Model driven development

Developing a DSL for configurator applications

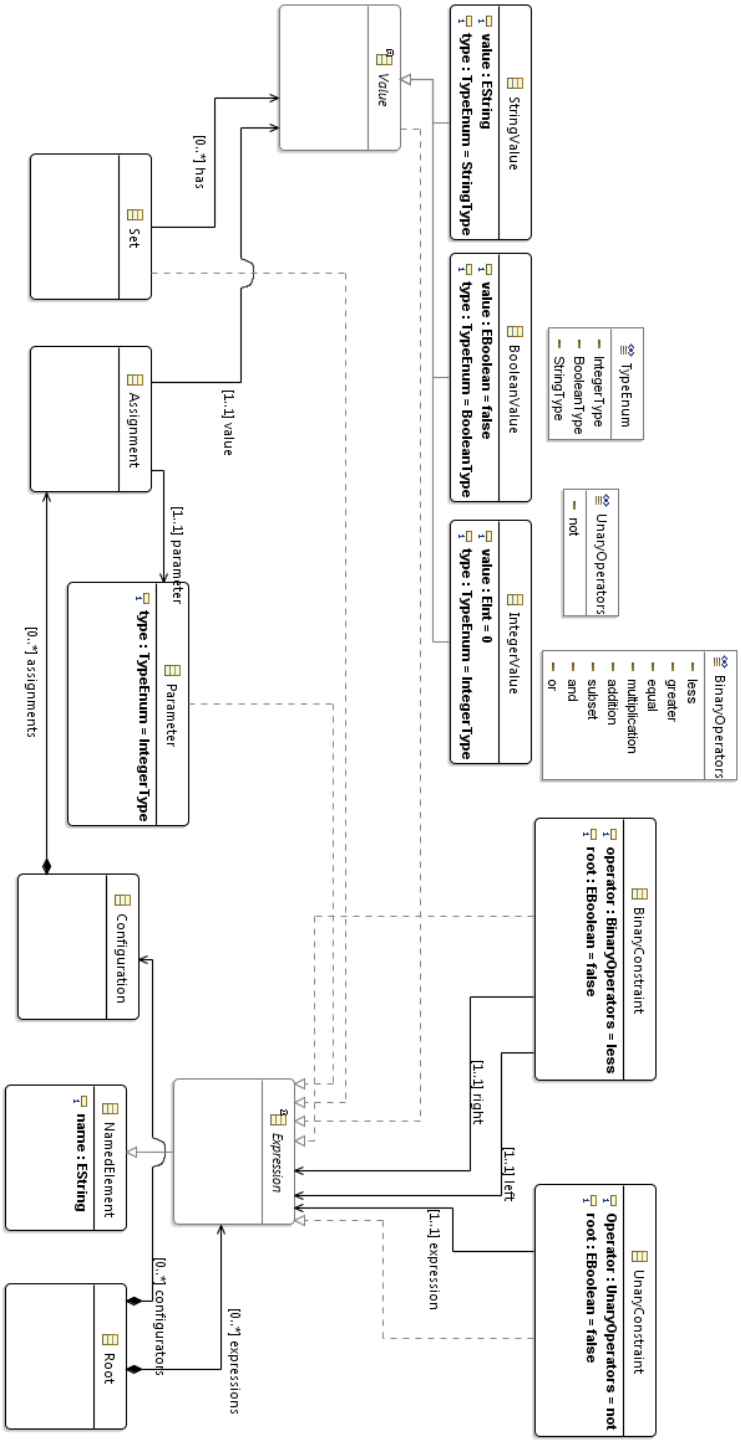
Group 20 [R. France]
Andreas Precht Poulsen, appo@itu.dk
Mark Thorhauge, marth@itu.dk
Mikkel Hvilshøj Funch, mhvf@itu.dk
Mikkel Thomsen, mikt@itu.dk

Example of our DSL

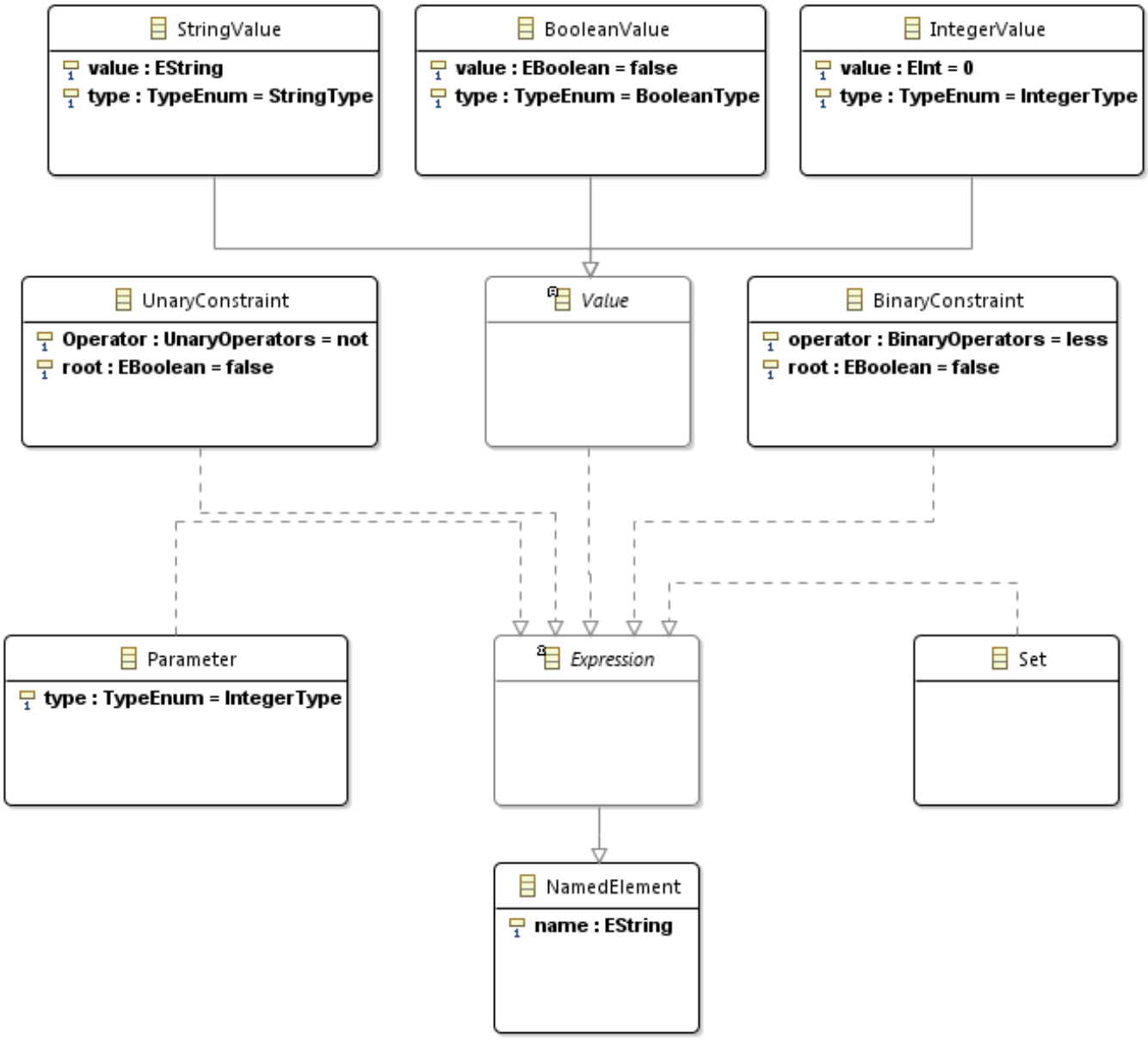
```
int width
int height
int length
string name
boolean convex
int maxVolume = 10000
int two = 2
string reservedName = "box"

uc true nameIsNotReserved ! nameIsReserved
bc nameIsReserved name == reservedName

bc true masterVolumeConstraint standardVolumeIfNotConvex ||
increaseVolumeIfConvex
bc standardVolumeIfNotConvex notConvex && volumeConstraint
bc increaseVolumeIfConvex convex && doubleVolumeConstraint

bc volumeConstraint volume < maxVolume
bc doubleVolumeConstraint volume < doubleVolume

bc doubleVolume maxVolume * two

bc groundSurface width * length
bc volume groundSurface * height

uc notConvex ! convex
```

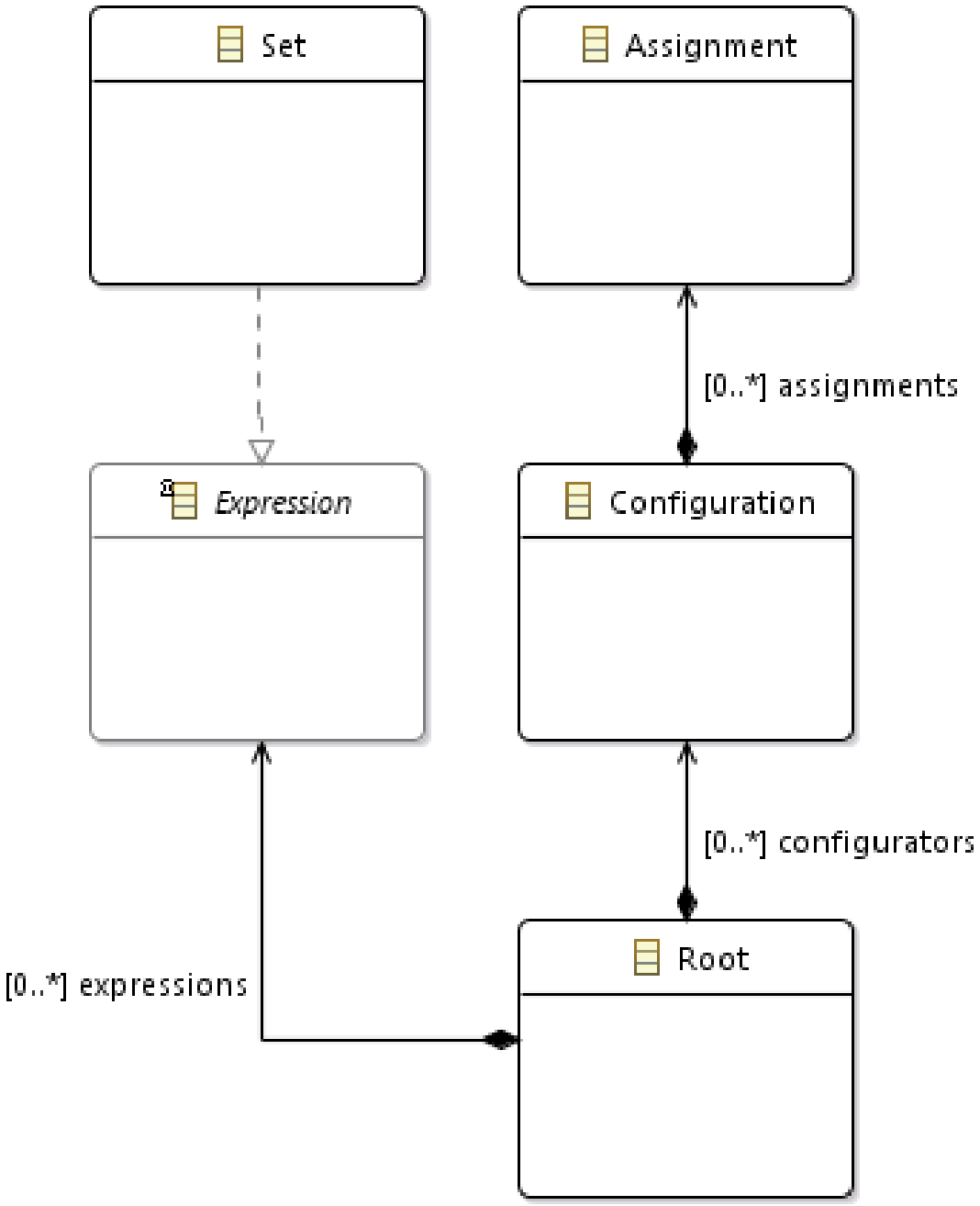
Views

Meta model

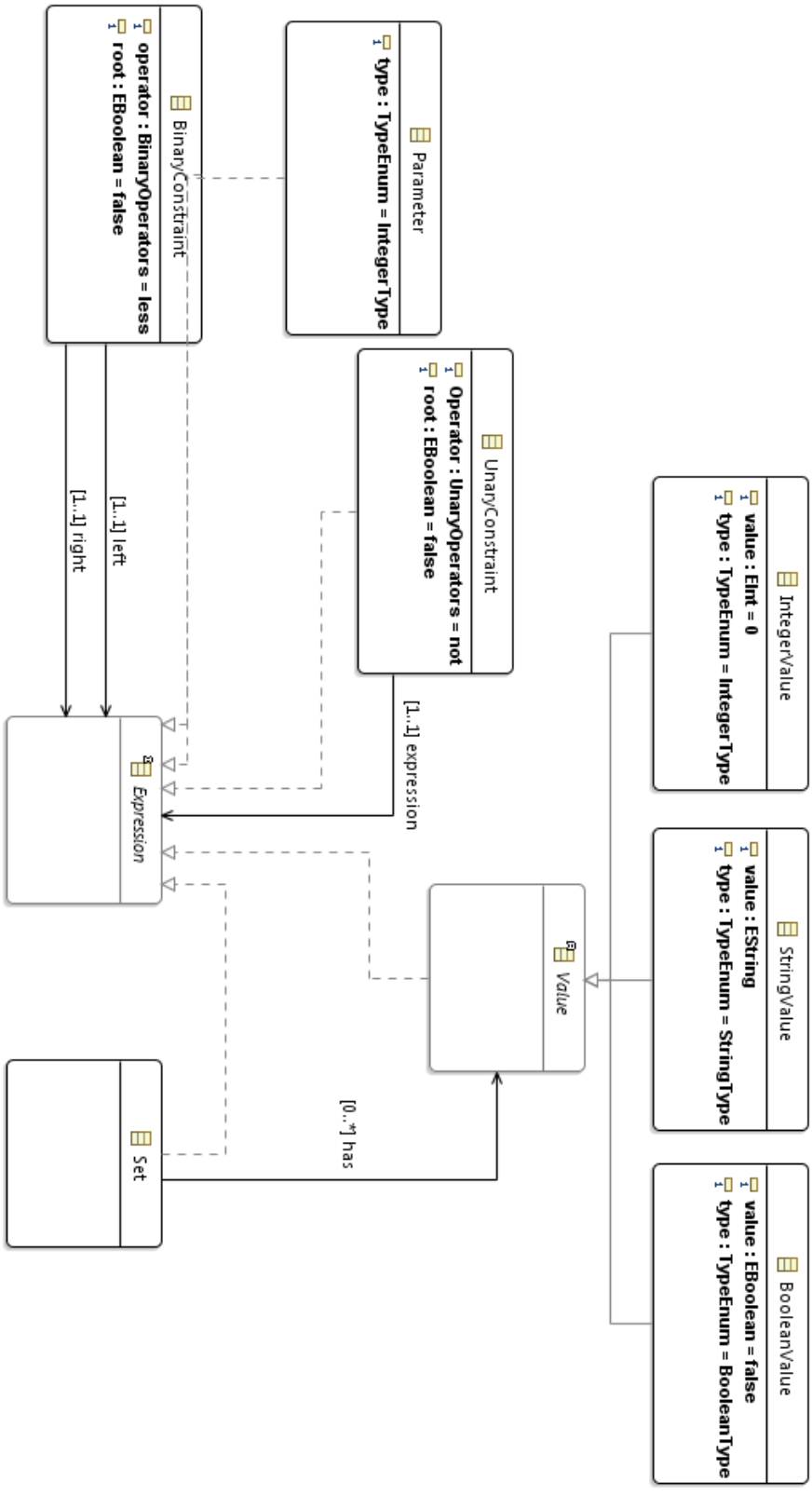


Taxonomy

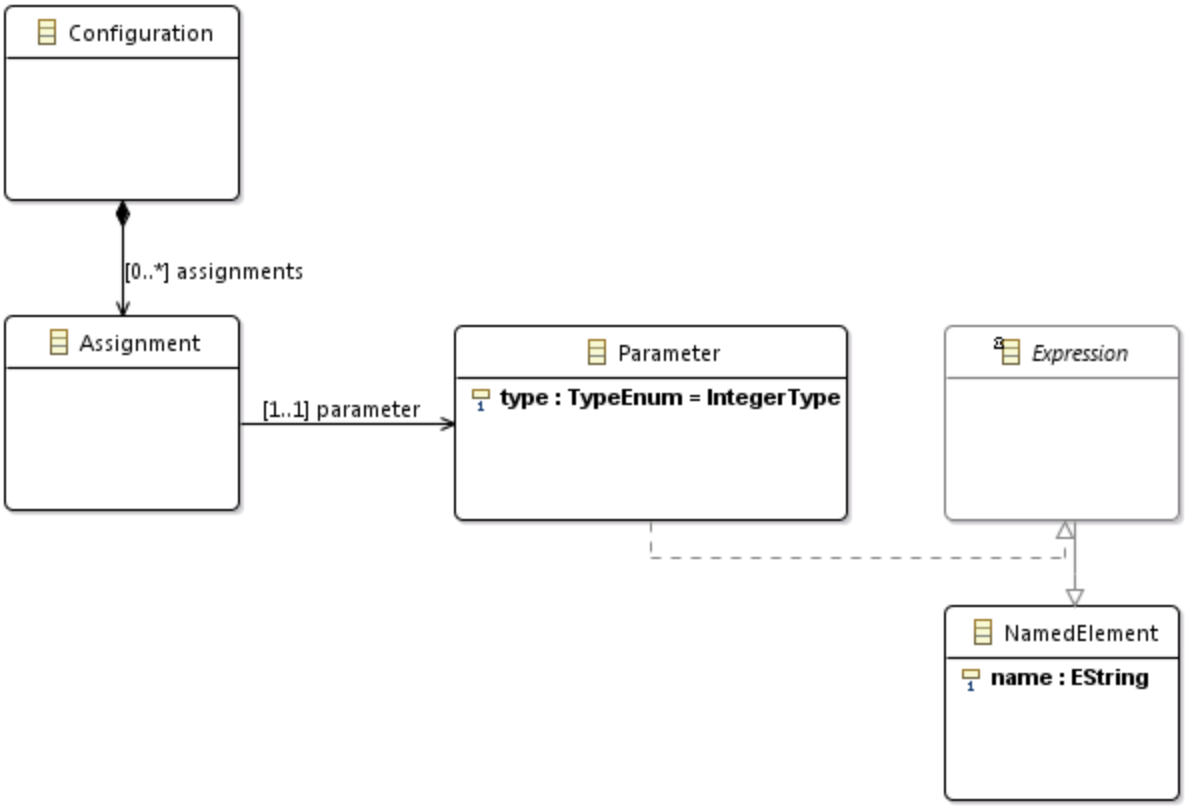

StringValue
value : EString
type : TypeEnum = StringType
BooleanValue
value : EBoolean = false
type : TypeEnum = BooleanType
IntegerValue
value : EInt = 0
type : TypeEnum = IntegerType
UnaryConstraint
Operator : UnaryOperators = not
root : EBoolean = false
Value
BinaryConstraint
operator : BinaryOperators = less
root : EBoolean = false
Parameter
type : TypeEnum = IntegerType
Expression
Set
NamedElement
name : EString


Set
Assignment
[0..*] assignments
Expression
Configuration
[0..*] configurators
[0..*] expressions
Root

Expressions



Configuration Assignment



Static semantics

```
package org.xtext.cfgdsl.validation

import ConfiguratorPackage.BinaryConstraint
import ConfiguratorPackage.BooleanValue
import ConfiguratorPackage.Configuration
import ConfiguratorPackage.Expression
import ConfiguratorPackage.IntegerValue
import ConfiguratorPackage.NamedElement
import ConfiguratorPackage.Parameter
import ConfiguratorPackage.Root
import ConfiguratorPackage.Set
import ConfiguratorPackage.StringValue
import ConfiguratorPackage.TypeEnum
import ConfiguratorPackage.UnaryConstraint
import ConfiguratorPackage.Value
import java.util.HashMap
import java.util.HashSet
import org.eclipse.emf.ecore.EObject
```

```
import static ConfiguratorPackage.BinaryOperators.*

class CfgDslValidator extends AbstractCfgDslValidator {

  /* Fall back for all types that are not constrained */
  def static dispatch boolean constraint(EObject it) {
    true
  }

  def static dispatch boolean constraint(NamedElement it) {
    name != null && !name.isEmpty
  }

  def static dispatch boolean constraint(Root it){
    constraintAssignment(it) && constraintParams(it)
  }

  def static dispatch boolean constraint(Configuration it) {
    constraintOneAssignmentPerParameter(it)
  }

  def static dispatch boolean constraint(Set it) {
    constraintSet(it)
  }

  def static dispatch boolean constraint(BinaryConstraint it) {
    constraintBinary(it) && constraintBinaryRoot(it)
  }

  def static dispatch boolean constraint(UnaryConstraint it) {
    constraintUnary(it)
  }

  def static dispatch boolean constraint(Value it) {
    constraintValueType(it)
  }

  def static dispatch boolean constraint(Parameter it){
    val ne = it as NamedElement
    constraint(ne) && type != null
  }

  /**
   * Check for unique parameters in the root element.
```

```
   * If the sizes of the params and unique params are not the same
   * we have duplicate entries
   */
  def static boolean constraintParams(Root it){
    val params = expressions.filter[e | e instanceof Parameter]
    val uniqueParams = params.fold(new HashSet<String>) [ s, e |
s.add(e.name); s ]
    params.size == uniqueParams.size
  }

  /**
   * Check that the value of each assignment has the correct type
   * according to the assigned type of the parameter
   */
  def static boolean constraintAssignment(Root it){
    val params = expressions.filter[e | e instanceof Parameter]
    val typeMap = params.fold(new HashMap<String, TypeEnum>)[ m, e |
val p = e as Parameter; m.put(p.name, p.type); m]
    configurators.forall [c |
      val strings  = c.assignments.filter [ a | a.value instanceof
StringValue ]
      val integers = c.assignments.filter [ a | a.value instanceof
IntegerValue ]
      val booleans = c.assignments.filter [ a | a.value instanceof
BooleanValue ]

      val stringsSatisfied  = strings.fold(true)  [ b, a | b &&
TypeEnum.STRING_TYPE.equals(typeMap.get(a.parameter.name)) ]
      val integersSatisfied = integers.fold(true) [ b, a | b &&
TypeEnum.INTEGER_TYPE.equals(typeMap.get(a.parameter.name)) ]
      val booleansSatisfied = booleans.fold(true) [ b, a | b &&
TypeEnum.BOOLEAN_TYPE.equals(typeMap.get(a.parameter.name)) ]

      stringsSatisfied && integersSatisfied && booleansSatisfied
    ]
  }

  /**
   * Check the type of the values are correctly typed
   */
  def static boolean constraintValueType(Value   ) {
    if(it instanceof IntegerValue) {
      val i = it as IntegerValue
      return TypeEnum.INTEGER_TYPE.equals(i.type)
    }
```

```
    if(it instanceof StringValue) {
      val i = it as StringValue
      return TypeEnum.STRING_TYPE.equals(i.type)
    }
    if(it instanceof BooleanValue) {
      val i = it as BooleanValue
      return TypeEnum.BOOLEAN_TYPE.equals(i.type)
    }
    return false
  }

  /**
   * Check for unique parameters in a given configuration
   */
  def static boolean constraintOneAssignmentPerParameter( Configuration
it ) {
    val list = assignments.filter()[a | a.parameter == null || a.value ==
null]
    if(list.size != 0)
      return false;

    if(!assignments.fold(true)[b, a | val p = a.parameter as Parameter; b
&& constraint(p)])
      return false;

    val uniqueParams = assignments.fold(new HashSet<String>) [ s, a |
s.add(a.parameter.name); s ]
    uniqueParams.size == assignments.size
  }

  /**
   * Check that the types of BinaryConstraints are good
   */
  def static boolean constraintBinary(BinaryConstraint it) {
    var b = false
    switch (operator) {
    case ADDITION:
      return valueResolver(left).equals(TypeEnum.INTEGER_TYPE) &&
valueResolver(right).equals(TypeEnum.INTEGER_TYPE)
    case AND:
      return valueResolver(left).equals(TypeEnum.BOOLEAN_TYPE) &&
valueResolver(right).equals(TypeEnum.BOOLEAN_TYPE)
    case EQUAL:
      return valueResolver(left).equals(valueResolver(right))
    case GREATER:
```

```
      return valueResolver(left).equals(TypeEnum.INTEGER_TYPE) &&
valueResolver(right).equals(TypeEnum.INTEGER_TYPE)
    case LESS:
      return valueResolver(left).equals(TypeEnum.INTEGER_TYPE) &&
valueResolver(right).equals(TypeEnum.INTEGER_TYPE)
    case MULTIPLICATION:
      return valueResolver(left).equals(TypeEnum.INTEGER_TYPE) &&
valueResolver(right).equals(TypeEnum.INTEGER_TYPE)
    case OR:
      return valueResolver(left).equals(TypeEnum.BOOLEAN_TYPE) &&
valueResolver(right).equals(TypeEnum.BOOLEAN_TYPE)
    case SUBSET:
      return valueResolver(left).equals(valueResolver(right))
    }
  }

  /**
   * Check that a root BinaryConstraint resolves to a boolean type
   */
  def static boolean constraintBinaryRoot(BinaryConstraint it) {
    if(root)
      return expressionResolver(left) && expressionResolver(right)
    false
  }

  /**
   * Helper to resolve the return type of an Expression
   */
  def static boolean expressionResolver(Expression it) {
    if(it instanceof BinaryConstraint) {
      val bc = it as BinaryConstraint
      if(bc.root) {
        return false;
      } else {
        return expressionResolver(left) && expressionResolver(right)
      }
    } else if(it instanceof UnaryConstraint) {
      val uc = it as UnaryConstraint
      return expressionResolver(uc.expression)
    }
    true
  }

  /**
   * Recursively resolve the type of an Expression
```

```
 */
def static TypeEnum valueResolver(Expression it) {
  var t = TypeEnum.STRING_TYPE;

  if(it instanceof StringValue)
    t = TypeEnum.STRING_TYPE
  if(it instanceof IntegerValue)
    t = TypeEnum.INTEGER_TYPE
  if(it instanceof BooleanValue)
    t = TypeEnum.BOOLEAN_TYPE
  if(it instanceof BinaryConstraint) {
    switch ((it as BinaryConstraint).operator) {
    case ADDITION:
      t = TypeEnum.INTEGER_TYPE
    case AND:
      t = TypeEnum.BOOLEAN_TYPE
    case EQUAL:
      t = TypeEnum.BOOLEAN_TYPE
    case GREATER:
      t = TypeEnum.BOOLEAN_TYPE
    case LESS:
      t = TypeEnum.BOOLEAN_TYPE
    case MULTIPLICATION:
      t = TypeEnum.INTEGER_TYPE
    case OR:
      t = TypeEnum.BOOLEAN_TYPE
    case SUBSET:
      t = TypeEnum.BOOLEAN_TYPE
    }
  }
  if(it instanceof Set) {
    val element = (it as Set).has.get(0)
    if(element instanceof StringValue)
      t = TypeEnum.STRING_TYPE
    if(element instanceof IntegerValue)
      t = TypeEnum.INTEGER_TYPE
    if(element instanceof BooleanValue)
      t = TypeEnum.BOOLEAN_TYPE
  }
  if(it instanceof UnaryConstraint) {
    t = TypeEnum.BOOLEAN_TYPE
  }
  t
}
```

```
  /**
   * Make sure that the expression of a unary constraint is a BooleanValue
   */
  def static boolean constraintUnary(UnaryConstraint it) {
    valueResolver(expression).equals(TypeEnum.BOOLEAN_TYPE)
  }

  /**
   * Check that a set is not empty and all values have the same type
   */
  def static boolean constraintSet(Set it) {
    has.size > 0 && has.fold(true) [ b, v | b && v.class == has.get(0).class ]
  }
}
```

Concrete syntax

```
Root returns Root:
      {Root}
      (configurators+=Configuration)*
      (expressions+=Expression)*;

Expression returns Expression:
      BinaryConstraint | Parameter | Set | StringValue |
      IntegerValue | BooleanValue | UnaryConstraint;

Value returns Value:
      StringValue | IntegerValue | BooleanValue;

Configuration returns Configuration:
      {Configuration}
      'cfg' (assignments+=Assignment ( "," assignments+=Assignment)*)?;

Assignment returns Assignment:
      'assign' parameter=[Parameter|EString] 'to' value=[Value|EString];

Parameter returns Parameter:
      type=TypeEnum name=EString;

EString returns ecore::EString:
      STRING | ID;

enum TypeEnum returns TypeEnum:
```

```
      IntegerType = 'int' | BooleanType = 'boolean' | StringType =
'string';

StringValue returns StringValue:
      type=TypeEnum name=EString '=' value=EString;

IntegerValue returns IntegerValue:
      type=TypeEnum name=EString '=' value=EInt;

BooleanValue returns BooleanValue:
      type=TypeEnum name=EString '=' value=EBoolean;

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

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

BinaryConstraint returns BinaryConstraint:
      'bc' name=EString left=[Expression|EString] operator=BinaryOperators
right=[Expression|EString] |
      'bc' root=EBoolean name=EString left=[Expression|EString]
operator=BinaryOperators right=[Expression|EString];

Set returns Set:
      'set' name=EString '=' ('[' has+=[Value|EString] ( ","
has+=[Value|EString])* ']' )?;

UnaryConstraint returns UnaryConstraint:
      'uc' name=EString Operator=UnaryOperators
expression=[Expression|EString] |
      'uc' root=EBoolean name=EString Operator=UnaryOperators
expression=[Expression|EString];

enum BinaryOperators returns BinaryOperators:
      less = '<' | greater = '>' | equal = '==' | multiplication = '*' |
addition = '+' | subset = 'subset' | and = '&&' | or = '||';

enum UnaryOperators returns UnaryOperators:
not = '!';
```

Back-ends

Code generation

The Java application uses a class ExpressionHolder which is automatically generated from a configuration file (.cfdgsl).
This contains the instantiation of all expression in the given model and all parameters. Here is an example of the instantiation of the StringValues. All the values are looped through xtext and all properties which needs to be set are then set for each value. This is done the same way for all expressions.

```
ConfiguratorPackageFactory factory = ConfiguratorPackageFactoryImpl.init();
Map<String, Value> values = new HashMap<String, Value>();
HashMap<String, Expression> constraintMap = new HashMap<String, Expression>();
expressions = new ArrayList<Expression>();
StringValue s;
«FOR expr : it.expressions.filter(typeof(StringValue))»
   s = factory.createStringValue();
   s.setName("«expr.name»");
   s.setType(TypeEnum.get("«expr.type»"));
   s.setValue("«expr.value»");
   expressions.add(s);
   values.put("«expr.name»", s);
   constraintMap.put("«expr.name»",s);
«ENDFOR»
```

The only expressions which require special treatment is the BinaryConstraint and the UnaryConstraint. These two types have properties which refer to other expressions(left right, and expression). These cannot be immediately set as the values may not yet have been instantiated if it is a Binary- or UnaryConstraint. Instead a new StringValue expression is created as placeholders with the name of the expression which should be there.

```
«FOR expr : it.expressions.filter(typeof(BinaryConstraint))»
   bc = factory.createBinaryConstraint();
   bc.setName("«expr.name»");
   bc.setOperator(BinaryOperators.«expr.operator.toString().toUpperCase»);
   bc.setRoot(«expr.root»);
   r = factory.createStringValue();
   r.setName("«expr.right.name»");
   l = factory.createStringValue();
   l.setName("«expr.left.name»");
   bc.setRight(r);
   bc.setLeft(l);
   constraintMap.put("«expr.name»", bc);
   expressions.add(bc);
```

«**ENDFOR**»

After the loops through the BinaryConstraint and UnaryConstraints, a new, statically typed loop, goes through all BinaryConstraint and UnaryConstraints and replace the StringValue placeholder with the actual expression by looking the name up in a hashmap from String Name to Expression Expression.

```
for (Map.Entry<String, Expression> entry : constraintMap.entrySet()) {
    Expression e = entry.getValue();
    if(e instanceof BinaryConstraint) {
        BinaryConstraint localbc = (BinaryConstraint) e;
        localbc.setLeft(constraintMap.get(localbc.getLeft().getName()));
        localbc.setRight(constraintMap.get(localbc.getRight().getName()));
    } else if(e instanceof UnaryConstraint) {
        UnaryConstraint localuc = (UnaryConstraint) e;
        localuc.setExpression(constraintMap.get(localuc.getExpression().getName())); } }
```

Java Validator

The validator class verifies that the user's inputs correctly satisfies the configurator. At first it checks that all the user inputs are of the correct type, eg. String in StringValues. Afterwards it start the main loop, which goes through all the Binary- and UnaryConstraints in the configurator. For each constraint with the root property set to true, it starts a recursive method to determine if the constraint are satisfied.

```
for (Expression e : expressions) {
    if (e instanceof BinaryConstraint) {
        BinaryConstraint b = (BinaryConstraint) e;
        if (b.isRoot()) {
            Expression res = validate((BinaryConstraint) e, map);
            if (res instanceof BooleanValue) {
                BooleanValue bv = (BooleanValue) res;
                return bv.isValue();
            } else {
                throw new RuntimeException(“”); } } } }
```

There are 3 methods involved in the recursive process: Validate(BinaryConstraint, ...), Validate(UnaryConstraint, ...) and getExpr(...). The two Validate methods are called when it needs to validate a constraint corresponding to the input of the method. The getExpr method is called whenever it needs to be determined what kind of expression right and left contains. This is done as some expressions require specific action at this point.

```
private static Expression getExpr(Expression expr, Map<String, Assignment> map) {
    Expression e;
    if (expr instanceof BinaryConstraint) {
        BinaryConstraint bc = (BinaryConstraint) expr;
```

```
        e = validate(bc, map);
    } else if (expr instanceof UnaryConstraint) {
        UnaryConstraint uc = (UnaryConstraint) expr;
        e = validate(uc, map);
    } else if (expr instanceof Parameter) {
        Parameter p = (Parameter) expr;
        Assignment a = map.get(p.getName());
        e = a.getValue();
    } else {
        e = expr;
    }
    return e; }
```

After the binaryconstraint-validate method has retrieved the two expressions in left and right, it continues to switch on the binary-operator which it uses. This switch contains all the binary-operators in the model and their behavior. It is written in a defensive manner, such that all operators make sure that the expressions are of the correct type/s for the operator.

```
BooleanValue b = config.createBooleanValue();
switch (bc.getOperator()) {
    case EQUAL:
        if (left instanceof IntegerValue && right instanceof IntegerValue) {
            IntegerValue l = (IntegerValue) left;
            IntegerValue r = (IntegerValue) right;
            b.setValue(l.getValue() == r.getValue());
        } // Almost the same for BooleanValue & StringValue
        else
            throw new RuntimeException();
        return b;
    case LESS: …
    ...
    b.setValue(false);
return b;
```

The unary-validator works in the same way, the switch, however, only contains the one unary-operator which exists in the system now.

The validator is written to be easy to extend such that changes in the model requires minimal amount of work. Only the addition of completely new constraints, such as ternary constraints, and more advanced types require considerable amount of work. Similarly, in the code generation, is it written with focus on extensibility.

PHP

Our interpreter client is written in PHP and constructs a GUI based on JSON, where the user may input parameters. They are then validated against the validator functions. How we construct the GUI is simply a matter of iterating over the JSON's parameters. We will not be displaying this in the report.

The checkParameters function checks that all parameters in the JSON is actually submitted in the form, and that they can be interpreted as a bool/numeric/string. The code only shows the check for strings, but it is pretty much the same for bool/int.

```
private function checkParameters()
 {
        $res = true;
        if (isset($this->json->root->expressions->parameter)) {
        foreach ($this->json->root->expressions->parameter as $param) {
        //Check that an assignment is in the post request
        $res = $res && isset($this->post[$param->name]);
        if ($param->type == "StringType" && trim($this->post[$param->name]) == '') {
          echo $this->post[$param->name] . " is not a non-empty string value <br/>";
          return false;
        }
        return true;
 }
```

The validateBinary checks all the binary constraints recursively by the use of the getExpr method. It returns a stdClass with the fields: name, type, and value.
After this it switches on the operator, and returns the appropriate type (int/boolean) depending on the operator. Some of the cases are left out, as they are very much alike. Among those are: greater, equals, addition, and, or, and set.

```
 private function validateBinary($binConstraint){
        $left = $this->getExpr($binConstraint->left)["element"];
        $leftType = $this->getExpr($binConstraint->left)["type"];

        $obj = new stdClass();
        $obj->name = "";
        switch($binConstraint->operator) {
        case "less":
        if($leftType == "IntegerType" && $rightType == "IntegerType") {
          $leftValue = (int) $left->value;
          $rightValue = (int) $right->value;
          $res = $leftValue < $rightValue;
          $obj->value = $res;
          $obj->type = "BooleanType";
          return $obj;
        }
```

```
        return $obj;
        }
 }
```

Validate unary is more hardcoded in the sense that only the negation is currently possible, and thus we are only working with boolean types.
For binary and unary constraints it simply calls the appropriate validate method, and return the negation of that.

```
private function validateUnary($unaryConstraint)
 {
        $left = $this->getExpr($unaryConstraint->expression)["element"];
        $leftType = $this->getExpr($unaryConstraint->expression)["type"];
        $obj = new stdClass();
        $obj->name = "";
        switch($leftType) {
        case "BooleanType" :
        if($this->boolval($left->value) == false) {
          $left->value = true;
        }
        else
          $left->value = false;
        $obj->value = $left->value;
        $obj->type = "BooleanType";
        return $obj;
        }
 }
```

The getExpr method is responsible for scanning all the expressions in the JSON. For parameters it is responsible for returning the parameter type, and the value that the user has inputted. For binary/unary/set types it simply resolves the name of the constraint to the actual constraint object, and the type, packed into an associative array. For values it does the same as parameter, except that the value is already given, and not taken from the GUI input.
For this reason, values, sets, unary constraints and part of the parameters (only integer is shown) are omitted.

```
private function getExpr($exprName){
        $expressions = $this->json->root->expressions;
        $ret = array("element" => "", "type" => "");
        if(isset($expressions->binaryConstraints)) {
          foreach ($expressions->binaryConstraints as $bc) {
            if ($bc->name == $exprName) {
              $ret["type"] = "BinaryConstraint";
              $ret["element"] = $bc;
              $res = $this->validateBinary($bc);
```

```
            $ret["type"] = $res->type;
            $ret["element"] = $res;
            return $ret;
          } } }
        if(isset($expressions->parameter)) {
          foreach ($expressions->parameter as $param) {
            if ($param->name == $exprName) {
              $value = $this->post[$param->name];
              if (is_numeric($value)) {
                $obj = new stdClass();
                $obj->value = $value;
                $obj->name = $param->name;
                $obj->type = "IntegerType";
                $ret["type"] = "IntegerType";
                $ret["element"] = $obj;
                return $ret;
          } } }
        return $ret;
}
```

Test strategy

To test that the applications and the DSL works as intended, test cases for both the constraints, concrete syntax and the integrated system have been created.
The test strategy for the constraints is to test passes and failures of each validation function, such that all outcomes of all functions have been reached, but not necessarily all code branches.
The concrete syntax has been tested with positive automated JUnit tests. The purpose of the automated tests is to check that the concrete syntax is correctly translated to dynamic instances.
The test strategy for the integrated system, which is concrete syntax, DSL, constraints, code generator and the two applications, is to make pairs of test cases that expects successful and unsuccessful validations of similar tests.
The purpose of these is to test a very specific functionality, such that whether a specific operator in a binary constraint behaves as expected. We included additional tests that combines the different functionalities of the DSL, as the code generator and interpreter has more complex implementation with greater internal functional dependency than the constraints. With combined tests that involves multiple parameters and outcomes, we did manual testing of the applications to reach all possible branches of the specific configurator.

Test cases

Test 1

```
int width
int height
int length
string name
boolean convex
int maxVolume = 10000
int two = 2
string reservedName = "box"

uc true nameIsNotReserved ! nameIsReserved
bc nameIsReserved name == reservedName

bc true masterVolumeConstraint standardVolumeIfNotConvex || increaseVolumeIfConvex
bc standardVolumeIfNotConvex notConvex && volumeConstraint
bc increaseVolumeIfConvex convex && doubleVolumeConstraint

bc volumeConstraint volume < maxVolume
bc doubleVolumeConstraint volume < doubleVolume

bc doubleVolume maxVolume * two

bc groundSurface width * length
bc volume groundSurface * height

uc notConvex ! convex
```

Test 2

```
int wheelSize
int cassetteSpeed
int frameSize
string bikeType

string racer = "racer"
string tt = "tt"
string cx = "cx"
string track = "track"
string bmx = "bmx"
int racerDiameter1 = 650
int racerDiameter2 = 700
int BMXMaxFrameSizePlusOne = 50
int frameSize48 = 48
int frameSize50 = 50
int frameSize52 = 52
int frameSize54 = 54
int frameSize55 = 55
int frameSize56 = 56
int frameSize58 = 58
int frameSize60 = 60
int frameSize62 = 62
int cassetteSpeed1 = 1
int cassetteSpeed6 = 6
int cassetteSpeed7 = 7
int cassetteSpeed8 = 8
int cassetteSpeed9 = 9
int cassetteSpeed10 = 10
int cassetteSpeed11 = 11

set cassetteSpeeds = [cassetteSpeed1, cassetteSpeed6, cassetteSpeed7,
cassetteSpeed8, cassetteSpeed9, cassetteSpeed10, cassetteSpeed11]
set frameSizes = [frameSize48, frameSize50, frameSize52, frameSize54, frameSize55
            , frameSize56, frameSize58, frameSize60, frameSize62]


bc true frameSizeAndCassetteSpeedValid frameSizeIsValid && cassetteSpeedIsValid
bc frameSizeIsValid frameSize subset frameSizes
bc cassetteSpeedIsValid cassetteSpeed subset cassetteSpeeds
bc true masterType isRacerTTCXOrTrack || isBMX

bc true isNotTTOrIsNotSingleSpeed isNotTT || notSingleSpeed

bc true BMXWithProperFrame isNotBMX || BMXFrame
bc BMXFrame frameSize < BMXMaxFrameSizePlusOne

bc true racerWithProperDiameter isNotRacer || racerDiameter
bc racerDiameter racerDiameterBC1 || racerDiameterBC2
bc racerDiameterBC1 wheelSize == racerDiameter1
```

```
bc racerDiameterBC2 wheelSize == racerDiameter2

bc true singleSpeedTrackOrBMX singleSpeedComposition || isNotBMXOrTrack
bc singleSpeedBMX isBMX && singleSpeed
bc singleSpeedTrack isTrack && singleSpeed
bc singleSpeedComposition singleSpeedBMX || singleSpeedTrack
bc singleSpeed cassetteSpeed == cassetteSpeed1
uc notSingleSpeed ! singleSpeed

bc isRacerTTCXOrTrack isRacerOrTT || isCXOrTrack
bc isRacerOrTT isRacer || isTT
bc isCXOrTrack isCX || isTrack
bc isBMXOrTrack isBMX || isTrack
bc isRacer bikeType == racer
bc isTT bikeType == tt
bc isCX bikeType == cx
bc isTrack bikeType == track
bc isBMX bikeType == bmx
uc isNotBMXOrTrack ! isBMXOrTrack
uc isNotRacer ! isRacer
uc isNotBMX ! isBMX
uc isNotTT ! isTT
```

Test 3

```
int weight
int length
int height

string material

string wood = "wood"
string metal = "metal"
string stone = "stone"

set materials = [wood, metal, stone]

int woodDensity = 80
int metalDensity = 500
int stoneDensity = 350

bc true isValidMaterial material subset materials

bc true correctWeightIfWood isNotWood || correctWeightWood
bc correctWeightWood calculatedWeightWood == weight
bc calculatedWeightWood area * woodDensity

bc true correctWeightIfMetal isNotMetal || correctWeightMetal
bc correctWeightMetal calculatedWeightMetal == weight
```

```
bc calculatedWeightMetal area * metalDensity

bc true correctWeightIfStone isNotStone || correctWeightStone
bc correctWeightStone calculatedWeightStone == weight
bc calculatedWeightStone area * stoneDensity

bc area length * height

bc isWood material == wood
bc isMetal material == metal
bc isStone material == stone

uc isNotWood ! isWood
uc isNotMetal ! isMetal
uc isNotStone ! isStone
```