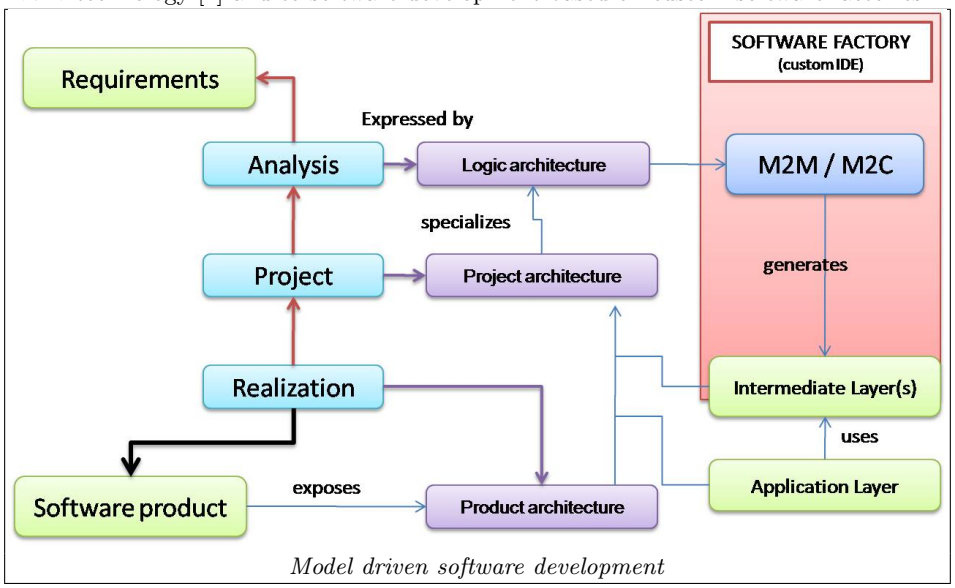
# Expressions: languages and models

Antonio Natali

Alma Mater Studiorum – University of Bologna
viale Risorgimento 2, 40136 Bologna, Italy
antonio.natali@unibo.it

**Abstract.** In this work we define the semantics of expressions by means of `M2M` (model-to-model) and `M2C` (model-to-code) transformations. The work is also an introduction to the usage of the `Xtext` technology [4] and to software development based on custom software factories.




*Model driven software development*

**Keywords:** meta-model, `MOF`, `Ecore`, `Xtext`, `Xtend`, semantics, model to code transformer, model-to-model mapping

# 1 Introduction

This work is about the definition of semantic models of arithmetic expressions (see the paper of Harel and Rumpe [1]) first defined as an object tree and then as a network of active entities (operators, sensors, etc.) that interact by exchanging messages.

To fully understand the content that follows, the reader should also consult the `Xtext` documentation [5] the book [3] at pg. 33, and [2].

# 2 A first meta-models for expressions

Traditionally, arithmetic expressions can be written according to the following syntax (the symbols included in '' are terminals):

Syntax of expressions

```
E ::= T | E '+' T | E '-' T
T ::= F | T '*' F | T '/' F
F ::= N | '(' E ')' | S
N ::= ( '0'..'9' )+
S ::= 's' ('0'..'9')* ;
```

The sub-language starting from the non-terminal `S` denotes here a *sensor*, defined as an entity able to capture information from the world and to make it available to other computational entities.

To formally define the syntax of expressions, we can use the `Xtext` framework (see [5] and [2]) by creating a `Xtext` project named *it.unibo.indigo.exp*.

The meta-model is a language built according to a proper set of rules expressed in `EBNF` notation like the previous ones, but rewritten so to remove left-recursive definitions:

Syntax of Expressions without left-recursive rules

```
E ::= T { RE }
RE ::= '+' T | '-' T
T ::= F  { RT }
RT ::= '*' F | '/' F
F ::= N | '(' E ')' | S
N ::= ( '0'..'9' )+
S ::= 's' ('0'..'9')* ;
```

An expression (`E`) is a term (`T`) followed (optionally) by a rest (`RE`) constituted by a list of elements of the form `+T` or `-T`. Each term is a factor (`F`) followed (optionally) by a rest (`RT`) constituted by a list of elements of the form `*F` or `/F`. Finally a factor is a number (`Num`), a sensor (`S`), or an expression included between a pair of parenthesis `( )`.

According to the rules above:

Examples of well-formed expression sentences

```
2 + 3 * 5
( 2+ 3 / (( 5 ))
s1-s2*3
```

## 2.1 Syntax rules in `Xtext`

The `Xtext` definition of the language is:

```
1  grammar it.unibo.xtext.Exp with org.eclipse.xtext.common.Terminals
2  generate exp "http://www.unibo.it/xtext/Exp"
3  //ARGUMENTS -Xmx512m -Xmx384m -XX:MaxPermSize=128m
4
5  EL : "exp" name=ID "=" exp=E ;
6
7  E  :  term=T ( re += RE )* ;
8  RE : op='+' term=T | op='-' term=T ;
9
10 T : factor=F ( rt += RT )* ;
11 RT  : op='*' factor = F | op='/' factor = F;
12
13 F : num=Num | "(" exp=E ")" | s = S;
14
15 Num : val=INT ;
16
17 //Sensor
18 S : id = SID ;
19 terminal SID : 's' ('0'..'9')* ;
```

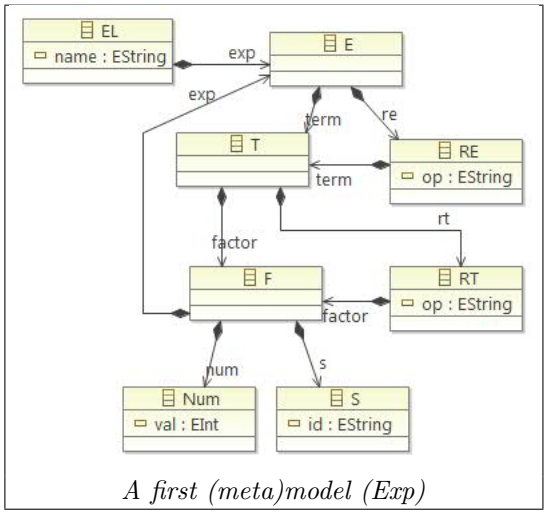
**Listing 1.1.** `Exp.xtext`

An Expression Element (`EL`) is composed by a name (`ID`) followed by an Expression (`E`). According to the rules above:

`Examples of well-formed Exp expression sentences`

```
exp e0=2 + 3 * 5
exp e1 =( 2+ 3 / (( 5 ))
exp e2 = s1-s2*3
```

After the `Xtext` generation process, we can find, in the *src-gen/it.unibo.* directory, the file named `Exp.ecore` that can be showed also as a diagram in `UML` style.




*A first (meta)model (Exp)*

**Note:** To exploit the graphical tools, it could be necessary to download a plugin including a `ECore Diagram Editor`

## 2.2 Custom Validation Rules

One of the main advantages of meta-modelling is the possibility to statically validate domain specific constraints. Let us suppose, for example, that the name of an expression sentence starts with an lower-case letter. This constraint can be expressed by inserting proper rules in a proper `ExpJavaValidator.java` file:

```java
/**
 * AN Unibo-DISI
 */
package it.unibo.xtext.myvalidation;

import it.unibo.xtext.exp.EL;
import it.unibo.xtext.exp.ExpPackage;
import it.unibo.xtext.validation.AbstractExpValidator;
import org.eclipse.xtext.validation.Check;

public class ExpJavaValidator extends AbstractExpValidator {
/*
 * Expression names must start with a lowerCase letter
 */
    @Check
    public void checkNameStartsWithLowChar(EL el) {
        if (!Character.isLowerCase(el.getName().charAt(0))) {
            error("Name cannot start with a capital",
                    ExpPackage.Literals.EL__NAME);
        }
    }

/*
* An expression name should not be 'exp'
*/
    @Check
    public void checkWrongName(final EL el) {
        if (el.getName().equals("exp")) {
            warning("Name should be different from exp",
                    ExpPackage.Literals.EL__NAME);
        }
    }
}
```

**Listing 1.2.** `ExpJavaValidator.java`

The `@Check` annotation advises the framework to use the method as a validation rule. If the name starts with a upper case letter, an error will be attached to the name of the expression sentence (and no generation process is started). If the name "`expr`" is used [1], a warning will be attached to the name of the expression sentence.

`Java` code of this kind can be automatically generated by specifications written in `Xtend`:

```
/*
 * generated by Xtext, modified by AN Unibo-DISI
 */
package it.unibo.xtext.validation
import org.eclipse.xtext.validation.Check
import it.unibo.xtext.exp.EL
import it.unibo.xtext.exp.ExpPackage

/**
 * Custom validation rules.
 *
 * see http://www.eclipse.org/Xtext/documentation.html#validation
 */
class ExpValidator extends AbstractExpValidator {
@Check
def void checkNameStartsWithLowChar(EL el) {
    if (!Character.isLowerCase(el.getName().charAt(0))) {
        error("Name cannot start with a capital",
                ExpPackage.Literals.EL__NAME);
    }
}

@Check
def checkWrongName(EL el) {
    if ( el.getName().equals("expr")) {
        warning("Name should be different from expr",
```

[1] The usage of the `exp` word is avoided by the system itself since it is a keyword

```
            ExpPackage.Literals.EL__NAME);
    }
}
}
```

**Listing 1.3.** `ExpValidator.xtend`

An Xtend class resides in a plain Eclipse Java project. As soon as the SDK is installed, Eclipse will automatically translate all the classes to Java source code. By default you will find it in a source folder `xtend-gen`.

## 3 Another meta-model for expressions

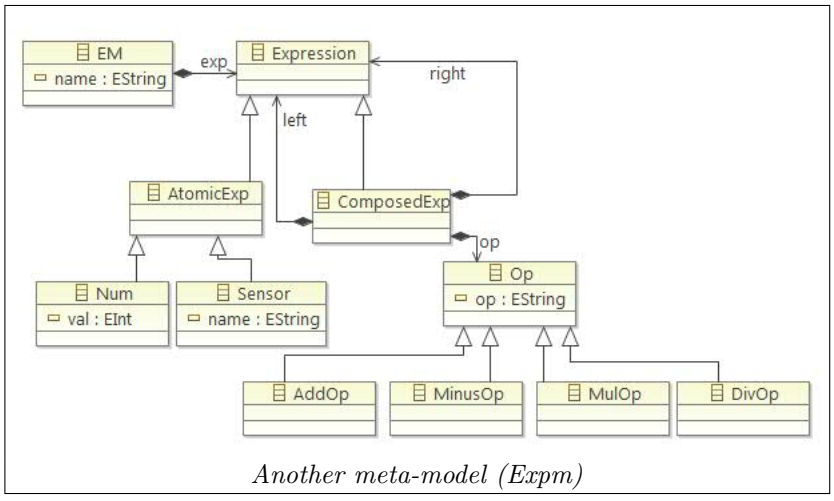
The definition of a semantic model of expressions could start from the idea that (from the structural point of view) expressions can be of two main types:

- *atomic expressions* : the simple values;
- *composed expressions*: expressions including operators.

Since we have only binary operators, composed expressions can be represented by using a binary tree; this tree can be the result of the processing of a new `Xtext` language (called here `expm`, defined in a project named *it.unibo.expm*) defined as follows:

```
grammar it.unibo.xtext.Expm with org.eclipse.xtext.common.Terminals
generate expm "http://www.unibo.it/xtext/Expm"
//ARGUMENTS -Xmx512m -Xmx384m -XX:MaxPermSize=128m

EM : "exp" name=ID "=" exp=Expression ;

Expression : AtomicExp | ComposedExp ;
ComposedExp : op=Op left=Expression right=Expression ;

AtomicExp : Num | Sensor ;

Num : val=INT ;
Op : AddOp | MulOp | DivOp | MinusOp ;

AddOp   : op='+' ;
MinusOp : op='-' ;
MulOp   : op='*' ;
DivOp   : op='/' ;

//Added for interaction
Sensor : name=ID ;
```

**Listing 1.4.** `Expm.xtext`




*Another meta-model (Expm)*

According to the rules, expression must be written in prefix form:

Examples of Expm expression sentences

```
exp e0= + 2 * 3 5
exp e1 = ( + 2 3 ) / (( 5 ))
exp e2 = - s1 * s2 3
```

Expression-handling rules can be now defined in a more simple way by means of the `expm` meta-model. Moreover, the `expm` version of an expression can be obtained by a model-to-model (`M2M`) transformer (see Section 5) that builds in automatic way the `expm-model` by starting from an instance of the `exp` meta-model. Finally, a user-defined `IDE` for expressions (let us call it `exp-IDE`) can exploit the `expm` binary-tree meta-model to introduce the proper custom semantics, including the view of an expression as a distributed computational network.

The following sections will explore these topics in more detail. We will introduce custom (code) generators structured according to what described in [2]. All the generators are defined within the project *it.unibo.exp* (that must depend on *it.unibo.expm*).

## 4 Working with models in Java

At this stage of the work, let us define some expression-evaluation object as a "classical" interpreter that takes in input a binary tree expressed as a `expm.Espression` and works according the *eval-apply* pattern. The interpreter can be written in two ways: as a `Java` class or as a set of `Xtend` rules.

The project *it.unibo.exp.tests*) includes the evaluators and some test for the evaluation of expressions represented as instances of the classes generated by `Xtext` from our custom language/meta-model.

### 4.1 An evaluation object (in `Java`)

Our first interpreter is expressed as a `Java` class working on binary expressions of type `Expm`.

```
1  package it.unibo.evaluator;
2  import it.unibo.xtext.expm.*;
3
4  public class EvalExpm {
5  protected EvalSensor evalSensor = new EvalSensor();
6
7      public double eval(String exp, Expression e) throws Exception {
8          evalSensor.setExp(exp);
9          return eval(e);
10     }
11     public double eval(Expression e) throws Exception {
12         if (e instanceof it.unibo.xtext.expm.Num) return eval((it.unibo.xtext.expm.Num) e);
13         if (e instanceof Sensor) return eval((Sensor) e);
14         if (e instanceof ComposedExp) return eval((ComposedExp) e);
15         throw new Exception("Wrong expression type");
16     }
17     public double eval(it.unibo.xtext.expm.Num e) {
18         return e.getVal();
19     }
20     public double eval(Sensor e) throws Exception {
21         return evalSensor.getVal(e.getName());
22     }
23     public double eval(ComposedExp e) throws Exception {
24         return apply(e.getOp(), eval(e.getLeft()), eval(e.getRight()));
25     }
26     public double apply(Op op, double left, double right) throws Exception {
27         if (op instanceof AddOp) return left + right;
28         else if (op instanceof MinusOp) return left - right;
29         else if (op instanceof MulOp) return left * right;
30         else if (op instanceof DivOp) return left / right;
31         else throw new Exception("Wrong argument");
32     }
33 }
```

**Listing 1.5.** `EvalExpm.java`

The evaluation of (expressions including) sensors is deferred to Subsection 4.5.

### 4.2 An evaluation object (in `Xtend`)

A conventional eval-apply interpreter can be written in `Xtend` in a more concise way thanks to *Polymorphic Dispatch*:

```
1  package it.unibo.evaluator
2  import it.unibo.xtext.expm.*
3
4  class EvalExpmXtend {
5  var EvalSensor evs = new EvalSensor()
6
7  def dispatch double eval(Expression e){ 0 } //Should not be here
8  def dispatch double eval(Num e){ e.^val }
9  def dispatch double eval(Sensor e){ evs.getVal(e.name) }
10 def dispatch double eval(ComposedExp e) {apply(e.op, eval(e.left), eval(e.right) )}
11
12 def dispatch double apply(Op op, double left, double right){ return 0;}//Should not be here
13 def dispatch double apply(AddOp op, double left, double right){ return left+right; }
14 def dispatch double apply(MinusOp op, double left, double right){ return left-right; }
15 def dispatch double apply(MulOp op, double left, double right){ return left*right; }
16 def dispatch double apply(DivOp op, double left, double right){ return left/right; }
17 }
```

**Listing 1.6.** `EvalExpmXtend.xtend`

### 4.3 Building expressions

In order to create a `Java`(`Ecore`) model of an expression starting from a well-formed expression sentence (String), the test-designer has included in the project *it.unibo.exp.tests*) the utility `Java` class named `ModelUtil` that can be reused by application designers when they need to work with expressions at application level.

```
1  package it.unibo.evaluator;
2  import java.io.StringReader;
3  import org.eclipse.emf.ecore.EObject;
4  import org.eclipse.xtext.parser.IParseResult;
5  import org.eclipse.xtext.parser.IParser;
6  import com.google.inject.Injector;
7  import it.unibo.xtext.ExpStandaloneSetup;
8  import it.unibo.xtext.ExpmStandaloneSetup;
9  import it.unibo.xtext.exp.EL;
10 import it.unibo.xtext.expm.EM;
11 import it.unibo.xtext.expm.Expression;
12 import it.unibo.xtext.mygenerator.ExpToExpm;
13
14 public class ModelUtil {
15
16    /*
17     * Create a Expm model
18     *      expS should be an expression written in prefix form like
19     *      exp e0 = + 2 3
20     */
21    public EM createExpmFromParser(String expS) throws Exception {
22        Injector injector = new ExpmStandaloneSetup()
23                .createInjectorAndDoEMFRegistration();
24        IParser parser = injector.getInstance(IParser.class);
25        IParseResult result = parser.parse(new StringReader(expS));
26        if (!result.getSyntaxErrors().iterator().hasNext()) {// NO errors
27            EObject eRoot = result.getRootASTElement();
28            return (EM) eRoot;
```

```
29          } else
30              throw new Exception("syntax errors in expression " + expS);
31      }
32
33      /*
34      * Create a Exp model :
35      *       expS should be an expression written in infix form like
36      *       exp e0 = 2 + 5
37      */
38      public EL createExpFromParser(String expS) throws Exception {
39          Injector injector = new ExpStandaloneSetup()
40                  .createInjectorAndDoEMFRegistration();
41          IParser parser = injector.getInstance(IParser.class);
42          IParseResult result = parser.parse(new StringReader(expS));
43          if (!result.getSyntaxErrors().iterator().hasNext()) {// NO errors
44              EObject eRoot = result.getRootASTElement();
45              return (EL) eRoot;
46          } else
47              throw new Exception("syntax errors in expression " + expS);
48      }
49
50
51      /*
52       * Create an object of type Expression
53       *      expS should be an expression written in infix form like
54       *      exp0 = 2 + 3 * 5
55       */
56      public Expression createExpressionFromParser(String expS) throws Exception {
57          return createExpression(createExpFromParser(expS));
58      }
59      public Expression createExpression(EL expSentence) {
60          return new ExpToExpm().mapToExpm(expSentence.getExp());
61      }
62
63  }
```

**Listing 1.7.** `ModelUtil.java`

The operation `createExpmFromParser` exploits the generated class *it.unibo.xtext.ExpmStandaloneSetup* to get an instance of the `expm` parser (the generated class *it.unibo.xtext.parser.antlr.ExpmParser.java*). The parser is used to generate the `AST` whose root of class `EM` is returned to the caller.

The operation `createExpFromParser` exploits the generated class *it.unibo.xtext.ExpStandaloneSetup* to get an instance of the `exp` parser (the generated class *it.unibo.parser.xtext.antlr.ExpParser.java*). The parser is used to generate the `AST` whose root f class `EL` is returned to the caller.

Moreover:

- The operation `Expression createExpression(EL expSentence)`: converts an `exp.EL` model into a `expm.expression` model;
- The operation `Expression createExpressionFromParser(String s)`: returns an object (model) of type `expm.Expression` given an expression written in the `exp` prefix syntax.

The class `ExpToExpm` provides an operation `mapToExpm` to convert an expression of type `exp.E` into and expression of type `expm.Expression`. It is an example of model-to-model (`M2M`) mapping discussed in Section 5.

### 4.4 Building and evaluating expressions

The project *it.unibo.exp.tests* includes tests for the evaluation of expressions:

```
1  package it.unibo.xtext.exp.tests;
2  import static org.junit.Assert.*;
3  import it.unibo.evaluator.EvalExpm;
4  import it.unibo.evaluator.EvalExpmXtend;
5  import it.unibo.evaluator.ModelUtil;
6  import it.unibo.xtext.expm.*;
```

```
7  import it.unibo.xtext.exp.*;
8
9  import org.junit.Test;
10
11 public class TestExpm {
12     protected EvalExpm evaluator        = new EvalExpm();
13     protected EvalExpmXtend evaluatorXtend = new EvalExpmXtend();
14     protected ModelUtil modelUtil       = new ModelUtil();
15
16 /*
17  * The value of expression  exp e0 = + 2 3 should be 5.0
18  * as result of both the Java and the Xtend evaluators
19  */
20     @Test
21     public void testExpmNoSensors() {
22         try {
23             String expms = "exp e0 = + 2 3";
24             EM em = modelUtil.createExpmFromParser(expms);
25             double res = evaluator.eval(em.getExp());
26             double res1 = evaluatorXtend.eval(em.getExp());
27             assertTrue("testExpmNoSensors", res == 5.0 && (res == res1));
28         } catch (Exception e) {
29             fail("testExpmNoSensors " + e.getMessage());
30         }
31     }
32
33     /*
34      * The value of expression  exp e0 = + - - s2 s3 s2 s3 should be 0.0
35      * as result of both the Java and the Xtend evaluators
36      * since the values of the sensors s1, s2 s 3 are set to 1.0 (see EvalSensor.java)
37      */
38     @Test
39     public void testExpWithSensors() {
40         try {
41             // exp e0=s2-s3-s2+s3
42             String expms = "exp e0 = + - - s2 s3 s2 s3";
43             EM em = modelUtil.createExpmFromParser(expms);
44             double res = evaluator.eval(em.getExp());
45             double res1 = evaluatorXtend.eval(em.getExp());
46             assertTrue("testExpmWithSensors", res == 0.0 && (res == res1));
47         } catch (Exception e) {
48             fail("testExpmWithSensors " + e.getMessage());
49         }
50     }
51 }
```

**Listing 1.8.** `TestExpm.java`

### 4.5 Evaluating expressions with sensors

In order to evaluate expressions including sensor names, the test designer defines a *mock* object that defines some predefined sensor and asks the user for the values of non-predefined sensors:

```
1  package it.unibo.evaluator;
2
3  public interface IEvalSensor {
4      public double getVal( String sensorName ) throws Exception;
5  }
```

**Listing 1.9.** `IEvalSensor.java`

```
1  package it.unibo.evaluator;
2  import it.unibo.baseEnv.basicFrame.EnvFrame;
3  import it.unibo.is.interfaces.IBasicEnvAwt;
4  import java.util.Hashtable;
5  import java.awt.Color;
6  import java.io.*;
7  import javax.swing.JPanel;
8
```

```
9  public class EvalSensor implements IEvalSensor {
10     protected Hashtable<String, Double> symTab = new Hashtable<String, Double>();
11     protected IBasicEnvAwt env;
12     protected boolean initdone = false;
13     protected String exp = "";
14
15     public EvalSensor() {
16         symTab.put("s1", 1.0);
17         symTab.put("s2", 1.0);
18         symTab.put("s3", 1.0);
19     }
20
21     protected void initEnv() {
22         env = new EnvFrame("Sensor evalutation " + exp, null, Color.white,
23                 Color.blue);
24         env.init();
25         initdone = true;
26     }
27
28     public void setExp(String exp) {
29         this.exp = exp;
30     }
31
32     public double getVal(String sensorName) throws Exception {
33         if (symTab.containsKey(sensorName))
34             return symTab.get(sensorName);
35         else {
36             // Double val = readFromIn(sensorName); //First prototype
37             Double val = readFromInputPanel(sensorName);
38             symTab.put(sensorName, val);
39             return val;
40         }
41     }
42
43     protected Double readFromInputPanel(String sensorName) {
44         if (!initdone)
45             initEnv();
46         env.writeOnStatusBar("Exp=" + exp, 24);
47 //      SensorInputPanel inpp = new SensorInputPanel("value of " + sensorName + " for " + exp);
48         JPanel inpp = null;
49         env.addPanel(inpp);
50         Double val = null;
51         while (val == null) {
52             try {
53 //              String inps = inpp.read();
54 //              env.println("value of " + sensorName + " >" + inps);
55 //              val = Double.parseDouble(inps);
56 //              env.removePanel(inpp);
57             } catch (Exception e) {
58                 env.println("Sorry rt " + sensorName + " RETRY");
59             }
60         }
61         return val;
62     }
63
64     protected Double readFromIn() throws Exception {
65         try {
66             Double val = Double.parseDouble(new BufferedReader(
67                     new InputStreamReader(System.in)).readLine());
68             return val;
69         } catch (Exception e) {
70             return null;
71         }
72     }
73 }
```

**Listing 1.10.** `EvalSensor.java`

## 5 The M2M transformer

The class `ExpToExpm` can be written by an application designer in order to transform (via the method *mapToExpm*) a binary expression of type `exp.E` into an expression of type `expm.Expression`. We report here the version of this class written in `Xtend` by a system designer within the project *it.unibo.xtext.exp*:

```
1  /*
2   * by AN Unibo-DISI
3   */
4  package it.unibo.xtext.mygenerator
5  import it.unibo.xtext.exp.*
6  import it.unibo.xtext.expm.*;
7  import java.util.*
8
9
10 class ExpToExpm{
11
12     def Expression mapToExpm(E e){
13        var stack = new M2MStack()
14        genExp( e, stack )
15        stack.pop
16     }
17
18     def genExp(E e, M2MStack stack){
19        genTerm(e.term, stack)
20        genRe(e.re, stack)
21     }
22
23    def genRe(List<RE> re, M2MStack stack) {
24        if(  re.size > 0 ) {
25            val e = ExpmFactory::eINSTANCE.createComposedExp()
26            genTerm( re.get(0).term, stack )
27            e.setOp( createOp( re.get(0).op ) )
28            e.setRight( stack.pop )
29            e.setLeft( stack.pop )
30            stack.push(e)
31            genRe( re.toList.tail.toList, stack)
32        }
33    }
34
35
36    def genTerm(T term, M2MStack stack) {
37        genFactor(term.factor, stack)
38        genRt(term.rt,stack)
39    }
40
41    def genRt(List<RT> rt, M2MStack stack) {
42        if( rt.size > 0 ){
43            val e = ExpmFactory::eINSTANCE.createComposedExp()
44            genFactor( rt.get(0).factor, stack )
45            e.setOp( createOp( rt.get(0).op ) )
46            e.setRight( stack.pop )
47            e.setLeft( stack.pop )
48            stack.push(e)
49            genRt( rt.toList.tail.toList, stack)
50        }
51    }
52
53    def genFactor(F e, M2MStack stack) {
54        if( e.exp != null ) genExp(e.exp, stack)
55        else if( e.s != null ) genSensor(e.s, stack)
56            else genNum(e.num, stack)
57    }
58
59    def  genNum(it.unibo.xtext.exp.Num e, M2MStack stack) {
60        val n = ExpmFactory::eINSTANCE.createNum()
61        n.setVal(e.^val)
62        stack.push( n )
63    }
64
65    def Op createOp( String opStr ) {
66        if( opStr.contains('+')) createAddOp()
```

```
        else if( opStr.contains('-')) createMinusOp()
        else if( opStr.contains('*')) createMulOp()
        else if( opStr.contains('/')) createDivOp()
    }


    def AddOp createAddOp() {
        val op = ExpmFactory::eINSTANCE.createAddOp()
        op.setOp("+")
        op
    }


    def MinusOp createMinusOp() {
        val op = ExpmFactory::eINSTANCE.createMinusOp()
        op.setOp("-")
        op
    }

    def MulOp createMulOp() {
        val op = ExpmFactory::eINSTANCE.createMulOp()
        op.setOp("*")
        op
    }

    def DivOp createDivOp() {
        val op = ExpmFactory::eINSTANCE.createDivOp()
        op.setOp("/")
        op
    }

/*
 * Sensor
 */
    def  genSensor(S e, M2MStack stack) {
        val sensor = ExpmFactory::eINSTANCE.createSensor()
        sensor.setName( e.id )
        stack.push( sensor  )
    }


}
```

**Listing 1.11.** `ExpToExpm.xtend`

To perform its job, the `M2M` transformer uses a stack support written in `Java`:

```
package it.unibo.xtext.mygenerator;
import it.unibo.xtext.expm.*;
import java.util.Vector;
/*
 * Used by ExprToExpm.ext M2M rules
 */
public class M2MStack {
public static Expression curExp;
private Vector<Expression> stack = new Vector<Expression>();
    public M2MStack(){
        System.out.println("*** M2MStack created" );
    }
    public void push(Expression e ){
        System.out.println(" ***** M2MStack push " + e );
        stack.add(e);
    }
    public Expression pop( ){
        System.out.println(" ***** M2MStack pop size=" + stack.size() );
        if( isEmpty() ) return null;
        Expression e = stack.lastElement();
        curExp = e;
        stack.removeElementAt(stack.size()-1);
        return e;
    }
    public boolean isEmpty( ){
        return stack.size()==0;
    }
```

```
28      public static Expression getCurExp(){
29          return curExp;
30      }
31  }
```

**Listing 1.12.** `M2MStack.java`

The class `ExpToExpm.xtend` defines a generator that creates the `expm` version of an expression by starting from the `exp` meta-model created by the parser of the `exp` language. Thus, such a generator performs a model-to-model (`M2M`) mapping between the abstract syntax tree (`AST`) associated to our concrete syntax and an internal representation based on a binary tree.

In this way, we can obtain a binary tree from an expression written in the concrete, infix syntax `exp` and call our evaluation object(s) on the result of the transformation done by `ExpToExpm`. For example:

```
1   package it.unibo.xtext.exp.tests;
2   import static org.junit.Assert.*;
3   import it.unibo.xtext.exp.EL;
4   import it.unibo.xtext.expm.Expression;
5   import it.unibo.xtext.mygenerator.ExpToExpm;
6   import it.unibo.evaluator.EvalExpJava;
7   import it.unibo.evaluator.ModelUtil;
8
9   import org.junit.Test;
10
11  public class TestEvalWithM2M {
12  protected  ModelUtil modelUtil = new ModelUtil();
13  protected  EvalExpJava evalutil = new EvalExpJava();
14
15      @Test
16      public void testTransformAndEval() {
17          try {
18              String exps = "exp e0=s2-s3-s2+s3";
19              EL el = modelUtil.createExpFromParser(exps);
20              assertTrue("testTransformAndEval el",
21                      (el.eAllContents().next() instanceof it.unibo.xtext.exp.E));
22              Expression e = new ExpToExpm().mapToExpm(el.getExp());
23              //Expression e = modelUtil.createExpression(el);
24              assertTrue(
25                      "testTransformAndEval e",
26                      (e.eAllContents().next() instanceof it.unibo.xtext.expm.AddOp));
27              double res = evalutil.eval(e);
28              assertTrue("testTransformAndEval", res == 0.0);
29          } catch (Exception e) {
30              fail("testTransformAndEval " + e.getMessage());
31          }
32      }
33
34      @Test
35      public void testEval() {
36          try {
37              String exps = "exp e0=s2-s3-s2+s3";
38              double res = evalutil.eval(exps);
39              assertTrue("testEval", res == 0.0);
40          } catch (Exception e) {
41              fail("testEval " + e.getMessage());
42          }
43      }
44  }
```

**Listing 1.13.** `TestEvalWithM2M.java`

### 5.1 Evaluating expressions with sensors in Java

The test `testEval` calls an operation `eval` of an evaluation utility class written in `Java` that exploits all the classes so far defined in order to evaluate an expression espressed a as String in prefix form in the syntax of the `exp` custom language.

```java
package it.unibo.evaluator;
import it.unibo.xtext.exp.EL;
import it.unibo.xtext.expm.Expression;

public class EvalExpJava {

    protected EvalExpm evalExpm = new EvalExpm();
    protected ModelUtil modelUtil = new ModelUtil();

    public double eval(String s) throws Exception {
        return evalExpm.eval(s, modelUtil.createExpressionFromParser(s));
    }
    public double eval(EL expSentence) throws Exception {
        return evalExpm.eval(modelUtil.createExpression(expSentence));
    }
    public double eval(String s, EL expSentence) throws Exception {
        return evalExpm.eval(s, modelUtil.createExpression(expSentence));
    }
    public double eval(Expression e) {
        try {
            return evalExpm.eval(e);
        } catch (Exception e1) {
            e1.printStackTrace();
            return 0;
        }
    }
    public double eval(String s, Expression e) {
        try {
            return evalExpm.eval(s, e);
        } catch (Exception e1) {
            e1.printStackTrace();
            return 0;
        }
    }
}
```

**Listing 1.14.** `EvalExpJava.java`

## References


1. D. Harel and B. Rumpe. Meaningful modeling: What's the semantics of "semantics"? *Computer*, 64:64–72, 2004.
2. A. Natali. Introduction to xtext. http://edu222.deis.unibo.it/contact.
3. A. Natali and A. Molesini. *Costruire sistemi software: dai modelli al codice*. Esculapio, 2009.
4. Xtext. Xtext 2.1 documentation. http://www.eclipse.org/Xtext/.
5. Xtext. Xtext home page. http://www.eclipse.org/Xtext/.