IT UNIVERSITY OF COPENHAGEN

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 {
  BMW {
   has
    carType ["SUV","Pickup","Minivan","Supercar","Sedan","Stationcar",
        "Microcar"],
    enginType ["Diesel","Gas","Electric","Hybrid"],
    doors ["5-Door","4-Door","2-Door"],
    seatType ["Racing Seats","Standard Seats","Hardwood Seats"],
    seatHeat ["Seat Heat":"No Seat Heat"],
    wheel [14-28],
    numberOfSeat [2,4,5,7,8],
    color
        ["DarkSalmon","FloralWhite","MidnightBlue","OliveDrab","RosyBrown","LemonChiffon",
        "DimGray"]




   Constrained by
    if carType = "Supercar"
    then (seatHeat can "Seat Heat") && (doors can "2-Door")

    if seatHeat = "Seat Heat"
    then seatType can "Standard Seats"

    if (wheel > 24) && (carType = ("SUV" | "Pickup"))
    then enginType can "Diesel","Gas"

    if enginType = ("Electric" | "Hybrid")
    then carType can "Sedan","Stationcar","Microcar"

    if doors = "5-Door"
    then carType can "SUV","Minivan","Sedan","Stationcar"

    if (seatType = "Hardwood Seats") && (color = "LemonChiffon")
    then (enginType can "Gas") && (seatHeat can "No Seat Heat")

    if numberOfSeat > 5
    then carType can "Stationcar","Minivan"

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

2 THE VIEWS OF THE META-MODEL

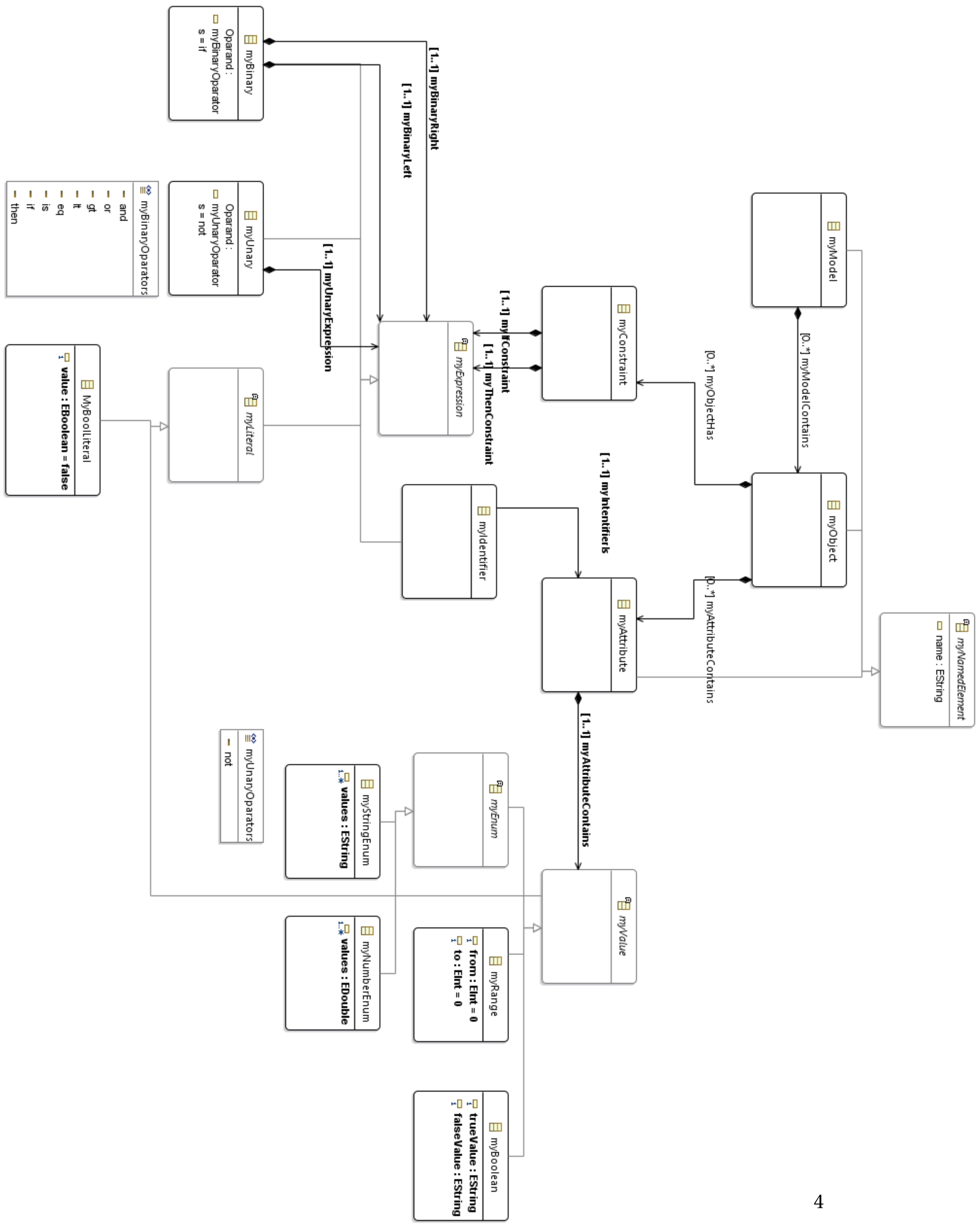

Figure 2.1: View of the whole meta-model

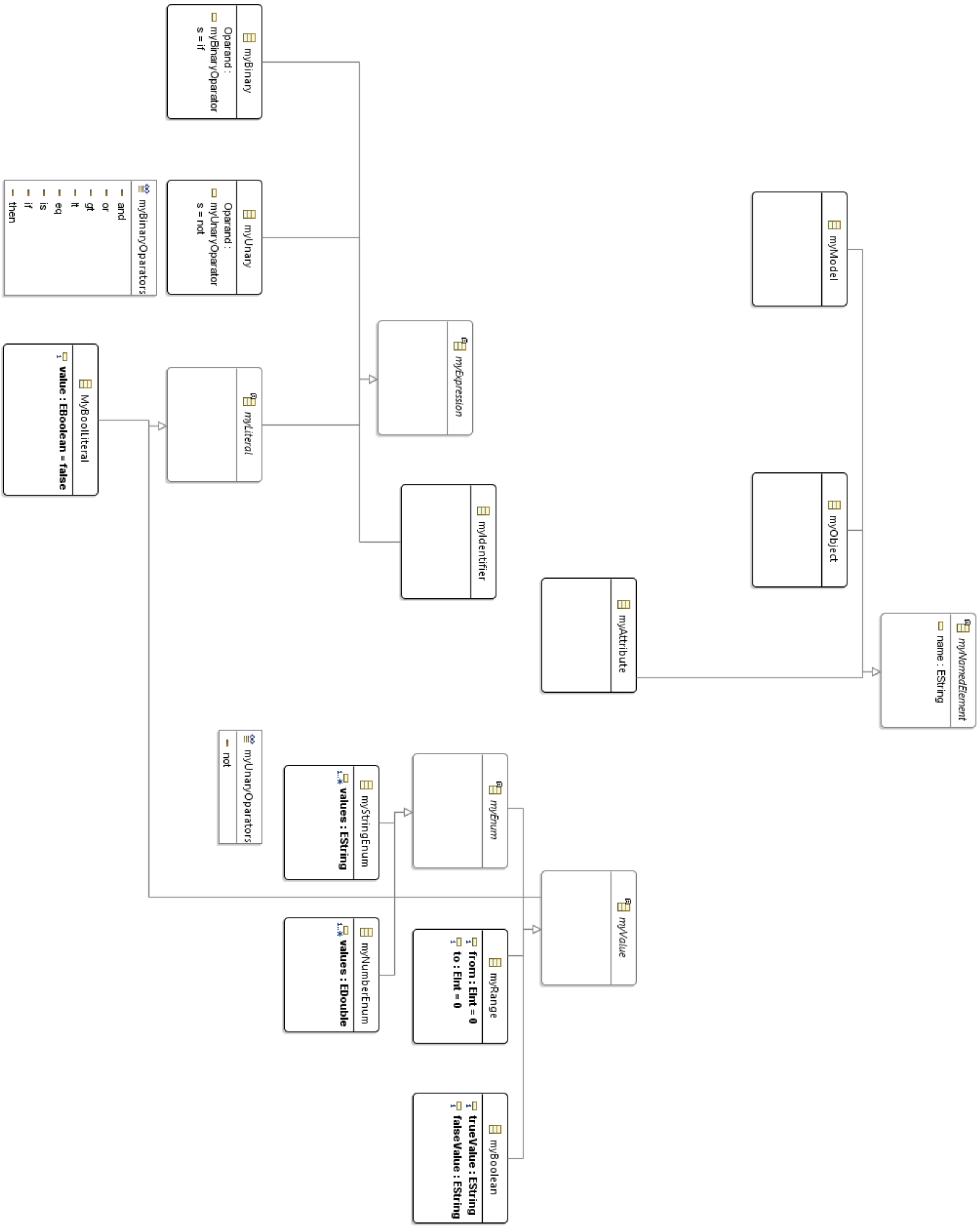

Figure 2.2: View of the taxonomy

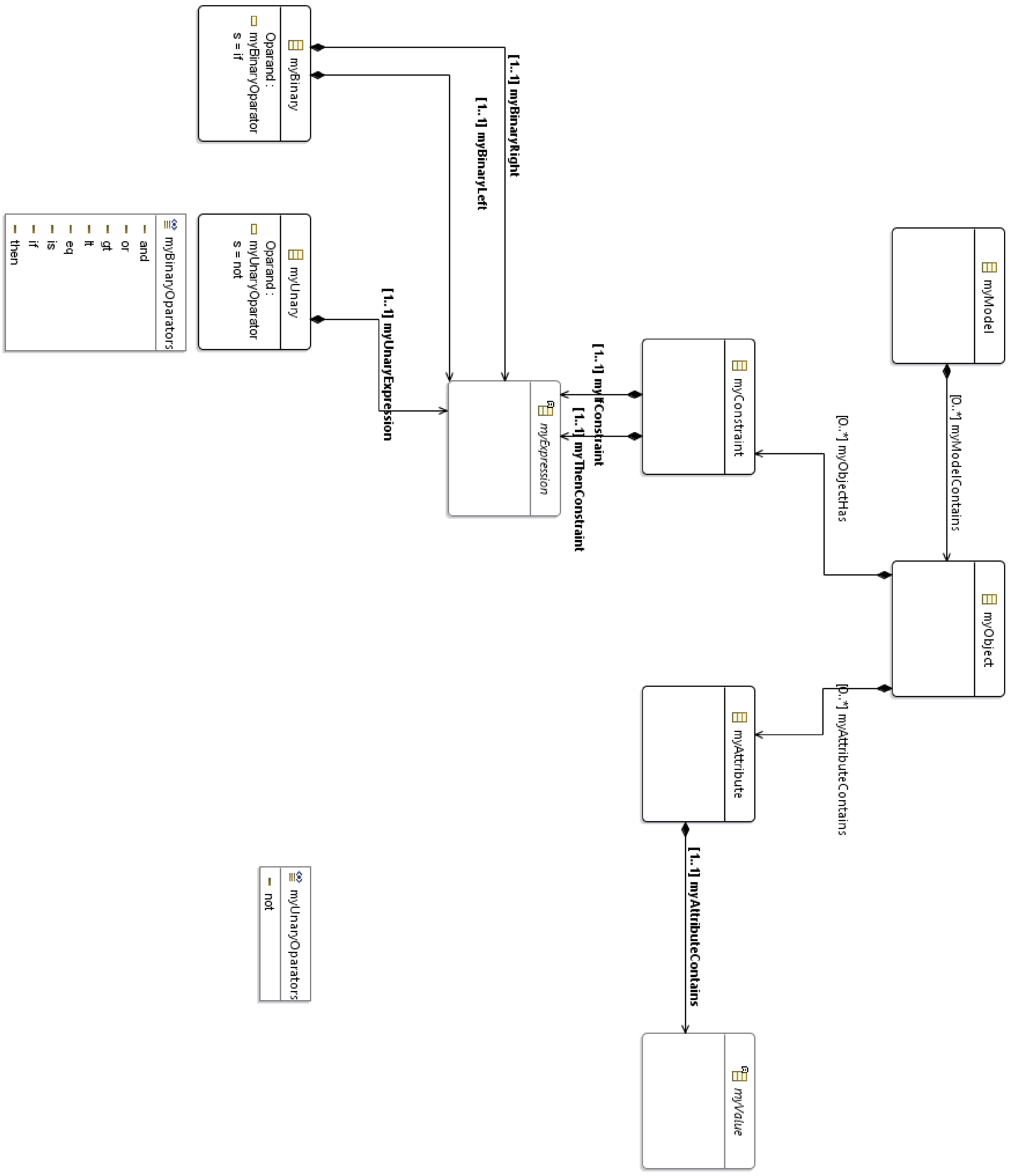

Figure 2.3: View of the partonomy

3 The Static semantics

Listing 2: Static Constraints

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

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

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

 }

 @Check
 def boolean constraint(myNumberEnum it) {
  try {
   if (values.length == 0) {
   error("All number enum must have a size of at least 1", it, null)
  }
  return true;
  } catch (Exception e) {
   e.printStackTrace();
  return false;
 }
}
@Check
def boolean constraint(myRange it) {
 try {
  if (from > to) {
  error("The start value in a range cannot be larger than the end value", it,
      null);
 }
 return true;
 } catch (Exception e) {
  e.printStackTrace();
  return false;
 }

}
@Check
def boolean constraint(myBoolean it) {
 try {
  if (trueValue == falseValue) {
  error("The values for boolean can’t be the same", it, null);
 }

 if (trueValue == "" || falseValue == ""){
  error("Boolean must be asigned a value",it, null);
 }

 return true;
 } catch (Exception e) {
  e.printStackTrace();
  return false;
 }
}
@Check
def boolean constraint (myStringEnum it) {
 try {
  if (values.length == 0 || values.exists[v | v.equalsIgnoreCase("")]) {
   error("String enum must contain a value",it, null)
  }
  return true
 } catch (Exception e) {
  e.printStackTrace();
  return false;
 }
}

 /* Helper method to go throuh the expression tree, to check if all values
     are valid
  * We don’t take the operan into account, there for it will be possible to
      set values that are outside a range scope
  * if ’<’ or ’>’ is used.
  */
  def boolean myValuesCheck(myBinary it, myIdentifier attribute){
   var boolean leftCorrect = false;
   var boolean rightCorrect = false;
   var myIdentifier att;
   // If left is a identifier, get the attribute
   if (it.myBinaryLeft instanceof myIdentifier) {
    att = it.myBinaryLeft as myIdentifier;
    leftCorrect = true;
   } else {
    att = attribute;
   }

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

 // If the right is a binary, then go one depth deeper.
   if (it.myBinaryRight instanceof myBinary){
    return myValuesCheck(it.myBinaryRight as myBinary, att)
   }
   //region Check the values in the left side
   if (it.myBinaryLeft instanceof myStringEnum) {
    // If the left is a string, then the value must either be a StringEnum or
        a Boolean
    val attributeValue = att.myIntentifierIs.myAttributeContains;

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

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

    if (it.oparand != myBinaryOparators.OR) {
     error("Operand cannot be other than ’|’",it,null)
     leftCorrect = false;
    }
   }
```

```
if (it.myBinaryLeft instanceof myNumberEnum) {
 val attributeValue = att.myIntentifierIs.myAttributeContains;
 // If the left is a number, then the value must either be a NumberEnum or
     a Range
 if (attributeValue instanceof myNumberEnum){
  leftCorrect = myNumberEnumValueCheck(attributeValue as myNumberEnum,
      myBinaryLeft as myNumberEnum)
 }

 if (attributeValue instanceof myRange){
  leftCorrect = myRangeValueCheck(attributeValue as myRange, myBinaryLeft
      as myNumberEnum)
 }
}
//endregion

//region Check the values in the right side
if (it.myBinaryRight instanceof myStringEnum) {
 val attributeValue = att.myIntentifierIs.myAttributeContains;

 if (attributeValue instanceof myStringEnum) {
  rightCorrect = myStringEnumValueCheck(attributeValue as myStringEnum,
      myBinaryRight as myStringEnum)
 }

 if (attributeValue instanceof myBoolean){
  rightCorrect = myBooleanValueCheck(attributeValue as myBoolean,
      myBinaryRight as myStringEnum)
 }
}

if (it.myBinaryRight instanceof myNumberEnum) {
 val attributeValue = att.myIntentifierIs.myAttributeContains;
 if (attributeValue instanceof myNumberEnum){
  rightCorrect = myNumberEnumValueCheck(attributeValue as myNumberEnum,
      myBinaryRight as myNumberEnum)
 }

 if (attributeValue instanceof myRange){
  rightCorrect = myRangeValueCheck(attributeValue as myRange, myBinaryRight
      as myNumberEnum)
 }
}
//endregion

// Check that both right, and left side is true
 if (leftCorrect && rightCorrect) {
  return true
 }
```

```
    return false;
   }

   def boolean myStringEnumValueCheck(myStringEnum it, myStringEnum
       expectedValue){
    val res = it.values.containsAll(expectedValue.values)
    return res;
   }

   def boolean myNumberEnumValueCheck(myNumberEnum it, myNumberEnum
       expectedValue){
    val res = it.values.containsAll(expectedValue.values)
    return res;
   }

   def boolean myBooleanValueCheck(myBoolean it, myLiteral expectedValue){
    // Apparently the expected value gets mapped to a string enum.
    val value = expectedValue as myStringEnum;
    val res = it.trueValue.equalsIgnoreCase(value.values.get(0)) ||
        it.falseValue.equalsIgnoreCase(value.values.get(0));
    return res;
   }

   def boolean myRangeValueCheck(myRange it, myLiteral expectedValue){
       if (expectedValue instanceof myNumberEnum) {
     val res = expectedValue.values.forall[v | it.from <= v && v <= it.to]
     return res;
    }
    if (expectedValue instanceof myRange){
     val res = it.from <= expectedValue.from && expectedValue.to <= it.to;
     return res;
    }
    return false;
   }
}
```

4 The Xtext grammar

Listing 3: Xtext Grammar from SmdpDsl.xtext

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

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

```


EString returns ecore::EString:
 STRING | ID;

myObject:
 name=EString
 '{'
  ('has' myAttributeContains+=myAttribute ( ","
      myAttributeContains+=myAttribute)* )?
  ('Constrained by' myObjectHas+=myConstraint (myObjectHas+=myConstraint)* )?
 '}';

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

myAttribute returns myAttribute:
 name=EString
 '['
  myAttributeContains=myValue
 ']';


myBinary returns myExpression:
  myUnary ({myBinary.myBinaryLeft=current} Oparand=myBinaryOparators
      myBinaryRight=myUnary)*
;
// ( and ) is with for fixing left recursion
myPrimary returns myExpression:
  myBoolean | myRange | myNumberEnum | myIdentifier | '(' myBinary ')' |
      myStringEnum
;

myUnary returns myExpression:
  {myUnary} (Oparand=myUnaryOparators
  myUnaryExpression=myPrimary) | myPrimary
;

myBoolean returns myBoolean:
  trueValue=STRING ':'
  falseValue=STRING
 ;

myIdentifier returns myIdentifier:
 myIntentifierIs=[myAttribute|ID];

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

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

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

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

enum myUnaryOparators returns myUnaryOparators:
    not = 'not';

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

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

EDouble returns ecore::EDouble:
 ('-'? 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,
     myBinaryOparators parentOperand){
    var myIdentifier att;
    var myBinaryOparators pOpe;
    // If left is a identifier, get the attribute
    if (it.myBinaryLeft instanceof myIdentifier) {
     att = it.myBinaryLeft as myIdentifier;
     pOpe = it.oparand;

    } else {
     att = attribute;
     pOpe = parentOperand;
    }

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

    if (it.myBinaryLeft instanceof myIdentifier && it.myBinaryRight instanceof
        myBinary) {
    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) {
 if (type instanceof myRange || type instanceof myNumberEnum) {
  return "parseInt(document.querySelector(\"#" + name + "\").value)";
 }
 return "document.querySelector(\"#" + name + "\").value"
}
```

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) {
 if (type instanceof myRange || type instanceof myNumberEnum) {
  return "Double.parseDouble(ChosenValues.get(\"" + name + "\"))";
 }
 return "ChosenValues.get(\"" + name + "\")"
}
```

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){
 if (it.myBinaryLeft instanceof myBinary && it.myBinaryRight instanceof
     myBinary){
     return generateThenConstraintString(it.myBinaryLeft as myBinary, null) +
         generateThenConstraintString(it.myBinaryRight as myBinary, null);
    }

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

    if (it.myBinaryLeft instanceof myIdentifier && it.myBinaryRight instanceof
       myValue) {
     var StringBuilder sb = new StringBuilder();
     if (it.myBinaryRight instanceof myStringEnum) {
      for(v: (it.myBinaryRight as myStringEnum).values) {
       sb.append("add(\""+ v +"\");");
      }
      return "removeNonPossibleValuesFromAttribute(\""+
          att.myIntentifierIs.name +"\", new ArrayList<String>(){{" +
          sb.toString +"}}, \"" + it.oparand + "\");"
     }
     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*, *Smd-*

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 intented and unintented 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
@Test
def void testMyModelWithoutMyObjects(){
 val model = '''
  CarFactory{

  }
 '''.parse;
 Assert::assertEquals(null, model.myModelContains.get(0).name);
}

//Model with one myObject without name
@Test
def void testMyModelWithMyObjectsWithoutName(){
 val model = '''
  CarFactory{
   {

   }

  }
 '''.parse;
 Assert::assertEquals(null, model.myModelContains.get(0).name);
}

//Model with 1 myObject
@Test
def void testMyModelWithOneMyObjects(){
 val model = '''
  CarFactory{
  BMW{

  }
  }
 '''.parse;
 Assert::assertEquals("BMW", model.myModelContains.get(0).name);
 Assert::assertEquals(1, model.myModelContains.size());
}


//Model with many myObjects
@Test
def void testMyModelWithManyMyObjects(){
 val model = '''
  CarFactory{
  BMW{

  }
  Lada{

  }
  }
 ''' .parse;
 Assert::assertEquals("BMW", model.myModelContains.get(0).name);
 //Only 1 myObject allowed
 try{
  val name = model.myModelContains.get(1).name;
 }
 catch(Exception e){
  Assert::assertTrue(e instanceof IndexOutOfBoundsException);
 }
 Assert::assertNotEquals(2, model.myModelContains.size());
 Assert::assertEquals(1, model.myModelContains.size());
}
```

Listing 9: Example of negative test from SmdpDslParserTest.xtend

```
//Model with myObjects with many attributes without comma - Negative test
@Test
def void testMyObjectWithManyMyAttributesWithoutComma(){
 //ConfiguratorProjectPackage.eINSTANCE.eClass()
 val model = '''
  CarFactory{
  BMW{
   has
    carType[]
    engineType[]
    wheel[]

  }
  }
 ''' .parse;
 Assert::assertEquals(1,
     model.myModelContains.get(0).myAttributeContains.size());
 Assert::assertEquals("carType",
     model.myModelContains.get(0).myAttributeContains.get(0).name);
 //Must be separated with comma
 try{
  val attribute = model.myModelContains.get(0).myAttributeContains.get(1).name
 }
 catch(Exception e){
  Assert::assertTrue(e instanceof IndexOutOfBoundsException);
 }
}
```

Listing 10: Test examples from SmdpDslValidatorTest.xtend

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

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

 var attribute = myObject.myAttributeContains.get(0) as myAttribute
 var values = attribute.myAttributeContains as myStringEnum;

 Assert::assertFalse(validator.constraint(values))
}

  @Test
 def void WithString(){
 val model = '''
 CarFactory {
  BMW {
   has
     carType ["sportscar", "SUV"]
  }
 }
 '''.parse;

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

 var attribute = myObject.myAttributeContains.get(0) as myAttribute
 var values = attribute.myAttributeContains as myStringEnum;

 Assert::assertTrue(validator.constraint(values))
}

  @Test
 def void DublicateAttributesName(){
 val model = '''
 CarFactory {
  BMW {
   has
     carType ["sportscar", "SUV"],
     carType ["sportscar", "SUV"]
  }
 }
 '''.parse;

 val myObject = model.myModelContains.get(0);
 Assert::assertFalse(validator.constraint(myObject))
}
```

Listing 11: Test examples from SmdpDslValidatorTest.xtend

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

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

Listing 12: Test examples from SmdpDslValidatorTest.xtend

```

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

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

```

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

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

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

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

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


  @Test
 def void AttributesName(){
 val model = '''
 CarFactory {
  BMW {
   has
     carType ["sportscar", "SUV"],
     car ["sportscar", "SUV"]
  }
 }
 '''.parse;

 val myObject = model.myModelContains.get(0);
 Assert::assertTrue(validator.constraint(myObject))
}

@Test
def void CorrectRangeValue(){
 val model = '''
 carFactory {
 BMW {
  has
   wheel [14-28]
 }
}'''.parse;
 val myObject = model.myModelContains.get(0);
 val myCon = myObject.myAttributeContains.get(0).myAttributeContains as
     myRange;
 Assert::assertTrue(validator.constraint(myCon))
}

@Test
def void WrongRangeValue(){
 val model = '''
 carFactory {
 BMW {
  has
   wheel [14-13]
 }
}'''.parse;
 val myObject = model.myModelContains.get(0);
 val myCon = myObject.myAttributeContains.get(0).myAttributeContains as
     myRange;
 Assert::assertFalse(validator.constraint(myCon))
}
```