LATVIJAS UNIVERSITĀTES RAKSTI. 2008, 733. sēj.:
DATORZINĀTNE UN INFORMĀCIJAS TEHNOLOĢIJAS 103.–139. lpp.

Model Transformation Languages L1, L2, L3 and Their Implementation

Edgars Rencis[1]
University of Latvia, IMCS, 29 Raiņa Blvd, Rīga, Latvia
Edgars.Rencis@lumii.lv

Abstract. In this paper a family of model transformation languages L1, L2, and L3 following the language L0 is introduced. The first language L0, not being part of this paper, is very simple and serves as a base language. It is implemented through an efficient compiler to C++ [1]. Each of the next languages L1, L2, and L3 is an extension of the previous one, and they are implemented by the bootstrapping method based on the language L0, that is, three compilers are written in L0: from L1 to L0, from L2 to L1, and from L3 to L2. The language L1 contains powerful pattern definition facilities, L2 – loops, and L3 – the branching facility. The language L3 is considered to be both sufficiently easy-to-use to serve as an intermediate language in the implementation of higher-level transformation languages, and expressive enough to be used in real model transformation tasks. The presented paper is an extended version of sections 4 – 6 of [10].

Keywords. Model transformation languages, L0, Lx, L1, L2, L3, compiler, bootstrapping.

1 Introduction

Although model transformation languages are the very heart of the MDA [2] – the most advanced architecture used to build systems nowadays – the implementation of various model transformation languages encountered in the world has not been very extensively researched. Actually, there exist only a few attempts to implement a model transformation language through some other language by using bootstrapping method [3-5]. The goal of this paper is to demonstrate the use of such an approach. It includes defining a sequence of model transformation languages and then implementing these languages by bootstrapping method one through another until the base transformation language is reached. In addition, another goal is to propose a language L3 that is, on the one hand, simple enough to be easily implementable, and, on the other hand, expressive enough to be used in practical model transformation tasks. Some the of results expounded on in this paper are also briefly outlined in [10].

[1] Partially supported by ESF (European Social Fund), project 2004/0001/VPD1/ESF/PIAA/04/NP/3.2.3.1/0001/0001/0063

The structure of this paper is the following:

1) the base transformation language L0 is described in Section 2 – it is very simple and contains only the basic transformation facilities; an efficient compiler to C++ is built for this language [1];
2) a sequence of model transformation languages L0', L1, L2, and L3 is introduced in Section 3; every next language of the so called Lx family is made based on the previous one by adding some new features; both the metamodel and the textual syntax is provided for each of languages; the language L3 is of a sufficiently high level to be used in practical model transformation tasks, however, it is still sufficiently easy-to-use to be used as an intermediate language in the implementation of higher-level model transformation languages (for example, the graphical transformation language MOLA [6,7,15]) by using the bootstrapping method;
3) the implementation of languages L0', L1, L2, and L3 is provided in Section 4; every next language is compiled to the previous one using the bootstrapping method.

2 Model transformation language L0

L0 is a textual model transformation language. It offers simple commands to work with arbitrary fixed instances of a given metamodel (for example, a command for creating a new instance, deleting an instance, getting and setting attribute's values, making and deleting links between instances, searching for instances etc.) and to handle simple control flows (it is done using the so called "goto" commands, as well as "else" branches that are attached to some L0 commands). To store persistent data, an in-memory repository has been developed at the University of Latvia, Institute of Mathematics and Computer Science [8].

An effective compiler from the language L0 to the language C++ has been developed. It means that it is possible to translate a program written in L0 into a C++ code, which can further be compiled to a ".dll" file. When it is done, the resulting ".dll" file can be executed on a metamodel given by the user.

A more detailed description of the language L0 is available in [1], however, an overview of this language (commands + metamodel) is given in the next sections of this paper in order to make this paper understandable without the necessity to read the abovementioned paper.

2.1 Command of the Transformation Language L0

Base model transformation language L0 is a fully procedural language and contains the following commands (that can be found in the body of any procedure or function) [9]:

1) **call** <subProgName> **(**<actualParamList>**)** – calls the subprogram with the given parameters;
2) **return** – returns the control to the calling program;
3) **return** <identifier> – returns the value of <identifier> to the calling program;

4) **first** <pointerName> **:** <className> [**else** <labelName>] – positions the pointer <pointerName> to an arbitrary instance of the class <className>. If there are no instances of the given class, the control is given to the label <labelName> ;
5) **first** <pointerName1> **:** <className> **from** <pointerName2> **by** <roleName> [**else** <labelName>] – positions the pointer <pointerName1> to such an arbitrary instance of the class <className> that is reachable from the pointer <pointerName2> by the role <roleName>. If there are no such instances, the control is given to the label <labelName>. After the command has been executed, the value set of the pointer <pointerName1> is limited to exactly those instances of the class <className>, which are reachable from the pointer <pointerName2> by the role <roleName> ;
6) **next** <pointerName> [**else** <labelName>] – positions the pointer <pointerName> to the next instance that satisfies conditions raised by the respective "first" command (the previous one with the same pointer <pointerName>) and that is not yet visited by commands "first" or "next". If there are no such instances, the control is given to the label <labelName> ;
7) **goto** <labelName> – gives the control the label <labelName> ;
8) **label** <labelName> – defines the label <labelName> ;
9) **addObj** <pointerName> **:** <className> – creates a new instance of the class <className> ;
10) **addLink** <pointerName1> **.** <roleName> **.** <pointerName2> – creates a link between instances <pointerName1> and <pointerName2> with the role name <roleName> at the end of the instance <pointerName2> ;
11) **deleteObj** <pointerName> – deletes the instance <pointerName> ;
12) **deleteLink** <pointerName1> **.** <roleName> **.** <pointerName2> – deletes the link between instances <pointerName1> and <pointerName2> with the role name <roleName> at the end of the instance <pointerName2> ;
13) **setPointer** <pointerName1> = <pointerName2> – positions the pointer <pointerName1> to the instance pointed to by the pointer <pointerName2> ;
14) **setPointerF** <pointerName> = <funcName> **(**<actualParamList>**)** – positions the pointer <pointerName> to the instance returned by the function <funcName> called with the given parameters ;
15) **setVar** <varName> = <binExpr> – sets the value of the variable <varName> to the value of the binary expression <binExpr> ;
16) **setVarF** <varName> = <funcName> **(**<actualParamList>**)** – sets the value of the variable <varName> to the value returned by the function <funcName> called by given parameters ;
17) **setAttr** <pointerName> **.** <attrName> = <binExpr> – sets the value of the attribute <attrName> of the instance <pointerName> to the value of the binary expression <binExpr> ;
18) **type** <pointerName> == <className> [**else** <labelName>] – if the pointer <pointerName> points to the instance of the class <className>, the control is given to the next command, otherwise the control is given to the label <labelName>. Inequality ("!=") is allowable instead of the equality as well;

19) **var** <varName> == <binExpr> [**else** <labelName>] – if the value of the variable <varName> is equal to the value of the binary expression <binExpr>, the control is given to the next command, otherwise the control is given to the label <labelName>. Any other comparison operators ("<", "<=", ">", ">=",or "!=") are allowable instead of the equality as well;
20) **pointer** <pointerName1> == <pointerName2> [**else** <labelName>] – if pointers <pointerName1> and <pointerName2> point to the same instance, the control is given to the next command, otherwise the control is given to the label <labelName>. Inequality ("!=") is allowable instead of the equality as well;
21) **attr** <pointerName> **.** <attrName> == <binExpr> [**else** <labelName>] – if the value of the attribute <attrName> of the instance <pointerName> is equal to the value of the binary expression <binExpr>, the control is given to the next command, otherwise the control is given to the label <labelName>. Any other comparison operators ("<", "<=", ">", ">=" or "!=") are allowable instead of the equality as well;
22) **link** <pointerName1> **.** <roleName> **.** <pointerName2> [**else** <labelName>] – if there exists a link with the role name <roleName> at the end of the instance <pointerName2> between instances <pointerName1> and <pointerName2>, the control is given to the next command, otherwise the control is given to the label <labelName> ;
23) **nolink** <pointerName1> **.** <roleName> **.** <pointerName2> [**else** <labelName>] – if there does not exist a link with the role name <roleName> at the end of the instance <pointerName2> between instances <pointerName1> and <pointerName2>, the control is given to the next command, otherwise the control is given to the label <labelName> ;
24) **DEBUG_ON –** turns on the debugging mode;
25) **DEBUG_OFF** – turns off the debugging mode.

Since the transformation language L0 is a strongly typified language, it is required that any variable is declared in a separate block in each procedure or function in the following manner:

1) **var** <varName> **:** <typeName> – declares a variable with a primitive data type (*Integer*, *Real*, *String* or *Boolean*)
2) **pointer** <pointerName> **:** <className> – declares a pointer to instances of the class <className>

There actually exists an extension of the language L0 – language L0+. In the language L0+, commands working in metamodel level are added. Namely, it is possible, for example, to make and delete classes, associations and attributes in L0+. So it is possible to make a specific metamodel in L0+ and then to execute the program written in L0 (or L0+) on this metamodel. As it is not the goal of this paper, commands of the language L0+ have not been discussed here.

2.2 The Metamodel of the Language L0

Since metamodels of the further introduced transformation languages will be based on the metamodel of the language L0, it is necessary to discuss this metamodel in detail (see Fig. 1).

The metamodel of the language L0 is quite intuitive – every transformation program (an instance of the class "Transformation") contains procedures/functions that in their turn contain command blocks starting with one command while every command does not have more than one next command. Every procedure/function has its variable definition block as well.

In the language L0, four types of commands exist:

1) instances of the class "GotoCom" – control flow commands;
2) instances of the class "FNCom" – instance searching commands ("first" and "next");
3) instances of the class "ECom" – commands with a possible "else" branch ("type", "var", "pointer", "attr", "link" and "noLink");
4) instances of the class „SCom" – other commands ("call", "return", "label", "addObj", "addLink", "deleteObj", "deleteLink", "setPointer", "setPointerF", "setVar", "setVarF", "setAttr", "DEBUG_ON" and "DEBUG_OFF").

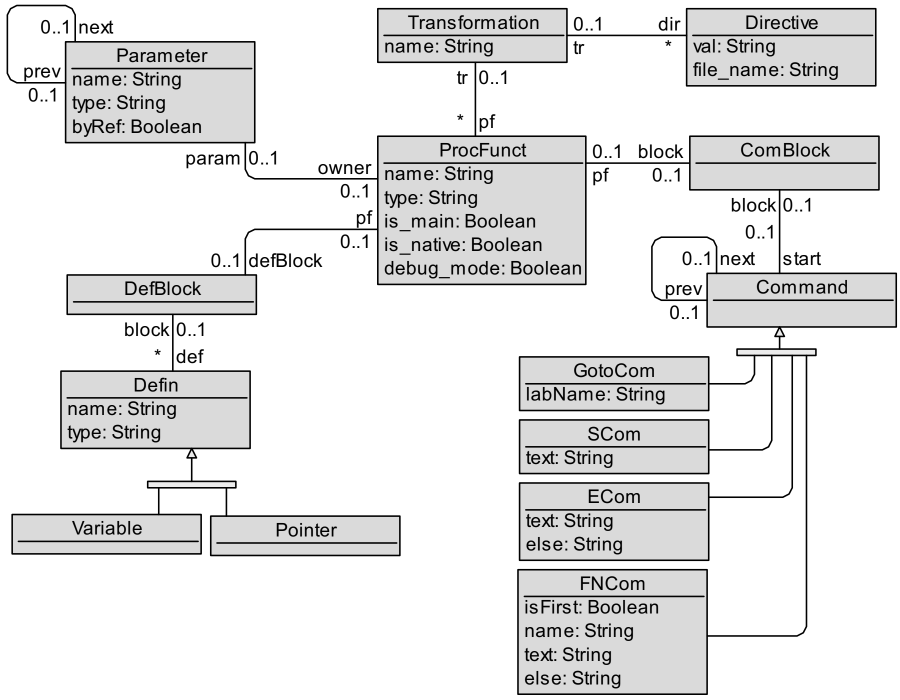

Fig 1. The metamodel of the language L0

2.3 Model Transformation Example in the Language L0

Let's assume we have given a metamodel consisting of two classes (Fig. 2). Students have name, age, and average marks in each of the eight bachelor's study examination periods. Instances of the class "Course" are courses of master studies and the attribute "hasGoodStudents" shows whether the average mark of all bachelor's examination periods for all adult students of the particular course is at least 8. The attribute "title" of the class "Course" is supposed to be unique. It must be mentioned that the given metamodel is not the best solution for such a fragment of the world, but it is in return very appropriate for the demonstration of the use of languages Lx.

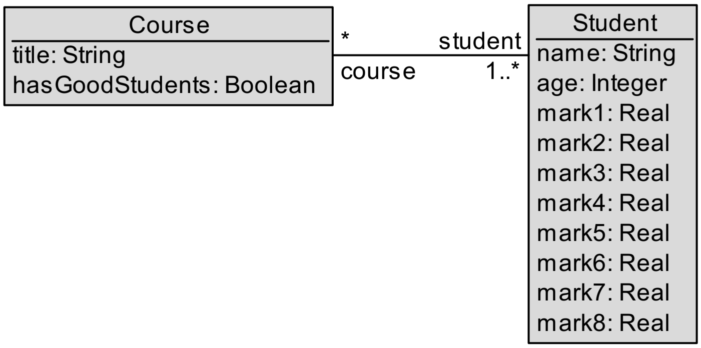

Fig. 2. Metamodel used in the example

The problem to solve is as follows – set the correct value of the attribute "hasGoodStudents" for the course named "Operating Systems". The solution written in the language L0 is given below.

```
transformation example;
 main procedure main();
  pointer c:Course;
  pointer s:Student;
  var x:Real;
  var avg:Real;
  var count:Integer;
 begin;
  first c:Course else endOfProg;
  label startFinding;
  attr c.title=="Operating Systems" else getNextCourse;
  goto courseFound;
  label getNextCourse;
  next c else endOfProg;
  goto startFinding;
  label courseFound;
  setVar x=0;
```

```
    setVar count=0;
    first s:Student from c by student
      else noMoreStudents;
    label startCounting;
    attr s.age>=18 else getNextStudent;
    setVar count=count+1;
    setVar avg=s.mark1;
    setVar avg=avg+s.mark2;
    setVar avg=avg+s.mark3;
    setVar avg=avg+s.mark4;
    setVar avg=avg+s.mark5;
    setVar avg=avg+s.mark6;
    setVar avg=avg+s.mark7;
    setVar avg=avg+s.mark8;
    setVar avg=avg/8;
    setVar x=x+avg;
    label getNextStudent;
    next s else noMoreStudents;
    goto startCounting;
    label noMoreStudents;
    var count>0 else writeGood;
    setVar x=x/count;
    var x<8 else writeGood;
    setAttr c.hasGoodStudents=false;
    goto endOfProg;
    label writeGood;
    setAttr c.hasGoodStudents=true;
    label endOfProg;
   end;
  endTransformation;
```

3 Model transformation languages L0’ until L3

Transformation languages Lx (or, the so called Lx language family) contain the transformation language L0 and its related transformation languages L0’, L1, L2, and L3. Each of these languages is built based on the previous language of this family by adding some extra features. The syntax and semantics of languages L0’, L1, L2, and L3 are described in this section.

3.1 Transformation Language L0’

Model transformation language L0’ (read – „*L0 prim*”) is based on the language L0. The new feature of L0’ is the possibility to make long arithmetic expressions (in L0, only unary and binary expressions were allowed).

Arithmetic expressions of an arbitrary length are allowed in L0'. It means that it is allowed to use each of the four arithmetic operators and traditional brackets ("(" and ")") when building long expressions. Variables, constants, attributes, and functions can be used as operands in such expressions. The use of operators with respect to the data types is shown in Table 1.

Table 1. The use of arithmetic operators with respect to data types

Operator	Left hand operand	Right hand operand	Result
+	*Integer*	*Integer*	*Integer*
	Integer	*Real*	*Real*
	Real	*Integer*	*Real*
	Real	*Real*	*Real*
	String	*String*	*String*
-	*Integer*	*Integer*	*Integer*
	Integer	*Real*	*Real*
	Real	*Integer*	*Real*
	Real	*Real*	*Real*
*	*Integer*	*Integer*	*Integer*
	Integer	*Real*	*Real*
	Real	*Integer*	*Real*
	Real	*Real*	*Real*
/	*Integer*	*Integer*	*Real*
	Integer	*Real*	*Real*
	Real	*Integer*	*Real*
	Real	*Real*	*Real*

The traditional operator execution sequence is taken into account (from the highest to the lowest):

1) function calls;
2) brackets;
3) multiplication and division;
4) addition and subtraction.

The metamodel of L0' is made by taking the metamodel of L0 and supplementing it with some new classes and associations. In this metamodel, the class "Expression" together with some other classes is added. Every expression can be attached either to some instance of the class "Ecom" (if it is a comparison) or to some instance of the class "Scom" (if it is an assignment). Every expression contains one starting primitive (instance of the class "Eelem"), and every expression's primitive has at most one next primitive. Primitives can be of various types – variables, attributes, function calls, constants, operators, and brackets (Fig. 3, bold classes and associations are new in L0').

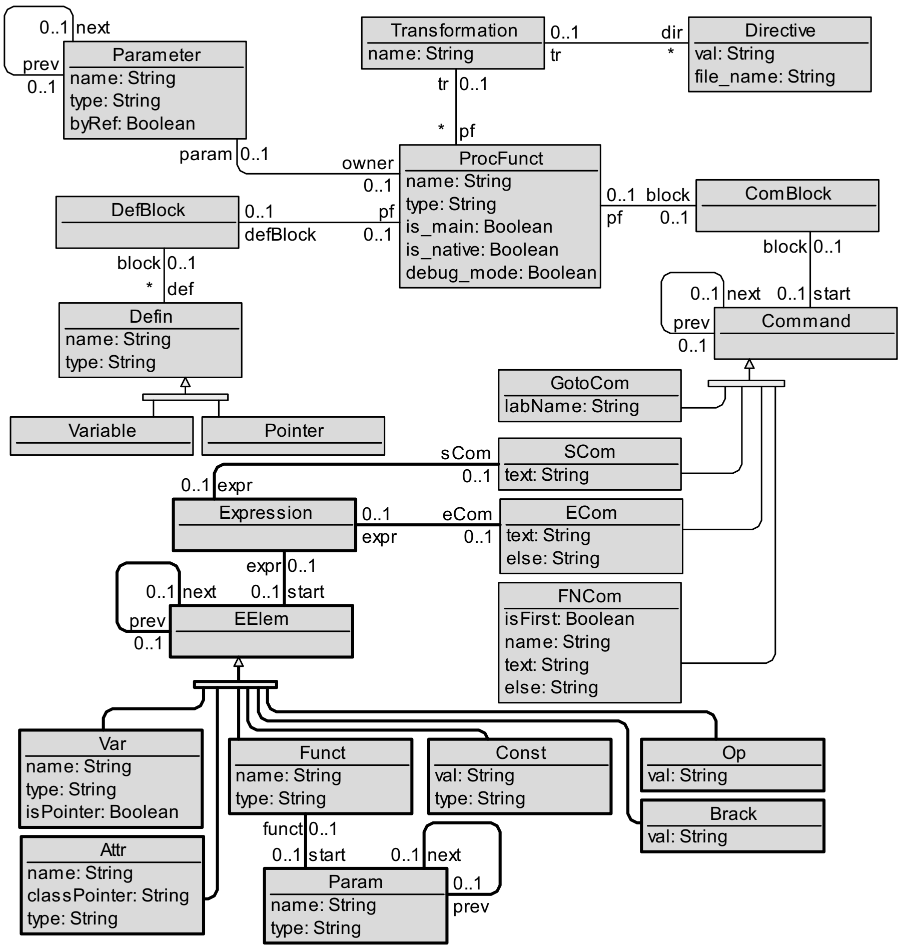

Fig. 3. The metamodel of the transformation language L0'

Commands of L0 are the same in L0'. The only difference is in those places where some binary expression could be in the language L0 – now an expression of an arbitrary length is allowed in the language L0'. So it needs to be specified how to write so long expressions. An arithmetic expression can be defined as one of the following:

1) a constant of the type *String* (for example, “17”);
2) a positive constant of the type *Integer* (for example, 17) or *Real* (for example, 17.0);
3) (-C), where C – a positive constant of the type *Integer* or *Real*;
4) a variable of the type *Integer*, *Real* or *String*;
5) an attribute of the type *Integer*, *Real* or *String* that is written in the following way – <pointerName> **.** <attributeName> **:** <typeName>, where <pointerName> is declared as a pointer to the class whose attribute is to inspect;

6) a function call, where the function is of the type *Integer*, *Real* or *String*;
7) (E), where E – an arithmetic expression;
8) E+F, where E and F – arithmetic expressions with compatible types;
9) E-F, where E and F – arithmetic expressions with compatible types;
10) E*F, where E and F – arithmetic expressions with compatible types;
11) E/F, where E and F – arithmetic expressions with compatible types.

For example, correct commands in the language L0' are as follows (if based on the metamodel shown in Fig. 4):

```
1) setVar x=x+y+2;
2) setVar s=z+":"+z+"..."+s1;
3) var x==i*(y+(17/2));
4) attr p.age!=i+person1.age:Integer-1;
5) var y==17.5+3*5/(x+y);
```

It is assumed in those commands that variables and pointer are defined like this:

```
var x:Real;
var y:Real;
var i:Integer;
var s:String;
var s1:String;
var z:String;
pointer p:Person;
pointer person1:Person;
```

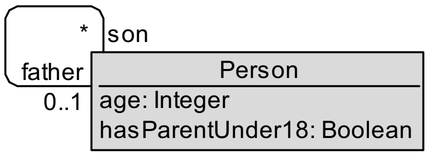

Fig.4. The metamodel used in L0' examples

The transformation that solves the problem proposed in Section "2.3. Model transformation example in the language L0" can resemble this in the language L0':

```
transformation example;
 main procedure main();
  pointer c:Course;
  pointer s:Student;
  var x:Real;
  var count:Integer;
 begin;
  first c:Course else endOfProg;
  label startFinding;
  attr c.title=="Operating Systems" else getNextCourse;
  goto courseFound;
  label getNextCourse;
  next c else endOfProg;
  goto startFinding;
```

```
    label courseFound;
    setVar x=0;
    setVar count=0;
    first s:Student from c by student
      else noMoreStudents;
    label startCounting;
    attr s.age>=18 else getNextStudent;
    setVar count=count+1;
    setVar x = x + ( s.mark1:Real + s.mark2:Real +
      s.mark3:Real + s.mark4:Real + s.mark5:Real +
      s.mark6:Real + s.mark7:Real + s.mark8:Real ) / 8;
    label getNextStudent;
    next s else noMoreStudents;
    goto startCounting;
    label noMoreStudents;
    var count>0 else writeGood;
    setVar x=x/count;
    var x<8 else writeGood;
    setAttr c.hasGoodStudents=false;
    goto endOfProg;
    label writeGood;
    setAttr c.hasGoodStudents=true;
    label endOfProg;
  end;
endTransformation;
```

3.2 Transformation Language L1

Transformation language L1 (if compared to L0’) is supplemented with a pattern matching facility, so that it is possible to search for some instances satisfying a given pattern. Any L1 pattern can contain conditions put on values of variables or attributes, links between instances and other. Although pattern matching can be considered to be one of the most fundamental modeling concept, the only thing that differs L1 metamodel from the metamodel of the language L0’ is one association between classes “FNCom” and “ComBlock” (Fig. 5, the new association is drawn in bold).

So it is now possible to attach the so called “suchthat” block to every instance searching command (these are instances of the class „FNCom”). This block can contain arbitrary L1 commands and thus the pattern can be specified.

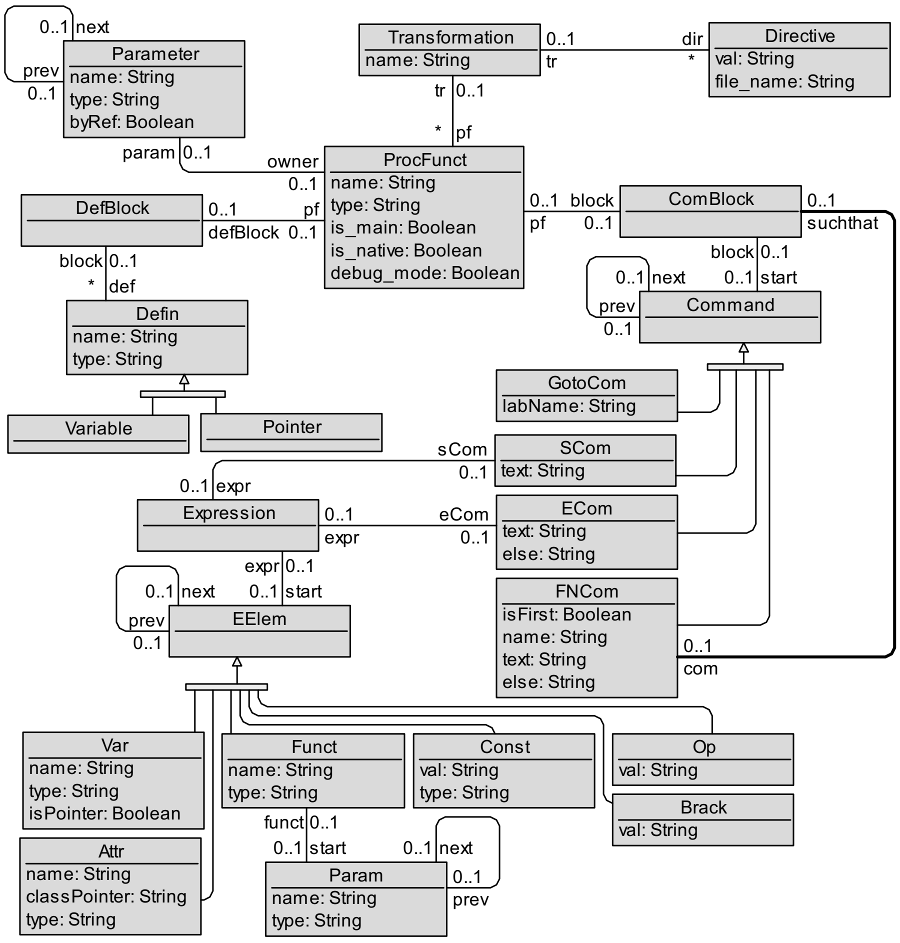

Fig. 5. The metamodel of the transformation language L1

In textual syntax, the only difference between languages L0' and L1 is in commands "first" and "next". Now it is possible to attach a pattern to them:

```
first <pointerName1> : <className> [ from
 <pointerName2> by <roleName> ] [ suchthat
begin
 <L1Commands>
end ];
next <pointerName> [ suchthat
begin
 <L1Commands>
end ];
```

What is the semantics of the "suchthat" block at all? Commands of this block can always give an answer to the question – does the particular instance satisfy the given pattern or not? Therefore the pattern matching block can be treated like a novel expression of the logical type (*Boolean*) that will further be called the *begin-end* expression [10]. In more formal terms – a *begin-end* expression is any construction built like this:

```
begin
 <L1Commands>
end
```

Now it is possible to define the semantics of a "suchthat" block (or a *begin-end* expression) – a *begin-end* expression is true if, taking into account particular instance and executing all the commands of the given block one by another (starting from the first one), it is possible to successfully reach the end of the block (meaning – successfully execute its last command).

What does it mean in L1 to successfully execute a command? In order to answer this question it will be enough to inspect commands of just two types – "goto" command and commands with a possible "else" branch ("ECom" and "FNCom" instances in the metamodel). In the case of any other L1 command it is assumed that these commands are always successfully executable. Let's take a more detailed view of the two types of commands mentioned above:

1) "goto" commands in the language L0 must be supplemented with exactly one label name (to which label the control must be given after the execution of this "goto" command). In L1, "goto" commands – if used in *begin-end* expressions – must be supplemented with no more than one label. It means the label attached to this command can be empty. If that is the case, the value of the particular *begin-end* expression becomes equal to false when reaching such a "goto" command, and no more commands of this block are to be executed. So the "goto" command is successfully executable if there is exactly one label name attached to it.
2) "ECom" and "FNCom" commands in L0 can contain no more than one "else" branch. If some command contains no "else" branch and it is the case when some comparison of instance searching fails, the control is given to the end of this particular procedure/function. In L1, a non-existing "else" branch in the situation the control would have given to the label specified in this "else" branch leads to the false value of the particular *begin-end* expression that contains this command. So a command that is an instance of the class "ECom" or an instance of the class "FNCom" is successfully executable if it contains either an "else" branch or the comparison, or instance searching does not fail.

Since the semantics of the instance searching commands ("first" and "next") might not be intuitively precisely clear, it is necessary to explain it in detail. In L0, the semantics of these commands are explained in the following manner:

1) When reaching the "first" command with a pointer <pointerName> to the class <className> attached, a possible value set is assigned for this pointer, that is – those instances of the class <className> are distinguished to which it will be further possible for this particular pointer to point. If there is no "from ... by ..." part in this command, the value set contains all the instances of the class <className>, otherwise the value set of <pointerName> is limited to exactly those instances of the class <className> that are reachable from the instance by the role specified in the "from ... by ..." part. After the value set is determined, an arbitrary instance from this set is assigned to <pointerName>

and then withdrawn from the set. If the set is found to be empty, the control is given to the label specified in the "else" branch (if it exists).

2) When reaching the "next" command with a pointer <pointerName> attached (the same pointer that has been attached to some "first" command before), an arbitrary instance of the previously made value set of this pointer is assigned to <pointerName> and then withdrawn from the set. If the set is found empty, the control is given to the label specified in the "else" branch (if it exists).
3) When reaching the "next" command with a pointer attached that is not yet processed in any "first" command (so the value set is not determined for it), program execution semantics is not defined.

In L1, the semantics of instance searching commands is adopted from the language L0, and some conditions according to the semantics of the pattern matching block are added:

1) When reaching the "first" command with a pointer <pointerName> attached, its value set is determined in the same way it was done in the case of the language L0. After that, an arbitrary instance of this value set that satisfies the given *begin-end* expression (if it exists) is assigned to <pointerName> and then withdrawn from the set. If there are no such instances, the control is given to the label specified in the "else" branch (if it exists).
2) When reaching the "next" command with a pointer <pointerName> attached that has previously determined value set (the "first" command on this pointer is executed before), an arbitrary instance of this value set that satisfies the given *begin-end* expression (if it exists) is assigned to <pointerName> and then withdrawn from the set. If there are no such instances, the control is given to the label specified in the "else" branch (if it exists).
3) When reaching the "next" command with such a pointer attached that is not yet processed in any "first" command (so the value set is not determined for it), program execution semantics is not defined.

Let's consider some examples now. A simple pattern based on which the first instance of the class "Person" is found, where the condition holds that the age of the particular person is 24 (examples used in this section are based on the metamodel shown in Fig. 4):

```
first p:Person suchthat
begin
 p.age==24;
end;
```

In this case, first such p from the class "Person" will be found whom it will be possible to successfully execute this only command – "p.age==24;". Since it is a command of type "ECom" and it does not contain an "else" branch, the only possible way for this command to be able to execute successfully is the way when the comparison holds. So the *begin-end* expression is true in this case if the value of the attribute "age" of the instance pointed to by p is equal to 24.

To find the next instance of the same class based on the same condition, the "next" command with a pattern matching block needs to be executed:

```
next p suchthat
begin
 attr p.age==24;
```

```
end
else no_more_persons;
```

A pattern based on which the first instance of the class “Person” is found whom a condition holds that it is 24 years old and it is the son of another person pointed to by the pointer *father*:

```
first p:Person suchthat
begin
    attr p.age==24;
 link father.son.p;
end
else no_such_persons;
```

A problem might arise – find the persons that have a 24 year-old son. In this case, the command in L1 that finds the first such person can look like this:

```
first parent:Person suchthat
begin
 first p:Person suchthat
 begin
  link parent.son.p;
  attr p.age==24;
 end;
end
else no_such_persons;
```

If this command executes and the control is not given to the “else” label, the pointer *parent* will point to such instance of the class “Person” that satisfies the condition specified above (moreover – the pointer *p* will point to the instance of the class “Person” that has the link with the given name to the instance pointer to by *parent*). The inner “first” command can be read as “exists”, that is, all the pattern can be read as “Find the first *parent* whom there exists such *p* that is in a relation *son* with the pointer *parent* and that is 24 years old”.

The transformation that solves the problem proposed in Section “2.3. Model transformation example in the language L0” can resemble this in the language L1:

```
transformation example;
 main procedure main();
  pointer c:Course;
  pointer s:Student;
  var x:Real;
  var count:Integer;
 begin;
  first c:Course suchthat
  begin
   attr c.title=="Operating Systems";
  end else endOfProg;
  setVar x=0;
  setVar count=0;
  first s:Student from c by student suchthat
  begin
   attr s.age>=18;
  end else noMoreStudents;
```

```
    label startCounting;
    setVar count=count+1;
    setVar x = x + ( s.mark1:Real + s.mark2:Real +
       s.mark3:Real + s.mark4:Real + s.mark5:Real +
       s.mark6:Real + s.mark7:Real + s.mark8:Real ) / 8;
    next s suchthat
    begin
     attr s.age>=18;
    end else noMoreStudents;
    goto startCounting;
    label noMoreStudents;
    var count>0 else writeGood;
    setVar x=x/count;
    var x<8 else writeGood;
    setAttr c.hasGoodStudents=false;
    goto endOfProg;
    label writeGood;
    setAttr c.hasGoodStudents=true;
    label endOfProg;
   end;
  endTransformation;
```

3.3 The Comparison of L1 and a First-Order Logic

How expressive exactly are the pattern definition blocks of the transformation language L1? What are the types of problems solvable by these constructions? This section is devoted to these questions.

Pattern definition block (or to be more precise – the *begin-end* expression attached to it) gives exactly one answer of the logical data type (true or false) for each object of the set under consideration. If looking at the pattern block in such a way, one can start to draw an analogy with formulae of first-order logic that are objects of the logical type as well. While transformation language L1 is known only by a small set of people, first-order logic is considered to be a classic and is ranked as one of the basic disciplines of mathematics. Therefore the comparison of L1 and a first-order logic would give us a better notion of the scope of L1.

Let's consider a many-sorted first-order logic [11]. According to the definition, the alphabet of such a language consists of seven sets of symbols:

1) a countable set $S \cup \{bool\}$ of sorts (or types) containing the special sort *bool* such that S is non-empty and does not contain *bool*;
2) logical connectives: $\wedge$ (conjunction), $\vee$ (disjunction), $\supset$ (implication) and $\equiv$ (equivalence) that are all of rank (*bool* $\times$ *bool* $\rightarrow$ *bool*), $\neg$ (negation) of rank (*bool* $\rightarrow$ *bool*) and $\perp$ (a bottom concept) of rank ($\varnothing \rightarrow$ *bool*);
3) quantifiers: $\forall_s$ (universal quantifier) and $\exists_s$ (existential quantifier) for every set $s \in S$;
4) an equality symbol: $=_s$ of rank ($s \times s \rightarrow$ *bool*) for every set $s \in S$;

5) variables: a countably infinite set $V_s = \{x_0, x_1, x_2, ...\}$ for every set $s \in S$ each variable x_i being of rank $(\varnothing \rightarrow s)$;
6) auxiliary symbols: "(" and ")";
7) an alphabet L of non-logical symbols consisting of:
   a. function symbols: a countable set $FS = \{f_0, f_1, ...\}$ and a rank function $r: FS \rightarrow S^+ \times S$ (S^+ contains all the words of S excepting the empty word, that is, all the strings of length $n>0$ whose all elements belong to the set S), assigning a pair $r(f) = (u,s)$ called rank to every function symbol f; the string u is called the arity of f, and the symbol $s \in S$ – the sort (or type) of f;
   b. constants: a countable set $CS_s = \{c_0, c_1, ...\}$ for every set $s \in S$ each c_i being of rank $(\varnothing \rightarrow s)$;
   c. predicate symbols: a countable set $PS = \{P_0, P_1, ...\}$and a rank function $r: PS \rightarrow S^* \times \{bool\}$ (S^* contains all the words of S including the empty word) assigning a pair $r(P) = (u, bool)$ to each predicate symbol P; the string u is called the arity of P.

It is assumed that all the sets V_s, FS, CS_s and PS is mutually disjoint for every possible value of $s \in S$.

Taking into account such a definition, terms and atomic formulae in the first-order logic are defined as follows:

1) every constant and every variable of sort s is a term of sort s;
2) if $t_1, ..., t_n$ are terms, each t_i of sort u_i, and f is a function symbol of rank $(<u_1, ..., u_n> \rightarrow s)$, then $f(t_1, ..., t_n)$ is a term of sort s;
3) every predicate symbol of arity $\varnothing$, as well as the bottom concept ($\bot$) is an atomic formula;
4) if t_1 and t_2 are terms of sort s, then $=_s(t_1, t_2)$ is an atomic formula;
5) if $t_1, ..., t_n$ are terms, each t_i of sort u_i, and P is a predicate symbol of arity $u_1, ..., u_n$, then $P(t_1, ..., t_n)$ is an atomic formula.

Formulae are defined as follows:

1) every atomic formula is a formula;
2) for any two formulae A and B, $(A \wedge B)$, $(A \vee B)$, $(A \supset B)$, $(A \equiv B)$ and $\neg A$ are also formulae;
3) for any variable x of sort s and any formula A, $\forall_s x(A)$ and $\exists_s x(A)$ are also formulae.

Let's look now at a subset of full many-sorted first-order logic called the language P^-, that contains only binary predicate symbols and functions with only one argument. In that case, the alphabet of the language P^- can be defined in the following manner:

1) a countable set $S \cup \{bool\}$ of sorts (or types) containing the special sort *bool* such that S is non-empty and does not contain *bool*;
2) logical connectives: $\wedge$ (conjunction) and $\vee$ (disjunction) of rank $(bool \times bool \rightarrow bool)$, $\neg$ (negation) of rank $(bool \rightarrow bool)$ and $\bot$ (a bottom concept) of rank $(\varnothing \rightarrow bool)$;
3) quantifiers: $\forall_s$ (universal quantifier) and $\exists_s$ (existential quantifier) for every set $s \in S$;
4) an equality symbol: $=_s$ of rank $(s \times s \rightarrow bool)$ for every set $s \in S$;
5) variables: a countably infinite set $V_s = \{x_0, x_1, x_2, ...\}$ for every set $s \in S$ each variable x_i being of rank $(\varnothing \rightarrow s)$;

6) auxiliary symbols: "(" and ")";
7) an alphabet L of non-logical symbols consisting of:
   a. function symbols: a countable set FS = $\{f_0, f_1, ...\}$ and a rank function r: FS $\rightarrow$ S $\times$ S, assigning a pair r(f) = (s,s) to every function symbol f;
   b. constants: a countable set $CS_s = \{c_0, c_1, ...\}$ for every set $s \in S$ each c_i being of rank ($\varnothing \rightarrow$ s);
   c. predicate symbols: a countable set PS = $\{P_0, P_1, ...\}$and a rank function r: PS $\rightarrow S^2 \times \{bool\}$, assigning a pair r(P) = (<s,s>, *bool*) to each predicate symbol P.

So terms and atomic formulae in P^- can be defined as follows:
1) every constant and every variable of sort s is a term of sort s;
2) if t is a term of sort u, and f is a function symbol of rank (u $\rightarrow$ s), then f(t) is a term of sort s;
3) $\perp$ is an atomic formula;
4) if t_1 and t_2 are terms of sort s, then $=_s(t_1, t_2)$ is an atomic formula;
5) if t_1 and t_2 are terms, each t_i of sort u_i, and P is a predicate symbol of arity $<u_1, u_2>$, then $P(t_1, t_2)$ is an atomic formula.

Formulae in P^- are defined as follows:
1) every atomic formula is a formula;
2) for any two formulae A and B, $(A \wedge B)$, $(A \vee B)$ and $\neg A$ are also formulae;
3) for any variable x of sort s and any formula A, $\forall_s x(A)$ and $\exists_s x(A)$ are also formulae.

Now it is possible to see some similarities between languages P^- and L1. Although different terms are used to define these two languages, it is possible to establish some links between them (see Table 2).

Table 2. Linking concepts of languages P^- and L1

Concept of P^-	**Concept of L1**
The set of sorts S	The set C $\cup$ {*Integer*, *Real*, *String*, *Boolean*} where C – the set of all classes found in the metamodel used
The bottom concept $\perp$	*Boolean* value *false*
Other logical connectives	Will be interpreted in the context
Existential quantifier $\exists_s$ where $s \in C$, and C – the set of all classes found in the metamodel used	The command "first"
Universal quantifier $\forall_s$ where $s \in C$, and C – the set of all classes found in the metamodel used	Will be interpreted by transforming the expression containing the universal quantifier into the form of that containing an existential quantifier
The equality symbol $=_s$ where $s \in$ {*Integer*, *Real*, *String*, *Boolean*}	The command "var"
The equality symbol $=_s$ where $s \in C$, and C – the set of all classes found in the metamodel used	The command "pointer"

The set of variables V_s where $s \in \{Integer, Real, String, Boolean\}$	Variables of primitive data types, declared by the keyword "var"
The set of variables V_s where $s \in C$, and C – the set of all classes found in the metamodel used	Pointers to instances, declared by the keyword "pointer"
Auxiliary symbols "(" and ")"	Will be interpreted in the context
The function symbol with one argument	The attribute of a class
Constants of types	Constants of primitive data types
Binary predicate symbols $P(t_1, t_2)$ where $t_1, t_2 \in C$, and C – the set of all classes found in the metamodel used	The command "link"

Theorem. For each formula of the predicate language P^-, there exists a *begin-end* expression in the language L1 of the same truth value.

Proof. A constructive proof is provided for this theorem. For the theorem to be proven it is sufficient to produce a valid *begin-end* expression for each type of formulae of P^- shown in Table 2. To do this, two auxiliary formulae need to be introduced:

1) expr: <P^- formula> → <L1 *begin-end* expression> – a function assigning an L1 *begin-end* expression to the given P^- formula;
2) insert: <L1 *begin-end* expression> × <*String*> → <L1 *begin-end* expression> – a function calculating a new *begin-end* expression from the existing one by adding the given label name (second parameter) to missing places of the initial expression (to "goto" commands without a label and to non-existing "else" branches of those commands that can contain an "else" branch).

All types of P^- formulae and their respective L1 *begin-end* expressions are shown in Table 3. (labels "unicalLabel", "unicalLabelForA", and "endLabel", as well as pointers "unicalPtrName1" and "unicalPtrName2", and variables "unicalVarName1" and "unicalVarName2" are considered to be unique in the whole given procedure/function).

Table 3. Construction of an L1 code from P^- formulae

F	expr(F)
$\perp$	**goto**;
$=_s(t_1,t_2)$ where $s \in \{Integer, Real, String, Boolean\}$	**setVar** unicalVarName1=t_1; **setVar** unicalVarName2=t_2; **var** unicalVarName1==unicalVarName2;
$=_s(t_1,t_2)$ where $s \in C$, and C – the set of all classes found in the metamodel used	**setPointer** unicalPtrName1=t_1; **setPointer** unicalPtrName2=t_2; **pointer** unicalPtrName2==unicalPtrName2;
$P(t_1, t_2)$	**setPointer** unicalPtrName1=t_1; **setPointer** unicalPtrName2=t_2; **link** unicalPtrName1.P.unicalPtrName2;

$A \wedge B$	expr(A)
	expr(B)

$A \vee B$	**insert**(expr(A),"unicalLabel")
	goto endLabel;
	label unicalLabel;
	expr(B)
	label endLabel;

$\neg A$	**insert**(expr(A),"unicalLabel") **goto**; **label** unicalLabel;

$\exists_s x(A)$	**first** x:S **suchthat**
	begin
	  expr(A)
	end;

$\forall_s x(A) \equiv \neg\exists_s x(\neg A)$	**first** x:S **suchthat**
	begin
	  insert(expr(A),"unicalLabelForA")
	  goto;
	  label unicalLabelForA;
	end else unicalLabel;
	goto;
	label unicalLabel;

It is worth mentioning that it is easier to use the form of an existential quantifier and to produce a *begin-end* expression based on that in the case of a universal quantifier.

In order to get a clearer understanding of the functions used to construct the L1 code, examples of all the different cases are given in Table 4.

Table 4. Construction of an L1 code from P^- formulae – examples

F	**expr(F)**
$\perp$	**goto**;

$=_{Integer}(x,17)$	**setVar** unicalVarName1=x;
	setVar unicalVarName2=17;
	var unicalVarName1==unicalVarName2;

$=_{Person}(p,q)$	**setPointer** unicalPtrName1=p;
	setPointer unicalPtrName2=q;
	pointer unicalPtrName1== unicalPtrName2;

father(p,q)	**setPointer** unicalPtrName1=p;
	setPointer unicalPtrName2=q;
	link unicalPtrName1.father.unicalPtrName2;

$(\text{father}(p,q) \wedge =_{Integer}(\text{age}(p),18))$	**setPointer** unicalPtrName1=p;
	setPointer unicalPtrName2=q;
	link unicalPtrName1.father.unicalPtrName2;
	setPointer unicalPtrName3=p;
	attr unicalPtrName3.age==18;

$((father(p,q) \wedge =_{Integer}(age(p), 18)) \vee =_{Integer}(age(q),18))$	**setPointer** unicalPtrName1=p;
	setPointer unicalPtrName2=q;
	link unicalPtrName1.father.unicalPtrName2 **else** unicalLabel;
	setPointer unicalPtrName3=p;
	attr unicalPtrName3.age==18 **else** unicalLabel;
	goto endLabel;
	label unicalLabel;
	setPointer unicalPtrName4=q;
	attr unicalPtrName4.age==18;
	label endLabel;

$\neg =_{Integer}(age(p),18)$	**setPointer** unicalPtrName1=p;
	attr unicalPtrName1.age==18 **else** unicalLabel;
	goto;
	label unicalLabel;

$\exists_{Person} p\ (=_{Integer}(age(p),18))$	**first** p:Person **suchthat**
	begin
	  setPointer unicalPtrName1=p;
	  attr unicalPtrName1.age==18;
	end;

$\forall_{Person} p\ (=_{Integer}(age(p),18)) \equiv \neg\exists_{Person} p\ (\neg =_{Integer}(age(p),18))$	**first** p:Person **suchthat**
	begin
	  setPointer unicalPtrName1=p;
	  attr unicalPtrName1.age==18 **else** unicalLabelForA;
	  goto;
	  label unicalLabelForA;
	end else unicalLabel;
	goto;
	label unicalLabel;

Although the construction of *begin-end* expressions is inductive in most cases, it is easy to see that it is indeed possible to construct a *begin-end* expression with the same truth value as that of the given P^- formula in each case. End of proof.

Actually, *begin-end* expressions are even more powerful than the predicate language mentioned above. This is so mainly because of three reasons [10]:

1) it is possible to operate with variables of primitive types in *begin-end* expressions;
2) a *begin-end* expression specifies the command execution order during the pattern matching (i.e., the order in which instances are traversed);
3) when a pattern is matched, all its elements are assigned an identity which can be used further for referencing these elements.

3.4 Transformation Language L2

The new feature of the language L2 if compared to the language L1 is the possibility to make loops. A special command exists in L2 with which it is possible to visit either all instances of the specified class or just those instances of the class that match the given pattern.

In the metamodel of L2, one class is added ("ForeachCom") if compared to the metamodel of L1 (Fig. 6, bold class and associations are new in L2). Two associations from this class to the class "ComBlock" exist – one for the commands of the loop and the other for the pattern definition block of the loop.

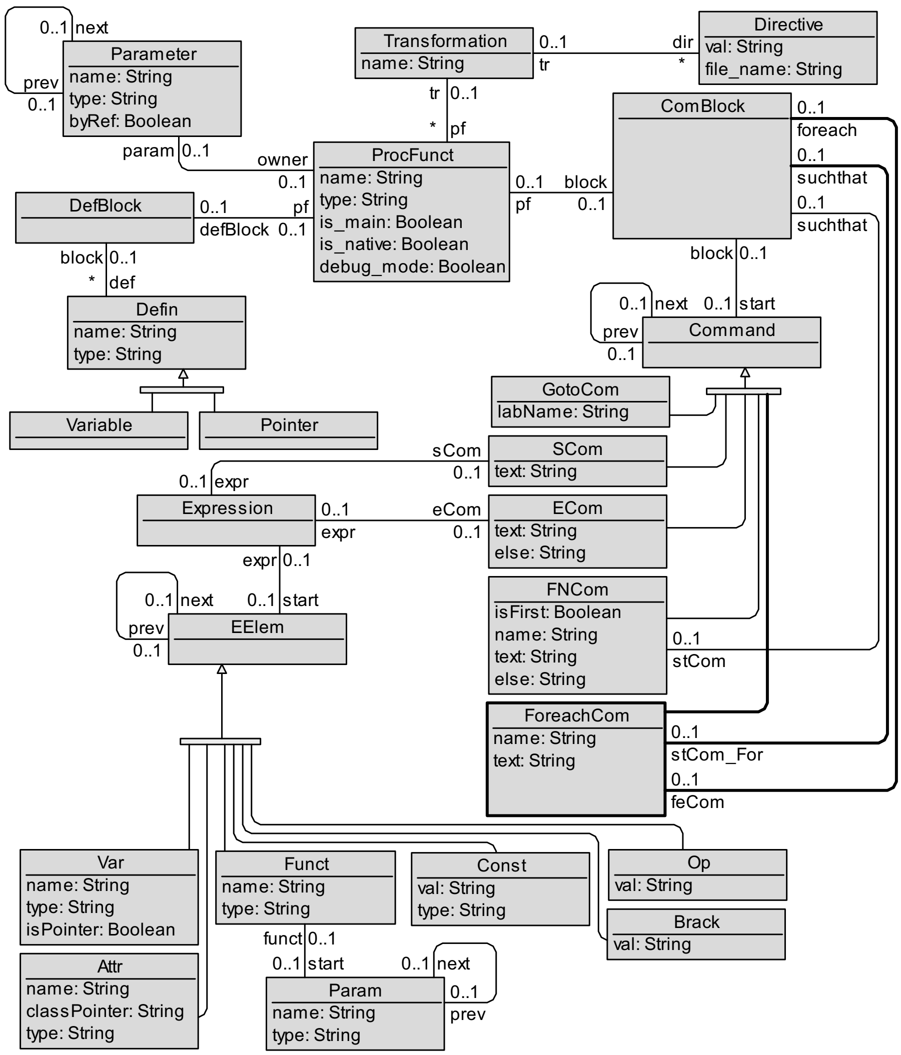

Fig. 6. The metamodel of the transformation language L2

The textual syntax for a loop command is as follows:

```
foreach <pointerName1> : <className> [ from
  <pointerName2> by <roleName> ] [ suchthat
begin
    <L2Commands>
end ]
do
begin
    <L2Commands>
end;
```

The semantics of the "suchthat" block is the same as in the case of the language L1. Since this block is optional, the semantics of the "foreach" command is as follows – every instance of the specified class that matches the given pattern (if such exists; otherwise it is considered that every instance is to be taken) is traversed and all the commands of the "do" block are executed for it.

Let's consider some examples. Increase the value of the attribute "age" of all instances of the class "Person" by 1 (all examples in this section are based on the metamodel seen in Fig. 4):

```
foreach p:Person do
begin
 setAttr p.age=p.age+1;
end;
```

Increase the age of all persons younger than 18 by 1:

```
foreach p:Person suchthat
begin
 attr p.age<18;
end
do
begin
 setAttr p.age=p.age+1;
end;
```

Nested loop example – set the value of the attribute "hasParentUnder18" to *true* for those persons that are sons of a person younger than 18:

```
foreach parent:Person suchthat
begin
 attr parent.age<18;
end
do
begin
 foreach p:Person suchthat
 begin
  link parent.son.p;
 end
 do
 begin
  setAttr p.hasParentUnder18=true;
 end;
end;
```

The transformation that solves the problem proposed in Section “2.3. Model transformation example in the language L0” can resemble this in the language L2:

```
transformation example;
 main procedure main();
  pointer c:Course;
  pointer s:Student;
  var x:Real;
  var count:Integer;
 begin;
  first c:Course suchthat
  begin
   attr c.title=="Operating Systems";
  end else endOfProg;
  setVar x=0;
  setVar count=0;
  foreach s:Student from c by student suchthat
  begin
   attr s.age>=18;
  end
  do
  begin
   setVar count=count+1;
   setVar x = x + ( s.mark1:Real + s.mark2:Real +
     s.mark3:Real + s.mark4:Real + s.mark5:Real +
     s.mark6:Real + s.mark7:Real + s.mark8:Real ) / 8;
  end;
  var count>0 else writeGood;
  setVar x=x/count;
  var x<8 else writeGood;
  setAttr c.hasGoodStudents=false;
  goto endOfProg;
  label writeGood;
  setAttr c.hasGoodStudents=true;
  label endOfProg;
 end;
endTransformation;
```

3.5 Transformation Language L3

The new feature of the language L3 if compared to the language L2 is the branching command – a standard “if-then-else” construction than can be used instead of constructions made using “goto” commands in some cases.

A new class is added in the metamodel of L3 if compared to the metamodel of L2 – “IfCom” (Fig. 7, bold class and associations are new in L0’). Three associations from this command to the class “ComBlock” exist – one for the “if” clause, one for the “then” clause, and one for the “else” clause of the command.

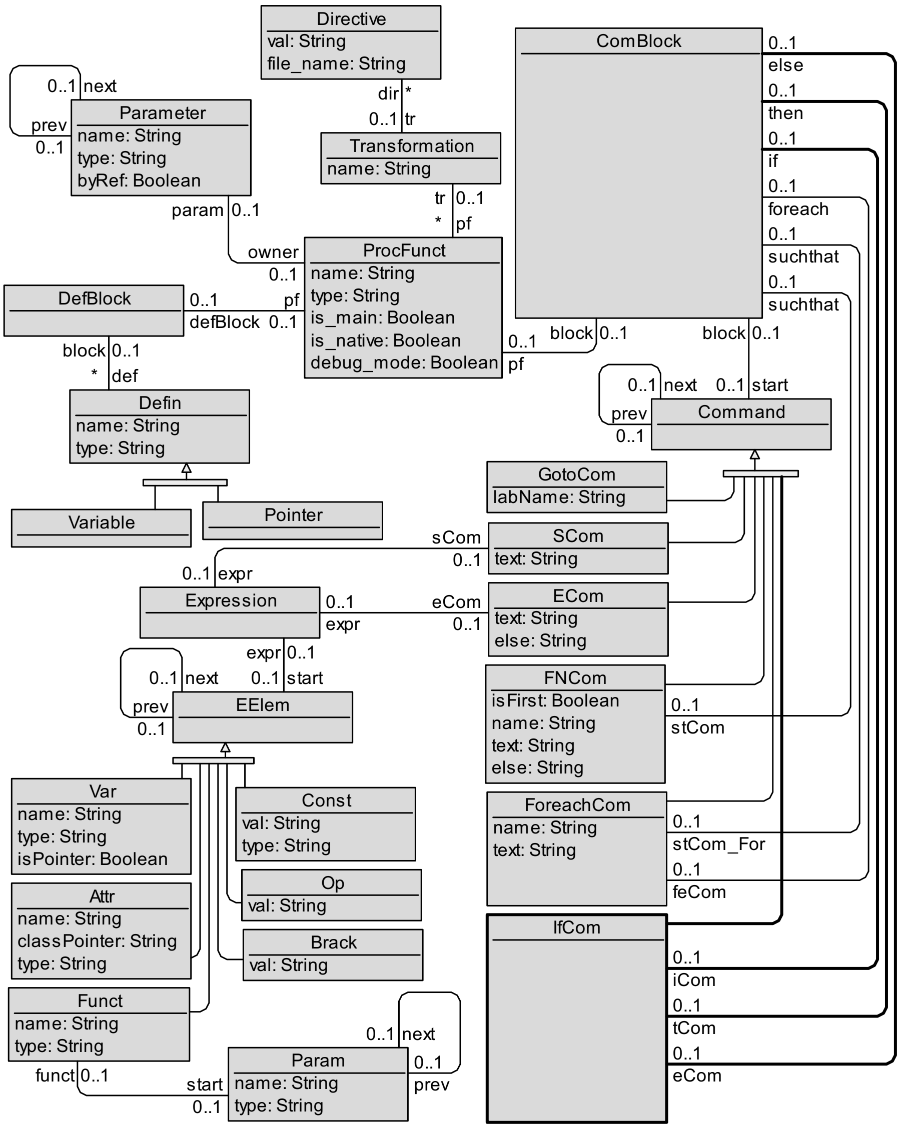

Fig. 7. The metamodel of the transformation language L3

The situation with “then” and “else” blocks is intuitively quite clear – these blocks must contain commands to be executed in the case of respectively true and false value of some condition. But what about the “if” block? This is again the case of *begin-end* expressions – an expression is attached to the “if” clause of an “IfCom” command, and so the condition of the “IfCom” command is true if the respective *begin-end* expression is true.

The textual syntax of the branching command is as follows:

```
if
begin
     <L3Commands>
end
then
begin
     <L3Commands>
end
[ else
begin
     <L3Commands>
end ];
```

Since the "else" part is optional, it is possible that no commands are to be executed in the case of false value of the condition.

Let's consider some examples. Let's assume we have a pointer p pointing to some instance of the class "Person". Increase the value of the attribute 'age" of this instance by 1 if it is less than 18 (all examples in this section are based on the metamodel shown in Fig. 4):

```
if
begin
 attr p.age<18;
end
then
begin
 setAttr p.age=p.age+1;
end;
```

Increase the age of the person pointed to by p by 1 if it is less than 18, otherwise decrease it by 1:

```
if
begin
 attr p.age<18;
end
then
begin
 setAttr p.age=p.age+1;
end
else
begin
 setAttr p.age=p.age-1;
end;
```

A more complicated example – assign a value "Less than hundred" or "Hundred or more" to a *String* variable *s* based on the fact whether the total age of all persons younger than 18 is less than 100 or not:

```
if
begin
 setVar sum=0;
 foreach p:Person suchthat
 begin
```

```
    attr p.age<18;
   end
   do
   begin
    setVar sum=sum+p.age;
   end;
   var sum<100;
  end
  then
  begin
   setVar s="Less than hundred";
  end
  else
  begin
   setVar s="Hundred or more";
  end;
```

In this example, the value of the variable *sum* could also be calculated before the branching command, however, it can easily be done in the same command when the semantics of the variable tells it is only needed in the “IfCom” command.

The transformation that solves the problem proposed in Section “2.3. Model transformation example in the language L0” can resemble this in the language L3:

```
transformation example;
 main procedure main();
  pointer c:Course;
  pointer s:Student;
  var x:Real;
  var count:Integer;
 begin;
  first c:Course suchthat
  begin
   attr c.title=="Operating Systems";
  end else endOfProg;
  setVar x=0;
  setVar count=0;
  foreach s:Student from c by student suchthat
  begin
   attr s.age>=18;
  end
  do
  begin
   setVar count=count+1;
   setVar x = x + ( s.mark1:Real + s.mark2:Real +
     s.mark3:Real + s.mark4:Real + s.mark5:Real +
     s.mark6:Real + s.mark7:Real + s.mark8:Real ) / 8;
  end;
  if
  begin
   var count>0;
```

```
      setVar x=x/count;
      var x<8;
     end
     then
     begin
      setAttr c.hasGoodStudents=false;
     end
     else
     begin
      setAttr c.hasGoodStudents=true;
     end;
     label endOfProg;
    end;
   endTransformation;
```

A full syntax definition of the transformation language L3 based on *Backus-Naur* notation [12] is given in Appendix A.

4 The implementation of model transformation languages L0', L1, L2, and L3

4.1 The Main Principles of the Implementation of L0' Until L3

As mentioned in the first sections of this paper, there already exists an effective implementation of the language L0, that is, a compiler to the language C++. Since languages L0', L1, L2, and L3 have been built based on the language L0, a very logical step would be the implementation of these languages through the language L0. So actually the basic idea of the implementation of languages L0' until L3, is to build a compiler for each of the languages L0', L1, L2, and L3 to L0. However, for the task to be accomplished more easily, each compiler will be built to the language that is a direct ancestor to it instead of building each compiler to the language L0. Since every next language in the Lx family was built based on the previous one by just adding some new features, such an implementation is possible. The algorithm used to implement these compilers is called the bootstrapping algorithm [13]. The bootstrapping principle is to build a compiler of one language to the other language that is written in the target language. So, translating this into the language of Lx – to write a compiler in L0 that transforms an L0' program into an L0 program, then to write a compiler in L0' that transforms an L1 program into an L0' program, then to write a compiler in L1 that transforms an L2 program into an L1 program, and finally to write a compiler in L2 that transforms an L3 program into an L2 program. When turning to the details of the actual implementation, it is worth mentioning that each compiler can actually be written in L0 (because L0 is a subset of every other Lx language). In doing so, the idea of bootstrapping algorithm is not violated – it can still be considered that each compiler is written in the target language perhaps just without using all features the language offers.

In contrast with the ideas of the traditional compiler building [14] in which an analysis of textual forms of programs is taken for a base, the idea of the bootstrapping algorithm is very convenient in the case of model transformation languages – any program in a model transformation language is considered as a particular model in the metamodel of that particular language. If both metamodels of the source and target languages are known, as well as the particular source model corresponding to the program to be compiled is known, the compilation task transforms into a standard model transformation task.

When consideringthe possibility to implement the bootstrapping algorithm, it must be verified whether it is really possible for each program of the source language to make an equivalent program in the target language (this could not be possible due to new features of the target language). Therefore it must be verified for the case of languages Lx immediately.

The main question about the compiler from L0’ to L0 is whether it is possible for every arithmetic expression of L0’ to build an equivalent construction in L0. It is easy to see that every such expression can be divided into some binary or unary expressions that are allowed in L0 (by assigning intermediate results to temporary variables). Consequently it is really possible to build a compiler from L0’ to L0.

In L1, a pattern matching is possible. The question that arises – is each L1 pattern block translatable into L0’? The answer is – yes. Every pattern block can be simulated with L0’ commands (see Table 5). The general idea is to make some extra labels and to add a label to commands without any labels to intercept those control flows that correspond to the situation in L1 the pattern matching fails.

Table 5. The principle of the implementation of a pattern definition block

L1	L0’

first <ptrName1>**:**<className>
[**from** <ptrName2> **by** <roleName>]
suchthat
begin
  <command_1>;
  <command_2>;
  ...
  <command_n>;
end
[**else** <labelName>];
	first <ptrName1>**:**<className> [**from** <ptrName2> **by** <roleName>] [**else** <labelName>];
	label ___L_i;
	<command_1> [**else** ___L_i+1];
	<command_2> [**else** ___L_i+1];
	...
	<command_n> [**else** ___L_i+1];
	goto ___L_i+2;
	label ___L_i+1;
	next <ptrName1> [**else** <labelName>];
	goto ___L_i;
	label ___L_i+2;

next <ptrName> **suchthat**
begin
  <command_1>;
  <command_2>;
  ...
  <command_n>;
end
[**else** <labelName>];
	next <ptrName> [**else** <labelName>];
	label ___L_i;
	<command_1> [**else** ___L_i+1];
	<command_2> [**else** ___L_i+1];
	...
	<command_n> [**else** ___L_i+1];
	goto ___L_i+2;
	label ___L_i+1;

	next <ptrName> [**else** <labelName>];
	goto ___L_i;
	label ___L_i+2;

Of course, “else” branches are attached to only those commands which can (but do not) have an “else” branch. A label in the form “___L_i” is considered to be a new label that cannot be found in any L0’ program written by the user. If a command of the pattern definition block proves to be an instance searching command with a “suchthat” block, it needs to be expanded to L0’ commands recursively.

The question about the language L2 – is every “foreach” loop writable in L1? Again, the answer is implicitly clear – it is! Since there are commands “goto” and “label” in L1, as well as “else” branches, the control flow can be moved around to one’s liking, inter alia, making a loop over either all instances of some particular class, or just those instances that match the specified pattern (pattern matching constructions are present in L1, so they do not need to be transformed in any way). The scheme of the implementation of loops is shown in Table 6.

Table 6. The principle of the implementation of a “foreach” loop

L2	L1

foreach <ptrName> **:** <className>
[**suchthat**
begin
  <command_1>;
  <command_2>;
  ...
  <command_n>;
end]
do
begin
  <do_command_1>;
  <do_command_2>;
  ...
  <do_command_k>;
end;
	first <ptrName> **:** <className>
	[**suchthat**
	begin
	<command_1>;
	<command_2>;
	...
	<command_n>;
	end]
	else __L_i;
	label __L_i+1;
	<do_command_1>;
	<do_command_2>;
	...
	<do_command_k>;
	next <ptrName> [**suchthat**
	begin
	<command_1>;
	<command_2>;
	...
	<command_n>;
	end]
	else __L_i;
	goto __L_i+1;
	label __L_i;

Again, labels in form “__L_i” are not supposed to be found in any L1 program written by the user.

The question about the language L3 – is every branching construction writable in L2? The answer is even more obvious in this case than in the previous ones – every branching construction can be simulated using "goto" and "label" commands in a standard way (Table 7). As shown above, "else" branches are added here to intercept the cases when the "if" condition (*begin-end* expression) fails.

Table 7. The principle of the implementation of a branching command

L3	L2
if	<if_command_1> [**else** _L_i];
begin	<if_command_2> [**else** _L_i];
<if_command_1>;	...
<if_command_2>;	<if_command_n> [**else** _L_i];
...	<then_command_1>;
<if_command_n>;	<then_command_2>;
end	...
then	<then_command_k>;
begin	**goto** _L_i+1;
<then_command_1>;	**label** _L_i;
<then_command_2>;	[<else_command_1>;
...	<else_command_2>;
<then_command_k>;	...
end	<else_command_l>;]
[**else**	**label** _L_i+1;
begin	
<else_command_1>;	
<else_command_2>;	
...	
<else_command_l>;	
end];	

Again, labels in form "_L_i" are not supposed to be found in any L2 program written by user.

4.2 Main Problems of the Implementation of L0' Until L3

When compiling a program from one language to another, new "label" commands may appear that must be unique in the scope of the whole particular procedure/function. Therefore, some limitations of naming conventions must exist. In the case of languages L0' until L3 these limitations are as follows – the user cannot make labels in L1 starting with three underscores ("_"), in L2 – starting with two underscores, and in L3 – starting with at least one underscore. If so, compilers of L1, L2, and L3 can make new labels starting with some underscores (so many that the user is allowed to make such a label in the target language, but is not allowed in the source language) and following by the symbol "L" and an integer uniquely generated for every new label. In the case of the L0' compiler, no labels are made, so no action needs to be performed.

In the case of L0’ compilation, some new variables may appear that must be unique in the scope of the whole particular procedure/function. Therefore, some limitations of naming conventions must exist. The limitation offered is similar to the case of labels – to forbid declaring variables starting with an underscore in languages L0’, L1, L2, and L3. So the compiler of L0’ will make variables starting with an underscore and followed by a string “var” and an integer.

When compiling an L2 program to an L1 program, two instance searching commands (“first” and “next”) arise from each loop command. If the loop command contains a pattern definition block, both instance searching commands will contain this block as well. In terms of models, this means that there is one command block assigned to two “FNCom” commands. When passing to the next step – the compilation of L1 – a problem arises – a compiler processes the first of two instance searching commands mentioned above and then deletes the pattern definition block from the metamodel (in order to make the model respective to L0’ program). So the information about the fact that the other command had this block as well is lost. Hence to solve this problem, it is forbidden to have one command block attached to more than one command. Therefore, in the compilation of L2, a copy of the command block must be made. It means that a copy of each command of the particular block must be made. There is no problem in regard to other commands, but a problem arises when “label” commands come in place – how to preserve the uniqueness of labels? The solution here is to make a new unique label from the old unique label by simply concatenating the label name with itself. For example, if the old label name was “__L17”, the new one is “__L17__L17”; if the old one was “myLabel”, the new one is “myLabelmyLabel”. The same conventions relate to “goto” commands and “else” branches as well.

4.3 The Whole Compilation Process – From L3 Up To L0

Before turning to the compilation process, one more concept needs to be explained, and that concept is the lexem of the transformation language L3. The metamodel of lexems – basic syntactical elements of a program – is very simple (Fig. 8). Lexems are one of the intermediate steps in the full compilation process from L3 up to L0.

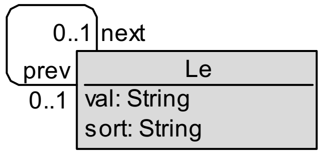

Fig. 8. The metamodel of lexems

The full compilation process transforming one text file to another consists of the following components:

1) initialization tasks – the deletion of any old models of metamodels of L3 and lexems, and the generation of the new model of lexems from the input text file specified by the user;

2) transformation of the model of lexems into an L3 model;
3) transformation of the L3 model into an L2 model;
4) transformation of the L2 model into an L1 model;
5) transformation of the L1 model into an L0' model;
6) transformation of the L0' model into an L0 model;
7) printing of the L0 model to an output text file specified by the name of the transformation.

Owing to the fact that pre- and post-conditions of each component are precisely clear, it is possible to easily use just a subset of all components instead of using the full compiler as well. It can be useful, for example, in cases when L3 is used as an intermediate language in the compilation of some higher-level language as it is done in the compilation of the graphical model transformation language MOLA [15] – there is no need to perform neither initialization tasks nor the "lexems to L3" transformation because the L3 model has got into the metamodel of L3 in some other way.

The full L3 to L0 compiler is available in the Lx homepage [16].

5 Conclusions and future work

It has been evidenced that the bootstrapping method justifies itself in the use of compiler building if operating with model transformation languages. With this method, it is possible to build higher and higher-level model transformation languages that are easy to compile to some lower-level language. The sequence of such languages – the Lx language family – has been stopped at the language L3 which is of sufficiently high level to be used in practical model transformation tasks. As a proof of this, a transformation-based graphical tool-building platform GrTP is being developed using the language L3 as its base language [17]. The other use case of the language L3 is the implementation of even higher-level languages. The graphical model transformation language MOLA [6,7] has been implemented by bootstrapping method, using the language L3 as an intermediate language in this process [15].

The further work relating to the Lx language family includes, but is not limited to supplementing languages L0', L1, L2, and L3 with the features of the language L0+ – the extended version of L0 [1].

References

1. S. Rikacovs, *The base transformation language L0+ and its implementation*, Scientific Papers, University of Latvia, "Computer Science and Information Technologies", 2008.
2. *MDA Guide Version 1.0.1.* OMG, document omg/03-06-01, 2003.
3. E.D.Willink, *UMLX - A Graphical Transformation Language for MDA*, 2nd OOPSLA Workshop on Generative Techniques in the context of Model Driven Architecture , OOPSLA'2003, Anaheim, 2003.
4. T. Clark, A. Evans, P. Sammut, J. Willans. *Language Driven Development and MDA*, BPTrends, MDA Journal, Oct 2004.

5. J. Bezivin, E. Breton, G. Dupe, P. Valduriez. *The ATL Transformation-based Model Management Framework*, Research Report No 03.08, 2003, IRIN, Universite de Nantes.
6. A. Kalnins, J. Barzdins, E. Celms, *Model Transformation Language MOLA*. Proceedings of MDAFA 2004, Vol. 3599, Springer LNCS, 2005, pp. 62-76.
7. MOLA project, http://mola.mii.lu.lv
8. J. Barzdins, G. Barzdins, R. Balodis, K. Cerans, A. Kalnins, M. Opmanis, K. Podnieks, *Towards Semantic Latvia*. Communications of the 7th International Baltic Conference on Databases and Information Systems (Baltic DB&IS'2006), Vilnius, 2006, pp. 203-218.
9. The Base Transformation Language L0, http://lx.mii.lu.lv/L0_plus_CurrVers_2_4.pdf
10. J. Barzdins, A. Kalnins, E. Rencis, S. Rikacovs, *Model Transformation Languages and their Implementation by Bootstrapping Method.* Pillars of Computer Science, Vol. 4800, Springer LNCS, 2008, pp. 130-145.
11. J. Gallier, *Logic for Computer Science: Foundations of Automatic Theorem Proving*, Wiley, 1986.
12. L.M. Garshol, *BNF and EBNF: What are they and how do they work?*, http://www.garshol.priv.no/download/text/bnf.html
13. B. Efron, R.J. Tibshirani, *An Introduction to the Bootstrap*, Chapman & Hall/CRC, 1994, p. 436.
14. A.V. Aho, R. Sethi, J.D. Ullman, *Compilers - Principles, Techniques, and Tools*, Addison-Wesley Publishing Company, 1986, p. 796.
15. A. Sostaks, A. Kalnins, *The implementation of MOLA to L3 compiler*, Scientific Papers, University of Latvia, "Computer Science and Information Technologies", 2008.
16. The Lx transformation language set home page, http://Lx.mii.lu.lv
17. J. Barzdins, A. Zarins, K. Cerans, A. Kalnins, E. Rencis, L. Lace, R. Liepins, A. Sprogis, *GrTP: Transformation Based Graphical Tool Building Platform*, MODELS 2007, Workshop on Model Driven Engineering Languages and Systems, 2007.

Appendix

A. L3 syntax definition based on Backus-Naur notation

```
<L3Program>         ::= <transformation> [ <L3Program> ]
<transformation>    ::= transformation <identifier> ;
<transfPartList> endTransformation;
<identifier>        ::= <letter> [ <string> ]
<specialID>         ::= <specialLetter> [ <string> ]
<string>            ::= <letter> [ <string> ] | <digit> [
<string> ]
<letter>            ::= <specialLetter> | _
<specialLetter>     ::= a | b | c | d | e | f | g | h | i |
j | k | l | m | n | o | p | q | r | s | t | u | v | w | x
```

```
| y | z | A | B | C | D | E | F | G | H | I | J | K | L |
M | N | O | P | Q | R | S | T | U | V | W | X | Y | Z
<digit>          ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
9
<transfPartList>  ::= <transfPart> [ <transfPartList> ]
<transfPart>      ::= <nativeProc> | <nativeFunct> |
<directive> | <debug> | <varDeclaration> | <procedure> |
<function>
<nativeProc>      ::= native procedure <identifier> ( [
<paramList> ] );
<paramList>       ::= <parameter> [ , <paramList> ]
<simpleParamList> ::= <identifier> [ , <simpleParamList>
]
<parameter>       ::= <identifier> : <primTypeOrMMElem>
<nativeFunct>     ::= native function <identifier> ( [
<paramList> ] ): <primTypeOrMMElem> ;
<directive>       ::= <dirType> " <fileName> ";
<dirType>         ::= useMM | include | useLib | useUnit
<fileName>        ::= [ <specialLetter> :\] [
<folderList> ] <string> [ . <string> ]
<folderList>      ::= <string> \ [ <folderList> ]
<debug>           ::= DEBUG_ON; | DEBUG_OFF;
<varDeclaration>  ::= <primitiveVarDecl> | <pointerDecl>
<primitiveVarDecl>::= var <specialID> : <primTypeName> ;
<primTypeName>    ::= Integer | Real | String | Boolean
<pointerDecl>     ::= pointer <identifier> :
<metaModelElement> ;
<metaModelElement>::= <letter> [ <stringPlus> ]
<stringPlus>      ::= <stringPlusElem> [ <stringPlus> ]
<stringPlusElem>  ::= <letter> | <digit> | # | ::
<procedure>       ::= [ main ] procedure <identifier> ( [
<paramList> ] ); [ <varList> ] begin; [ <L3CommandList> ]
end;
<function>        ::= function <identifier> ( [
<paramList> ] ): <primTypeOrMMElem> ; [ <varList> ]
begin; [ <L3CommandList> ] end;
<primTypeOrMMElem>::= <primTypeName> | <metaModelElement>
<varList>         ::= <varDeclaration> [ <varList> ]
<L3CommandList>   ::= <L3Command> [ <L3CommandList> ]
<L3Command>       ::= <call> | <return> | <first> |
<next> | <goto> | <label> | <addObj> | <addLink> |
<deleteObj> | <deleteLink> | <setPointer> | <setPointerF>
| <setVar> | <setVarF> | <setAttr> | <type> | <var> |
<pointer> | <attr> | <link> | <noLink> | <debug> |
<foreach> | <if>
<call>            ::= call <identifier> ( [
<simpleParamList> ] );
<return>          ::= return [ <identifier> ] ;
```

```
<first>           ::= first <identifier> :
<metaModelElement> [ from <identifier> by
<metaModelElement> ] [ suchthat begin [ <L3CommandList> ]
end ] [ else <specialID> ] ;
<next>            ::= next <identifier> [ suchthat begin
[ <L3CommandList> ] end ] [ else <specialID> ] ;
<goto>            ::= goto [ <specialID> ] ;
<label>           ::= label <specialID> ;
<addObj>          ::= addObj <identifier> :
<metaModelElement> ;
<addLink>         ::= addLink <identifier> .
<metaModelElement> . <identifier> ;
<deleteObj>       ::= deleteObj <identifier> ;
<deleteLink>      ::= deleteLink <identifier> .
<metaModelElement> . <identifier> ;
<setPointer>      ::= setPointer <identifier> =
<identifier> ;
<setPointerF>     ::= setPointerF <identifier> =
<identifier> ( [ <simpleParamList> ] );
<setVar>          ::= setVar <specialID> = <expression> ;
<setVarF>         ::= setVarF <specialID> = <identifier>
( [ <simpleParamList> ] );
<setAttr>         ::= setAttr <identifier> .
<metaModelElement> = <expression> ;
<type>            ::= <identifier> <pointerRelOp>
<metaModelElement> [ else <specialID> ] ;
<var>             ::= <identifier> <relationOperator>
<expression> [ else <specialID> ] ;
<pointer>         ::= <identifier> <pointerRelOp>
<identifier> [ else <specialID> ];
<attr>            ::= attr <identifier> .
<metaModelElement> <relationalOperator> <expression> [
else <specialID> ];
<link>            ::= link <identifier> .
<metaModelElement> . <identifier> [ else <specialID> ] ;
<noLink>          ::= noLink <identifier> .
<metaModelElement> . <identifier> [ else <specialID> ] ;
<foreach>         ::= foreach <identifier> :
<metaModelElement> [ from <identifier> by
<metaModelElement> ] [ suchthat begin [ <L3CommandList> ]
end ] do begin [ <L3CommandList> ] end;
<if>              ::= if begin [ <L3CommandList> ] end
then begin [ <L3CommandList> ] end [ else begin [
<L3CommandList> ] end ] ;
<expression>      ::= <boolExpr> | <notBoolExpr>
<boolExpr>        ::= true | false
<notBoolExpr>     ::= <exprPart> [ <arithmeticOper>
<notBoolExpr> ]
```

```
<exprPart>         ::= <const> | <identifier> |
<attribute> | <functionCall>
<arithmeticOper>  ::= + | - | * | /
<const>            ::= <integerConst> | <realConst> |
<stringConst>
<integerConst>    ::= <positiveNum> | (- <positiveNum> )
<positiveNum>     ::= <digit> [ <positiveNum> ]
<realConst>       ::= <positiveNum> . <positiveNum> | (-
<positiveNum> . <positiveNum> )
<stringConst>     ::= ″ <extendedString> ″
<extendedString>  ::= <symbol> [ <extendedString> ]
<symbol>           ::= <letter> | <digit> |
<relationOperator> | ~ | ` | ! | @ | # | $ | % | ^ | & |
( | ) | { | } | : | < | > | ? | [ | ] | ; | ' | \ | , | .
| <space>
<attribute>       ::= <identifier> . <metaModelElement> :
<primTypeName>
<functionCall>    ::= <identifier> ( [ <simpleParamList>
] )
<relationOperator>::= <pointerRelOp> | < | > | <= | >=
<pointerRelOp>    ::= == | !=
<space>            ::=
```

Note – the non-terminal symbol <space> is considered to be a space symbol (32th symbol in the ASCII code table).