because of missing commas. The next 5 tests cases are examples of our testing of our Xtend constraints for validation.

Listing 8: Test examples from SmdpDslParserTest.xtend

```

//Model with 0 myObjects
2  @Test
   def void testMyModelWithoutMyObjects(){
4    val model = '''
       CarFactory{
6
           }
8    ''' .parse;
       Assert::assertEquals(null, model.myModelContains.get(0).name);
10 }

//Model with one myObject without name
12 @Test
   def void testMyModelWithMyObjectsWithoutName(){
14   val model = '''
16     CarFactory{
17       {
18
19     }
20
21   }
22   ''' .parse;
       Assert::assertEquals(null, model.myModelContains.get(0).name);
24 }

//Model with 1 myObject
26 @Test
   def void testMyModelWithOneMyObjects(){
28   val model = '''
30     CarFactory{
31       BMW{
32
33     }
34   }
35   ''' .parse;
       Assert::assertEquals("BMW", model.myModelContains.get(0).name);
       Assert::assertEquals(1, model.myModelContains.size());
38 }

//Model with many myObjects
40
42 @Test
   def void testMyModelWithManyMyObjects(){
44   val model = '''
46     CarFactory{
47       BMW{
48

```

```

    Lada{
50
    }
52
    }
    ''' .parse;
54    Assert::assertEquals("BMW", model.myModelContains.get(0).name);
    //Only 1 myObject allowed
56    try{
        val name = model.myModelContains.get(1).name;
58    }
    catch(Exception e){
60        Assert::assertTrue(e instanceof IndexOutOfBoundsException);
    }
62    Assert::assertNotEquals(2, model.myModelContains.size());
    Assert::assertEquals(1, model.myModelContains.size());
64 }

```

Listing 9: Example of negative test from SmdpDslParserTest.xtend

```

//Model with myObjects with many attributes without comma - Negative test
2 @Test
def void testMyObjectWithManyMyAttributesWithoutComma(){
4    //ConfiguratorProjectPackage.eINSTANCE.eClass()
    val model = '''
6    CarFactory{
    BMW{
8        has
        carType[]
10       engineType[]
        wheel[]
12
    }
14    }
    ''' .parse;
16    Assert::assertEquals(1,
        model.myModelContains.get(0).myAttributeContains.size());
    Assert::assertEquals("carType",
        model.myModelContains.get(0).myAttributeContains.get(0).name);
18    //Must be separated with comma
    try{
20        val attribute = model.myModelContains.get(0).myAttributeContains.get(1).name
    }
22    catch(Exception e){
        Assert::assertTrue(e instanceof IndexOutOfBoundsException);
24    }
}

```

Listing 10: Test examples from SmdpDslValidatorTest.xtend

```

@Test
2  def void WithEmptyString(){
    val model = '''
4  CarFactory {
    BMW {
6      has
        carType ["sports", ""]
8    }
    }
10  '''.parse;

12  val myObject = model.myModelContains.get(0);

14  var attribute = myObject.myAttributeContains.get(0) as myAttribute
    var values = attribute.myAttributeContains as myStringEnum;
16
    Assert::assertFalse(validator.constraint(values))
18 }

20  @Test
    def void WithString(){
22  val model = '''
    CarFactory {
24  BMW {
        has
26  carType ["sportscar", "SUV"]
    }
28  }
    '''.parse;
30
    val myObject = model.myModelContains.get(0);
32
    var attribute = myObject.myAttributeContains.get(0) as myAttribute
    var values = attribute.myAttributeContains as myStringEnum;
34
    Assert::assertTrue(validator.constraint(values))
36 }

38  @Test
    def void DuplicateAttributesName(){
40  val model = '''
    CarFactory {
42  BMW {
        has
44  carType ["sportscar", "SUV"],
46  carType ["sportscar", "SUV"]
    }
48  }

```

```

    ''' .parse;
50
    val myObject = model.myModelContains.get(0);
52    Assert::assertFalse validator.constraint(myObject))
    }

```

Listing 11: Test examples from SmdpDslValidatorTest.xtend

```

@Test
2  def void WrongRangeConstraint(){
    val model = '''
4  carFactory {
    BMW {
6      has
        seatHeat ["Seat Heat":"No Seat Heat"],
8        wheel [14-28]

10     Constrained by
        if seatHeat = "Seat Heat"
12     then wheel can 14,29
    }
14 }''' .parse;
    val myObject = model.myModelContains.get(0);
16    val myCon = myObject.myObjectHas.get(0);
    Assert::assertFalse(validator.constraint(myCon))
18 }

```

Listing 12: Test examples from SmdpDslValidatorTest.xtend

```

2    Constrained by
        if seatHeat = "Seat Heat"
4    then wheel can "wrong"
    }
6 }''' .parse;
    val myObject = model.myModelContains.get(0);
8    val myCon = myObject.myObjectHas.get(0);
    Assert::assertFalse(validator.constraint(myCon))
10 }

12 @Test
    def void CorrectStringEnumConstraint(){
14    val model = '''
        carFactory {
16    BMW {
        has
18    seatHeat ["Seat Heat":"No Seat Heat"],
        wheel ["4-wheel", "two-wheel"]
20

```

```

        Constrained by
22         if seatHeat = "Seat Heat"
           then wheel can "4-wheel"
24     }
}'''
.parse;
26 val myObject = model.myModelContains.get(0);
val myCon = myObject.myObjectHas.get(0);
28 Assert::assertTrue(validator.constraint(myCon))
}

30
    @Test
32 def void CorrectBooleanConstraint(){
val model = '''
34 carFactory {
BMW {
36     has
        seatHeat ["Seat Heat":"No Seat Heat"],
38     wheel ["4-wheel":"two-wheel"]

        Constrained by
            if seatHeat = "Seat Heat"
            then wheel can "4-wheel"
        }
44 }'''
.parse;
val myObject = model.myModelContains.get(0);
46 val myCon = myObject.myObjectHas.get(0);
Assert::assertTrue(validator.constraint(myCon))
48 }

50
    @Test
def void WrongBooleanConstraint(){
52 val model = '''
carFactory {
54 BMW {
    has
56     seatHeat ["Seat Heat":"No Seat Heat"],
        wheel ["4-wheel":"two-wheel"]

58     Constrained by
        if seatHeat = "Seat Heat"
        then wheel can "wrong"
62     }
}'''
.parse;
64 val myObject = model.myModelContains.get(0);
val myCon = myObject.myObjectHas.get(0);
66 Assert::assertFalse(validator.constraint(myCon))
}

68

```

```

70     @Test
72     def void AttributesName(){
73         val model = '''
74         CarFactory {
75             BMW {
76                 has
77                     carType ["sportscar", "SUV"],
78                     car ["sportscar", "SUV"]
79             }
80         }
81         ''''.parse;
82
83         val myObject = model.myModelContains.get(0);
84         Assert::assertTrue(validator.constraint(myObject))
85     }
86
87     @Test
88     def void CorrectRangeValue(){
89         val model = '''
90         carFactory {
91             BMW {
92                 has
93                     wheel [14-28]
94             }
95         }''''.parse;
96         val myObject = model.myModelContains.get(0);
97         val myCon = myObject.myAttributeContains.get(0).myAttributeContains as
98             myRange;
99         Assert::assertTrue(validator.constraint(myCon))
100     }
101
102     @Test
103     def void WrongRangeValue(){
104         val model = '''
105         carFactory {
106             BMW {
107                 has
108                     wheel [14-13]
109             }
110         }''''.parse;
111         val myObject = model.myModelContains.get(0);
112         val myCon = myObject.myAttributeContains.get(0).myAttributeContains as
113             myRange;
114         Assert::assertFalse(validator.constraint(myCon))
115     }

```
