LATVIJAS UNIVERSITĀTES RAKSTI. 2008, 733. sēj.:
DATORZINĀTNE UN INFORMĀCIJAS TEHNOLOĢIJAS 75.–102. lpp.

The Base Transformation Language L0+ and Its Implementation

Sergejs Rikacovs

University of Latvia, IMCS, 29 Raiņa blvd, Rīga, Latvia
sergejs.rikacovs@lumii.lv

Abstract. An efficient implementation of high level model transformation languages is well known as a complex problem. It is believed that the most appropriate way to implement transformation languages is bootstrapping. However, bootstrapping is not possible without an efficient base language. In this paper, a new low level model transformation language L0+ is proposed, for which there exists an efficient implementation. This language can be used as a base language in the bootstrapping process. L0+ does not have advanced pattern definition facilities, but the expressive power of this language is comparable to some more advanced languages. In spite of the fact that L0+ is quite a low level language, it can also be used for the development of model transformations directly. The presented paper is an extended version of the second chapter of [1].

Keywords: model transformation language, compiler, bootstrapping.

1 Introduction

During the last few years a new approach to complex system building was developed – MDA. Model transformations are considered to be one of the pillars of this approach [2].

Model transformation languages are a comparatively new kind of languages. The first standardization effort in this area was OMG MOF 2.0 Query/Views/Transformations (QVT) request for Proposal (RFP) [3]. In response to this request, QVT specification was developed [4]. QVT defines the standard way of defining transformations. According to this specification, source and target models should correspond to the MOF metamodel. QVT consists of three sublanguages:

- Relations – a high level declarative language;
- Core – also a declarative language, but it is more verbose and does not provide an implicit creation of trace instances, the semantics of the Relations language is defined in terms of this language;
- Operational mappings – a language allowing either to define transformations using a fully imperative approach (operational transformations), or to complement relational transformations with imperative operations implementing the relations (hybrid approach).

There are several independent model transformation languages apart from QVT. These languages can be divided into two groups: graphical and textual languages. It is worth noting that QVT belongs to both groups, because of its dual forms: the graphical and the textual one. While specifying transformations in a graphical form, transformation developer has the opportunity to represent mappings between patterns of source and target models in a direct and natural way. Graphical languages, such as GReAT[5] or MOLA[6], define transformations as a set of transformation rules. Every rule has a pattern part and an action part, in which the pattern specifies instance sets to be processed according to transformations described in the rule action part. The difference between various graphical transformation languages is in the expressiveness of patterns and control flow structures. Typical representatives of textual model transformation languages are [7, 8, 9]. Most of the textual model transformation languages are declarative languages also having some kind of pattern definition facilities. A recursion is usually used as the main control flow facility.

However, the research in the area of model transformations and effective implementation of model transformation languages is still topical.

In this paper, a new low level model transformation language called L0+ is proposed. This language has two important features:

- it is very simple, so being easy learnable by transformation developers;
- it has a very efficient implementation (principles of this implementation are also described in this paper).

The **main** use case for this language is the implementation of higher level languages (through the bootstrapping approach). The so called Lx language family is implemented this way [10]. Despite the importance of the use case mentioned above, several other use cases also exist. As an example (quite a significant one), the Transformation Based Graphical Tool Building Platform [11] can be mentioned, where L0+ is used as the main language for transformation development.

A more important feature of this language is that L0+ works not only with the model as a significant part of model transformations do, but also with the metamodel level (a notable example of a transformation language containing metamodel processing facilities is Viatra [12]). More precisely, L0+ = L0 + MM, where L0 is oriented towards model processing, and MM is oriented towards metamodel processing.

2 The Base Transformation Language L0

2.1 Basic Ideas

The language L0 contains minimal but sufficient constructions for model processing:

- creation/deletion of objects;
- getting/setting the value of an attribute of an object;
- creation/deletion of links;

- iteration through instances;
- low level control flow instructions;
- labels;
- unconditional control flow transfer operator;
- conditional control flow transfer operators.

2.2 Precise Definition of L0

To give a precise definition of L0 syntax and semantics, we should precisely fix the allowed metamodeling constructs. OMG suggests using MOF 2.0 for such purposes [13]. However, more simple approaches are usually used in practice. We will follow this tradition and will use a subset of MOF 1.4[14] seen in Fig. 1. to define metamodeling constructs to be allowed in metamodel definitions.

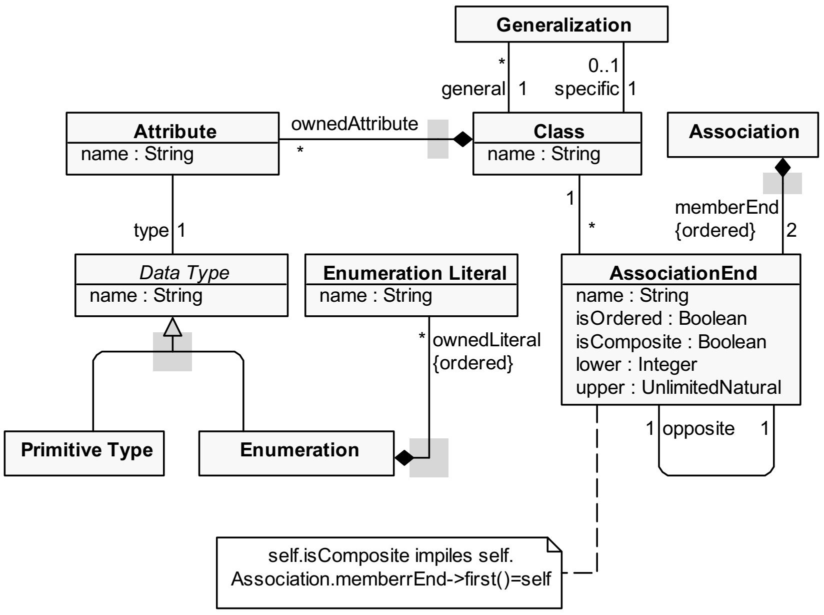

Fig. 1. Meta-metamodel

One can notice that packages are not present as an independent concept in this metamodel. In L0+, packages are simulated through qualified names.

L0 transformation program contains the following elements:

- transformation header, i.e. **transformation** <transformationName>;
- global variable definition part;
- the "useMM" directive – a path to a metamodel definition file is given through this directive. This file can contain following commands:
    - **class** <className>**;**
      Defines a class with a given name.

  - **attr** <className>.<attrName>**:**<ElementaryTypeName>**;**
    Defines an attribute with a given name and type.
  - **assoc** <className>.**{**ordered**}**<card><roleName>**/**
    <roleName><card>**{**ordered**}.**<className>**;**
    Defines an association with corresponding properties.
  - **compos** <compositeClassName>**.{**ordered**}**<card><roleName>**/**
    <roleName><card>**{**ordered**}.**<partClassName>**;**
    Defines a composition with corresponding properties.
  - **rel** <subClassName>**.subClassOf.**<superClassName>**;**
    Defines a generalization relationship between given classes.
  - **enum** <enumName>**:{** <enumLiteral1> **,** < enumLiteral2>**};**
    Defines an enumeration with given elements.
- A "native" subprogram (function or procedure) declaration part (headers of C++ functions used in the transformation). Like in every programming language, there can be some tasks for which L0 is not quite suitable (for instance, string processing, text parsing, etc.). To deal with these situations in L0, there exists a possibility to call a C++ function. For example, there is a String data type in L0, but the language per se does not define some such useful operations as Length, CharAt, and Substring on this type. If needed, transformation developers can easily implement these functions in C++ and then access them from L0 by using the concept of "native" subprograms.
- An L0 subprogram definition part (it is expected that exactly one subprogram of this part is labeled with the reserved word **main** thus defining the entry point for the transformation). An L0 subprogram definition also consists of several parts:
  - the subprogram header;
  - local variable definitions;
  - the keyword **begin**;
  - the subprogram body definition;
  - the keyword **end**;
- a transformation footer, i.e. **endTransformation;**

An elementary unit of any L0 transformation program is a **command** (an imperative statement). Before giving a detailed description of individual commands, it should be noted that the name of the meta-model element (i.e. class name, role name, attribute name, enumeration name, and enumeration literal name) can be specified in two different ways:

- as a String literal; for example, **addObj** x : Person;
- as a String variable; for example, **addObj** x : (s); In this case, the name of a meta-model element will be equal to the value of the corresponding String variable at the command execution time.

L0 contains the following commands:

1. **transformation** <transformationName>; Starts the transformation definition.
2. **endTransformation**; Ends the transformation definition.
3. **pointer** <pointerName> : <className>; Defines a pointer to an object of the class <className>.

3.1. **pointer** <pointerName> : **Void** ; Void pointer can point to objects of an arbitrary class.

4. **var** <varName> : <ElementaryTypeName>; Defines a variable of elementary data type – Boolean, Integer, Real or String.
5. **procedure** <procName>**(**<formalPrmList>**)**; Formal parameter list consists of formal parameter definitions separated by ",". A parameter definition consists of its name, the parameter type (the type can be an elementary type, a class from the meta-model or the reserved word **Void**), and the passing method (parameters can be passed by reference or by value). If the parameter is passed by reference, its type name is preceded by the **&** character.
6. **function** <funcName>**(**<formalPrmList>**)**:<returnTypeName>; Return type name can be an elementary type name, class name or the reserved word **Void**.
7. **begin**; Starts subprogram definition.
8. **end**; Ends subprogram definition.
9. **return**; Returns execution control to the caller.
10. **return** <identifier>; Returns the value of <identifier> to the caller. The type of <identifier> must coincide with the return type of the function. <identifier> is an elementary variable name or a pointer name. Instead of <identifier>, the reserved word `null` can be used. In this case function return type must be class or Void.
11. **call** <subProgName>**(**<actualPrmList>**)**; Actual parameter list can be empty. It consists of binary expressions (<binExpr>) separated by ",". More about <binExpr> can be found in the item 23.
12. **first** <pointerName> **:** <className> **else** <labelName>; Positions <pointerName> to an arbitrary [the first object (ordering is implementation dependent)] object of <className>. Typically, this command is used in combination with **next** command to traverse all objects of the given class. If <className> has no objects, <pointerName> becomes equal to null, and execution control is transferred to <labelName>. <className> in this command must be the same as or a subclass of the class used in the pointer definition. If it is a subclass, the value set of the pointer is narrowed (for the following executions of `next`).

    12.1. **first** <pointerName> : **(**<stringVarClassName>**)** **else** <labelName>;
13. **first** <pointerName>$_1$**:** <className> **from** <pointerName>$_2$ **by** <roleName> **else** <labelName>; Positions <pointerName>$_1$ to an arbitrary [the first (object ordering is implementation dependent)] object, which is reachable from <pointerName>$_2$ by a link <roleName>. Typically, this command is used in combination with the **next** command to traverse all objects connected to the given object by a link with the specified type. If there are no such objects, <pointerName>$_1$ becomes equal to null, and execution control is transferred to <labelName>. It should be noted that this command specifies (narrows) the value set of the pointer, which is taken into account when performing the **next** execution and assignment. After the command is executed, the value set of the pointer is narrowed to those objects, which are reachable from <pointerName>$_2$ by links with the given type (specified by <roleName>).

    13.1. **first** <pointerName>$_1$ **: (**<stringVarClassName>**)** **from** <pointerName>$_2$ **by** **(**<stringVarRoleName>**)** **else** <labelName>;
14. **next** <pointerName> **else** <labelName>; Gets the next object satisfying conditions formulated during the execution of "first" command and not visited

(iterated) with this variable yet. If there is no such object, <pointerName> becomes null, and execution control is transferred to <labelName>.

15. **goto** <labelName>; Unconditionally transfers control to <labelName>. <labelName> should be located in the current subprogram definition.
16. **label** <labelName>; Defines a label with the given name.
17. **addObj** <pointerName>**:**<className>; Creates a new object of the class <className>.
    17.1. **addObj** <pointerName>**:(**<stringVarClassName>**)**;
18. **addLink** <pointerName>$_1$**.**<roleName>**.**<pointerName>$_2$; Creates a new link (of type specified by <roleName>) between objects pointed to by <pointerName>$_1$ and <pointerName>$_2$, respectively.
    18.1. **addLink** <pointerName>**.(**<stringVarRoleName>**).**<pointerName>;
19. **deleteObj** <pointerName>; Deletes an object pointed to by <pointerName>.
20. **deleteLink** <pointerName>$_1$**.**<roleName>**.**<pointerName>$_2$; Deletes a link, whose type is specified by <roleName>, between objects pointed to by <pointerName>$_1$ and <pointerName>$_2$, respectively.
    20.1. **deleteLink** <pointerName>$_1$**.(**<stringVarRoleName>**).**<pointerName>$_2$;
21. **setPointer** <pointerName>$_1$**=**<pointerName>$_2$; Sets <pointerName>$_1$ to the object pointed to by <pointerName>$_2$. If the value set of <pointerName>$_1$ does not contain the object pointed to by <pointerName>$_2$, then <pointerName>$_1$ is set to **null**. In place of <pointerName>$_2$ **null** can be used. In this case <pointerName>$_2$ will not point to any object (it will point to **null**).
22. **setPointerF** <pointerName>**=**<funcName>**(**<actualPrmList>**)**; Sets <pointerName>$_1$ to the object returned by <funcName>.
23. **setVar** <varName> **=** <binExpr>; <binExpr> is a binary expression consisting of the following elements: elementary variables, subprogram parameters, literals, attribute values (<pointerName>.<attrName>) and standard operators (+,-,*,/,&&,||,!) of elementary types. Besides the traditional way (i.e. <pointerName>.<attrName>) of getting/setting values of object attributes, there is a special way to do it: <pointerName>.(<stringVarAttrName>). The result type of this operation is **String**. For example, **setVar** <varName> **=** <pointerName>**.**(<stringVarAttrName>). Here, the value of the attribute is stored as a string in <varName>.
24. **setVarF** <identifier>**=**<funcName>**(**<actualPrmList>**)**; This command can be used to obtain the result value of the function of an elementary type. Identifier is a name of a variable. Variable type must coincide with the return type of the function.
25. **setAttr** <pointerName>**.**<attrName>**=**<binExpr>; Sets the value of the attribute <attrName> of the object pointed to by <pointerName> to the value of <binExpr>.
    25.1. setAttr <pointerName>**.(**<stringVarAttrName>**) =** <stringExpr>**;**
26. **type** <pointerName> **==** <className> **else** <labelName>; If the type of the object is identical to <className>, the control is transferred to the next command, else the control is transferred to <labelName>. Instead of equality symbol **==** inequality symbol **!=** can be used. Inheritance is not taken into account (i.e. this command works as oclIsTypeOf meaning the result of the comparison will be true, if and only if types are identical).

26.1. **type** <pointerName> **==** **(**<stringVarClassName>**)** **else** <labelName>;

27. **var** <varName>**==**<binExpr> **else** <labelName>; If condition is not true, the control is transferred to <labelName>. Instead of equality symbol, other (<, <=, >, >=, !=) relational operators compatible with argument types can be used.
28. **attr** <pointerName>**.**<attrName> **==** <binExpr> **else** <labelName>; If condition is not true, the control is transferred to <labelName>. Instead of equality symbol other (<, <=, >, >=, !=) relational operators compatible with argument types can be used.
29. **link** <pointerName>**.**<roleName>**.**<pointerName> **else** <labelName>; Checks whether there is a link (which type is specified by <roleName>) between objects pointed to by <pointerName>$_1$ and <pointerName>$_2$, respectively. If condition is not true, the control is transferred to <labelName>.

    29.1. **link** <pointerName>**.(**<stringVarRoleName>**).**<pointerName> **else** <labelName>;
30. **noLink** <pointerName>**.**<roleName>**.**<pointerName> **else** <labelName>; Checks whether there is no link (its type is specified by <roleName>) between objects pointed to by <pointerName>$_1$ and <pointerName>$_2$, respectively. If condition is not true, the control is transferred to <labelName>.

    30.1. **noLink** <pointerName>**.(**<stringVarRoleName>**).**<pointerName> **else** <labelName>;
31. **pointer** <pointerName>$_1$**==**<pointerName>$_2$ **else** <labelName>; Checks whether objects pointed to by <pointerName>$_1$ and <pointerName>$_2$, respectively, are identical. Instead of **==** inequality symbol **!=** can be used. If condition is not true, the control is transferred to <labelName>. Instead of <pointer2> `null` can be used.

It is easy to see that the language L0 contains only the very basic facilities for defining transformations. At the same time, it is obviously **complete** in the sense of its functional capabilities. Namely, this is why L0 is called the base transformation language.

2.2.1 Object-Oriented L0 constructs

L0 was designed as a low level language, and originally it was not supposed to be used for direct development of transformations. However, experiments proved that it is possible to use this language for direct development of transformations without significant loss of development speed.

For example, L0 is used for development of transformations in the context of Transformation Based Graphical Tool Building Platform. The total size of the source code of transformations developed in this project exceeds 20000 lines of L0. It is clear that it is becoming more and more difficult to ensure adequate modularization of code base of such a size with the only modularization facility being the concept of sub procedure.

Today the most popular code modularization method is OO. According to [15], OO has several fundamental elements:

- Class
- Object

- Method
- Message passing
- Inheritance
- Encapsulation
- Abstraction
- Polymorphism

It is easy to see that when we are working with model transformation language we have many of these constructs readily available through the metamodel definition or because of the fact that we are working with models. However, several important concepts are absent (most notably the concepts of method, message passing and polymorphism).

To let L0 users take advantages of OO approach, L0 is supplemented with the notion of method. It can be defined in the following ways:

- **procedure** <ClassName>**::**<methodName>**(**<formPrmList>**);**
- **function** <ClassName>**::**<methodName>**(**<formPrmList>**):**<ReturnType>**;**

To reference to the object, this method is called on from the body of the method, reserved word **this** can be used.

As it can be seen, the only difference between the method declaration and the ordinary function or procedure declaration is the fact that method declaration is linked to a certain class.

In a similar way constructs for method calling are introduced:

- **call** <pointerName>**.**<methodName>**(**<actPrmList>**);**
- **setVarF** <varName> **=** <pointerName>**.**<methodName>**(**<actPrmList>**);**
- **setPointerF** <pointerName> **=** <pointerName>**.**<methodName>**(**<actPrmList>**);**

Every method call is polymorphic – it depends on the actual type of the object this particular method is called on. An example of using these new constructs can be found in section 2.3.

It should be noted that there are transformation languages providing much more advanced constructs. For example, in QVT Operational Mappings it is allowed to define the so called mapping operation. That can be defined in the following way:

mapping <dirkind> <contexttype>::<mappingname> (<parameters>,) : <result-parameters>
when {<exprs>}
where { <exprs>}
{
init { … }
population { … }
end { … }
}

A mapping operation is syntactically described by a signature, a guard (a *when* clause), a mapping body and a post-condition (a *where* clause). The **init** section contains a code to be executed before the instantiation of the declared outputs. The **population** section contains a code for populating the result parameters, and the **end** section contains an additional code to be executed before exiting from the operation. In its simplest case (in case **when** and **where** clauses are not used), a QVT mapping operation is almost equivalent to an L0 method.

2.3 Example of an L0 Transformation

Let us consider oriented graphs. **Fig. 2.** presents one possible metamodel for oriented graphs.

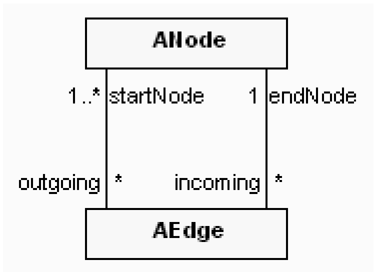

Fig. 2. Metamodel for oriented graphs

According to this metamodel, a graph in **Fig. 3.** corresponds to the instance found in **Fig. 4.**.

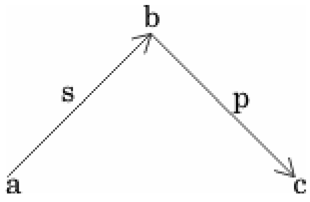

Fig. 3. An example of an oriented graph

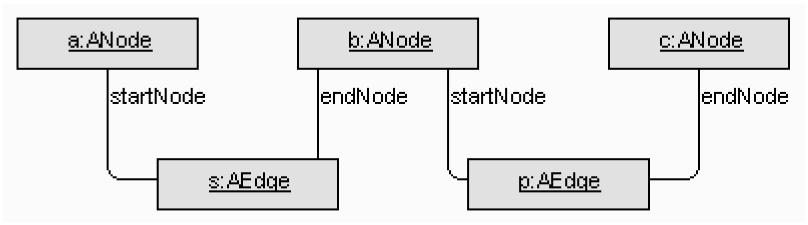

Fig. 4. Instances of the metamodel for oriented graphs

Several other metamodels (for example, the one found in **Fig. 5.**) are also possible for oriented graphs.

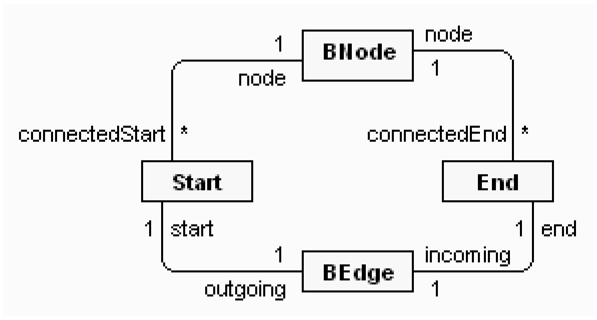

Fig. 5. Another metamodel for oriented graphs

According to this metamodel, a graph found in **Fig. 3.** will correspond to the instances found in **Fig. 6.**.

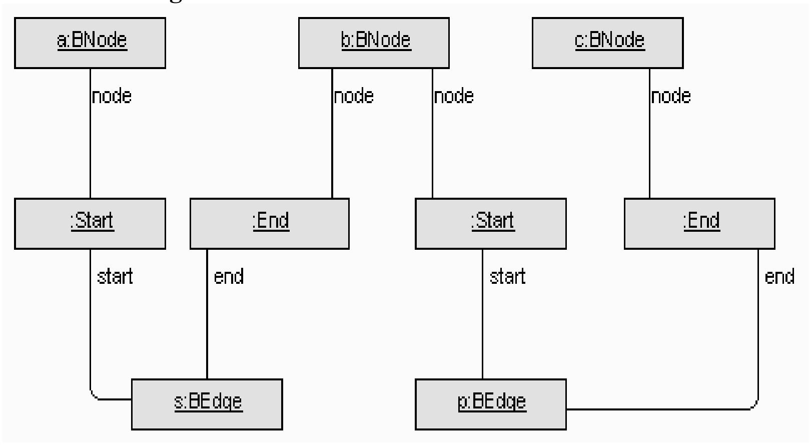

Fig. 6. Instances of another metamodel for oriented graphs

As it can be seen from the examples, we get different instances (models) for one and the same graph. At the same time it seems that these different models are quite close to each other. A natural problem arises – how to define a transformation taking a graph model corresponding to the metamodel A and producing a graph model corresponding to the metamodel B. The basic idea is to create one BNode for every ANode and to transform every AEdge to BEdge with corresponding Start and End.

To simplify this transformation we add a mapping association to the metamodel between classes ANode and BNode.

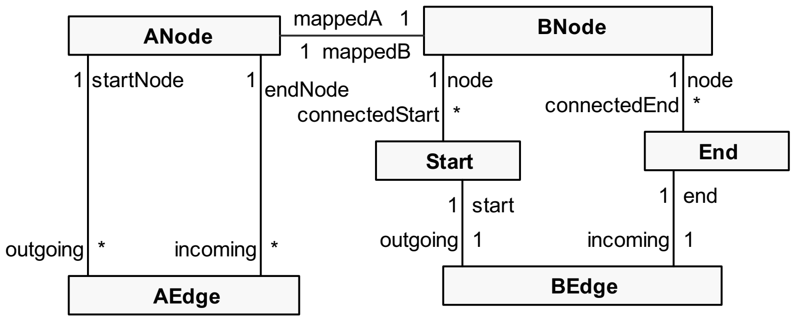

Fig. 7. Metamodel for oriented graphs with a mapping association

Transformation program in L0 implementing this algorithm can be found below.

```
transformation Graphs;
main procedure Graph2Graph();
  pointer a : ANode;
  pointer b : BNode;
  pointer aEd : AEdge;
  pointer bEd : BEdge;
  pointer edgeStart  : Start;
  pointer edgeEnd    : End;
  pointer aEdgeStNode : ANode;
  pointer aEdgeEnNode   : ANode;
  pointer mapBNode : BNode;
begin;
//copy nodes;
 first a : ANode else aNodeProcessed;
label loopANode;
   addObj  b : BNode;
   addLink a . mappedB . b;
   next a else aNodeProcessed;
   goto loopANode;
label aNodeProcessed;
//copy edges;
 first aEd : AEdge else aEdgesProcessed;
label loopAEdge;
    addObj bEd : BEdge;
    addObj edgeStart : Start;
    addObj edgeEnd : End;
    addLink bEd.start.edgeStart;
    addLink bEd.end.edgeEnd;
    //quit if not found;
    first aEdgeStNode : ANode  from aEd by startNode
    else aEdgesProcessed;
    first mapBNode : BNode  from aEdgeStNode by mappedB
    else aEdgesProcessed;
```

```
      addLink edgeStart.node.mapBNode;
    first aEdgeEnNode : ANode  from aEd by endNode
    else aEdgesProcessed;
    first mapBNode : BNode  from aEdgeEnNode by mappedB
    else aEdgesProcessed;
    addLink edgeEnd . node. mapBNode;
    next aEd else aEdgesProcessed;
    goto loopAEdge;
label aEdgesProcessed;
end;
endTransformation;
```

This transformation can be rewritten to use object-oriented L0 constructs:

```
transformation graphsOO;

procedure ANode::mapToBNode();
  pointer b : BNode;
begin;
   addObj  b : BNode;
   addLink this . mappedB . b;
end;

procedure AEdge::mapToBEdge();
  pointer bEd : BEdge;
  pointer edgeStart  : Start;
  pointer edgeEnd    : End;

  pointer aEdgeStNode : ANode;
  pointer aEdgeEnNode   : ANode;
  pointer mapBNode : BNode;
begin;
    addObj bEd : BEdge;
    addObj edgeStart : Start;
    addObj edgeEnd : End;
    addLink bEd.start.edgeStart;
    addLink bEd.end.edgeEnd;

    first aEdgeStNode : ANode  from this by startNode
    else quit;
    first mapBNode : BNode from aEdgeStNode by mappedB
    else quit;
    addLink edgeStart.node.mapBNode;
    first aEdgeEnNode : ANode  from this by endNode
    else quit;
    first mapBNode : BNode from aEdgeEnNode by mappedB
```

```
        else quit;
        addLink edgeEnd . node. mapBNode;

        label quit;
    end;

    //main procedure Graph2Graph_OO();
    procedure Graph2Graph_OO();
      pointer a : ANode;
      pointer aEd : AEdge;
    begin;

    //copy nodes;
    first a : ANode else aNodeProcessed;
    label loopANode;

       call a.mapToBNode();

       next a else aNodeProcessed;
       goto loopANode;
    label aNodeProcessed;

    //copy edges;
    first aEd : AEdge else aEdgesProcessed;
    label loopAEdge;

       call aEd.mapToBEdge();

       next aEd else aEdgesProcessed;
       goto loopAEdge;
    label aEdgesProcessed;

    end;
    endTransformation;
```

It is obvious that the body of the procedure "Graph2Graph_OO" is now more readable (and thus maintainable) than that of "Graph2Graph".

2.4 L0 and Higher Level Model Transformation Languages

One of the main features of model transformation languages is pattern definition facilities. In transformation languages, the pattern is used to select a set of objects satisfying some known constraints (there are several kinds of constraints: type constraints, attribute value constraints, and structure constraints). L0+ does not provide pattern definition facilities. It is an intentional decision, because, on the one hand, an efficient implementation of patterns is one of the main challenges in the implementation of transformation languages [16], and ,on the other hand, pattern match can be relatively easily specified with the help of L0 imperative constructs.

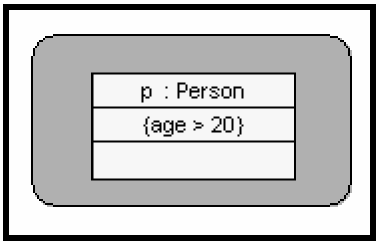

Fig. 8. Example of MOLA pattern

For instance, an example of MOLA **foreach** loop containing a pattern can be seen in Fig. **8.**. Semantically this means to iterate through all the instances of the class Person that satisfy the given attribute constraint (the value of the attribute "age" must be greater than 20). Despite the fact that L0 does not have explicit pattern definition facilities, the abovementioned MOLA pattern can be relatively easily specified in L0:

```
first p : Person else done;
  label loop_Person;
    var p.age > 20 else try_next_inst;
    //...;
    //process matched instance;
    //...;
  label try_next_inst;
  next p else done;
  goto loop_Person;
label done;
```

In more general terms, automatic pattern matching is a process that can be reduced to iteration through instances and checking a list of elementary conditions. These conditions are quite trivial – for example, check if the value of some attribute of the given object is equal to the corresponding value in the pattern specification. Another example – check whether or not there is a link with the given type between two objects. Consequently, if a language allows iterating through instances and there are conditional control flow operators, and it is possible to conduct abovementioned checks, it is possible to select a set of objects that satisfies the given constraints in this language. That allows us to assert that our language will be as powerful as a typical transformation language, but certainly transformation specification will be more verbose in this case.

3 An Extension of the Base Transformation Language L0 – Metamodel Processing Constructs

There are use cases for model transformation languages where an access not only to a model level, but also to a metamodel level is necessary. A substantial part of the existing transformation languages does not allow to process entities found at the metamodel level. To overcome this drawback, in the case of L0 we supplement it with constructs for metamodel processing, thus obtaining the language L0+.

3.1 Choosing Metamodel Processing Constructs

To allow transformation developer to process metamodels, we provide constructs to work with concepts defined in the metamodel found in figure 1. These are:

- classes
- attributes
- generalizations hierarchies
- associations
- enumerations and enumeration literals

Language users should have the possibility to create new, delete the existing and iterate through the existing elements of the metamodel. To satisfy these requirements, new commands for processing metamodel elements are introduced. These elements are identified by their names or by combinations of names.

The first activity a metamodel processing usually starts from, is the creation of the metamodel. While constructing the metamodel, the user can create classes, attributes of these classes, and associations (including composition) between classes. It is possible to create generalization hierarchies as well. In a similar way users can delete classes, attributes, associations, and generalization hierarchies. Precise syntax for these commands can be found in section 3.2.1.

The next group of commands deals with metamodel element scanning. Taking into account the fact that metamodel elements are identified by their names or by combinations of names, iterating through the elements of the metamodel actually means to iterate through the names of these elements. For example, to traverse classes of the metamodel, the commands **firstClass** and **nextClass** can be used. The semantics of these commands is close to the semantics of the ordinary **first** and **next** commands (again, precise syntax can be found in section 3.2.2.). Analogous commands are introduced for scanning each kind of metamodel elements:

- associations starting from the given class
- direct associations starting from the given class
- attributes of the given class
- subclasses of the given class
- enumerations
- enumeration elements

With the word “direct” we understand associations and attributes that are defined in the given class and not in superclasses of this class.

Using the abovementioned L0+ constructs, it is possible to dynamically explore and modify an arbitrary (potentially unknown at the compile time) metamodel. Now let us get back to the model level.

When working with a model which corresponds to a metamodel, we are not limited with only those metamodel elements that are known at the compile time. Since we have the possibility to explore an arbitrary metamodel, we need to have a way both to reference the name of an arbitrary element of this metamodel, and to reference objects of an arbitrary class (this can be done with the help of a **Void** pointer). With such a possibility it is sufficient to be able to process arbitrary models of an arbitrary metamodel.

For example, with the abovementioned L0+ constructs it is possible to create a new class at runtime and populate it with instances.

```
var currClassName : String;
pointer x   : Void;
//...;
addClass (currClassName);
addObj x : (currClassName);
//...;
```

The situation with attributes is special in some way, because problems with expression compilation can arise in case the attribute type is not known. One can notice that the type of a dynamically created attribute is unknown only at the compile time, but is known to the programmer creating this program. That is why the values of dynamically created attributes can be manipulated only as strings. To get the value of an attribute of a previously unknown type, a special form (in which the name of the attribute is specified as a String variable) of the command getting the attribute value should be used. For instance, **setVar** attrValStr = x.(attrNameStr); Here, x is a pointer name and attrNameStr is a **String** variable containing the name of the attribute. attrValStr is a String variable as well. After execution of this command, the value of attrValStr will be equal to the value of the corresponding attribute of the object encoded as a string. If attrNameStr value is equal to "weight" and x points to an object for which the value of an **Integer** attribute named "weight" is equal to 10, then attrValStr will be equal to a String value "10".

It should be noted that for **Void** pointers it is the only way to receive the value of an attribute. To simplify conversions between different representation forms of values, special built-in functions are introduced:

- IsInt (str : String) : Boolean;
- IsReal (str : String) : Boolean;
- IsBool (str : String) : Boolean;
- StrToInt (str : String) : Integer;
- StrToBool (str : String) : Bool;
- StrToReal (str : String) : Real;
- IntToStr (i : Integer) : String;
- BoolToStr (b : Bool) : String;
- RealToStr (r : Real) : String;

3.2 The Definition of Metamodel Processing Commands

Before giving a precise definition of L0+ commands, it should be stressed that the names of metamodel elements can be specified in two ways (a similar situation was with the names of metamodel elements in the case of L0):

- as literals, for instance, **addClass** Person;.
- as String variables, for instance, **addObj** x : (s); In this case the name of the metamodel element will be equal to the value of the corresponding String variable.

3.2.1 Metamodel Building Commands

This part of language definition describes commands for dynamic meta-model building.

1. **addClass** <clName>**;**
   Dynamically creates a class with a specified name. If a class with specified name already exists, a warning message is issued.

   1.1. **addClass (**<strVarClName>**);**

2. **addAttr** <clName>**.**<attrName>**:**<ElementaryTypeName>**;**
   Dynamically creates an attribute belonging to the specified class with a specified name and type. If an attribute with specified properties already exists, a warning message is issued.

   2.1. **addAttr (**<strVarClName>**).(**<strVarAttrName>**):**
   (<strVarElemTypeName>**);**

3. **addAssoc** <clName>**.{ordered}**<card><roleName>**/**
   <roleName><card>**{ordered}.**<clName>**;**
   Dynamically creates an association with specified properties. If an association with specified properties already exists, a warning message is issued.

   3.1. **addAssoc** (<strVarClName>**).{ordered}**<card>**(**<strVarRoleName>**)/**
   (<strVarRoleName>**)**<card>**{ordered}.(**<strVarClName>**);**

4. **addCompos** <compositeClName>**.{ordered}**<card><roleName>**/**
   <roleName><card>**{ordered}.**<partClName>**;**
   Dynamically creates a composition with specified properties. If a composition with specified properties already exists, a warning message is issued.

   4.1. **addCompos (**<strVarClName>**).{ordered}**<card>**(**<strVarRoleName>**)/**
   (<strVarRoleName>**)**<card>**{ordered}.(**<strVarClName>**);**

5. **addRel** <subClName>**.subClassOf.**<superClName>;
   Dynamically creates a generalization relation between the specified classes. If a generalization relation between these classes already exists, a warning message is issued.

   5.1. **addRel (**<strVarSubClName>**).subClassOf. (**<strVarSuperClName>**)**;

6. **deleteClass** <clName>**;**
   Deletes a class with a specified name.

   6.1. **deleteClass (**<strVarClName>**);**

7. **deleteAttr** <clName>**.**<attrName>**;**
   Deletes an attribute with the given properties.

   7.1. **deleteAttr (**<strVarClName>**).(**<strVarAttrName>**);**

8. **deleteAssoc** <clName>**.**<roleName>**.**<clName>**;**
   Deletes an association with a given role name between the given classes.

   8.1. **deleteAssoc (**<strVarClName>**).(**<strVarRoleName>**).(**<strVarClName>**);**

9. **deleteRel** <subClName>**.subClassOf.**<superClName>;
   Deletes a generalization relation between the specified classes.

   9.1. **deleteRel (**<strVarSubClName>**).subClassOf. (**<strVarSuperClName>**)**;

10. **addEnum** <enumName>**:{** <enumElem1>, <enumElem2>, … };
    Dynamically creates an enumeration with a specified name and specified enumeration literals.

    10.1. **addEnum (**<strVarEnumName>**):{<**enumElemName**>,**
    (strVarEnumElemName**),…};**

11. **deleteEnum** <enumName>;
    Deletes an enumeration with a specified name.

    11.1. **deleteEnum** (<strVarEnumName >);

12. **addEnumElem** <enumElemName> **to** <enumName>;
    Adds an enumeration literal to an enumeration.

    12.1. **addEnumElem** (<strVarEnumElemNameIn>) **to** (<strVarEnumNameIn>);

13. **deleteEnumElem** <enumElemName> **from** <enumName>;
    Removes an enumeration literal form an enumeration.

13.1. **deleteEnumElem** (<strVarEnumElemNameIn>) **from** (<strVarEnumNameIn>);

3.2.2 Meta-Model Element Scanning Commands

This part of language definition describes commands for scanning meta-model elements.

1. **firstClass** <strVarClNameOut> **else** <labName>**;**
   Stores the name of the first class (ordering is implementation dependent) in the <strVarClNameOut>. If there are no classes, then <strVarClNameOut> value is not changed and execution control is transferred to <labName>. Typically, this command is used in combination with the **nextClass** command to iterate through all class names.

2. **nextClass** <strVarClNameOut> **else** <labName>**;**
   Stores the name of the next class which has not yet been visited (iterated) in <strVarClNameOut>. If there is no such class, <strVarClNameOut> value is not changed, and execution control is transferred to <labName>.

3. **firstAssoc** <clName>**.**<strVarRoleNameOut>**/** <strVarInvRoleNameOut>**.**<strVarClNameOut> **else** <labName>**;**
   Stores the role name, inverse role name, and target class name of the first association (ordering is implementation dependent) starting from <clName> in <strVarRoleNameOut>, <strVarInvRoleNameOut> and <strVarClNameOut>, respectively. If there are no associations starting from <clName>, then <strVarRoleNameOut>, <strVarInvRoleNameOut> and <strVarClNameOut> values are not changed and execution control is transferred to <labName>. Typically, this command is used in combination with the **nextAssoc** command to iterate through all associations starting from the given class (or from ancestors of this class).
   3.1. **firstAssoc (**<strVarClNameIn>**).**<strVarRoleNameOut>**/** <strVarInvRoleNameOut>**.**<strVarClNameOut> **else** <labName>**;**

4. **firstAssocDirect** <clName>**.**<strVarRoleNameOut>**/** <strVarInvRoleNameOut>**.**<strVarClNameOut> **else** <labName>**;**
   This command is similar to the previous one, the difference is that it takes into account only those associations which are defined exactly for this class and ignores associations which are defined in ancestor classes.
   4.1. **firstAssocDirect (**<strVarClNameIn>**).**<strVarRoleNameOut>**/** <strVarInvRoleNameOut>**.**<strVarClNameOut> **else** <labName>**;**

5. **nextAssoc** <clName>**.**<strVarRoleNameOut>**/** <strVarInvRoleNameOut>**.**<strVarClNameOut> **else** <labName>**;**
   Stores the role name, inverse role name, and target class name of the next association starting from <clName>, which has not yet been visited (iterated), in

<strVarRoleNameOut>, <strVarInvRoleNameOut> and <strVarClNameOut>, respectively. If there are no such associations, <strVarRoleNameOut>, <strVarInvRoleNameOut>, <strVarClNameOut> values are not changed, and execution control is transferred to <labName>.

5.1. **nextAssoc (**<strVarClNameIn>**).**<strVarRoleNameOut>**/** <strVarInvRoleNameOut>**.**<strVarClNameOut> **else** <labName>**;**

6. **nextAssocDirect** <clName>**.**<strVarRoleNameOut>**/** <strVarInvRoleNameOut>**.**<strVarClNameOut> **else** <labName>**;**
Similar to the previous one, but associations from ancestors are ignored.
6.1. **nextAssocDirect (**<strVarClNameIn>**).**<strVarRoleNameOut>**/** <strVarInvRoleNameOut>**.**<strVarClNameOut> **else** <labName>**;**

7. **firstAttr** <clName>**.**<strVarAttrNameOut>**.**<strVarAttrTypeNameOut > **else** <labName>**;**
Stores the name and type name of the first attribute (ordering is implementation dependent) of <clName> in <strVarAttrNameOut>, <strVarAttrTypeNameOut >, respectively. If <clName> has no attributes, then <strVarAttrNameOut>, <strVarAttrTypeNameOut > values are not changed and execution control is transferred to <labName>. Typically, this command is used in combination with the **nextAttr** command to iterate through all attributes of the given class (including ancestor attributes).
7.1. **firstAttr** **(**<strVarClNameIn>**).**<strVarAttrNameOut>.<strVarAttrTypeNameOut> **else** <labName>**;**

8. **firstAttrDirect** <clName>**.**<strVarAttrNameOut>**.**<strVarAttrTypeNameOut > **else** <labName>**;**
This command is similar to the previous one, the difference is that it takes into account only those attributes which are defined exactly in this class and ignores attributes which are defined in ancestor classes.
8.1. **firstAttrDirect** **(**<strVarClNameIn>**).**<strVarAttrNameOut>**.**<strVarAttrTypeNameOut> **else** <labName>**;**

9. **nextAttr** <clName>**.**<strVarAttrNameOut>**.**<strVarAttrTypeNameOut > **else** <labName>**;**
Stores the name and type name of the next attribute of <clName>, which has not yet been visited (iterated), in <strVarClNameOut> and <strVarAttrTypeNameOut>, respectively. If there are no such attributes, <strVarClNameOut> and <strVarAttrTypeNameOut > values are not changed and execution control is transferred to <labName>.
9.1. **nextAttr** **(**<strVarClNameIn>**).**<strVarAttrNameOut>**.**<strVarAttrTypeNameOut> **else** <labName>**;**

10. **nextAttrDirect** <clName>**.**<strVarAttrNameOut>**.**<strVarAttrTypeNameOut> **else** <labName>**;**
    Similar to the previous one, the difference is that it takes into account only those attributes which are defined exactly in this class and ignores attributes which are defined in ancestor classes.
    10.1. **nextAttrDirect** **(**<strVarClNameIn>**)**.<strVarAttrNameOut>**.**<strVarAttrTypeNameOut> **else** <labName>**;**

11. **firstSubClass** <superClName>**.**<strVarSubClNameOut> **else** <labName>**;**
    Stores the name of the first subclass (ordering is implementation dependent) in <strVarSubClNameOut>. If there are no subclasses, then <strVarSubClNameOut> value is not changed and execution control is transferred to <labName>. Typically, this command is used in combination with the **nextSubClass** command to iterate through all subclasses.
    11.1. **firstSubClass** **(**<strVarSuperClNameIn>**).**<strVarSubClNameOut> **else** <labName>**;**

12. **nextSubClass** <superClName>**.**<strVarSubClNameOut> **else** <labName>**;**
    Stores the name of the next subclass of <superClName>, which has not yet been visited (iterated), in <strVarSubClNameOut>. If there are no such classes, <strVarSubClNameOut> value is not changed, and execution control is transferred to <labName>.
    12.1. **nextSubClass** **(**<strVarSuperClNameIn>**).**<strVarSubClNameOut> **else** <labName>**;**

13. **firstEnum** <strVarEnumNameOut> **else** <labName>;
    Stores the name of the first enumeration (ordering is implementation dependent) in <strVarEnumNameOut>. If there are no enumerations, then <strVarEnumNameOut> value is not changed and execution control is transferred to <labName>. Typically, this command is used in combination with the **nextEnum** command to iterate through all enumeration names.

14. **nextEnum** <strVarEnumNameOut> **else** <labName>;
    Stores the name of the next enumeration which has not yet been visited (iterated) in <strVarEnumNameOut>. If there are no such enumerations, <strVarEnumNameOut> value is not changed and execution control is transferred to <labName>.

15. **firstEnumElem** <enumName>**.**<strVarEnumElemNameOut> **else** <labName>;
    Stores the name of the first <enumName> enumeration literal (ordering is implementation dependent) in the <strVarEnumElemNameOut>. If there are no enumeration literals in <enumName>, then <strVarEnumElemNameOut> value is not changed and execution control is transferred to <labName>. Typically, this command is used in combination with the **nextEnumElem** command to iterate through all given enumeration literals.

15.1. **firstEnumElem (**<strVarEnumNameIn>**).**<strVarEnumElemNameOut>
else <labName>;

16. **nextEnumElem** <enumName>**.**<strVarEnumElemNameOut> **else** <labName>;
Stores the name of the next <enumName> enumeration literal, which has not yet been visited (iterated) in <strVarEnumElemNameOut>. If there are no such enumeration literals, <strVarEnumNameOut> value is not changed and execution control is transferred to <labName>.
16.1. **nextEnumElem (**<strVarEnumNameIn>**).(** <strVarEnumElemNameOut>**)**
else <labName>;

17. **existsClass** <className> **else** <labName>**;**
If a class with a specified name exists, execution control is transferred to the next command, otherwise execution control is passed to <labName>.
17.1. **existsClass (**<strVarClassNameIn>**) else** <labName>**;**

18. **existsEnum** <enumName> **else** <label>**;**
If an enumeration with a specified name exists, execution control is transferred to the next command, otherwise execution control is passed to <labName>.
18.1. **existsEnum (**<strVarEnumNameIn>**) else** <labName>**;**

19. **existsEnumElem** <enumName>**.**<enumElemName> **else** <label>**;**
If an enumeration with a specified name has an enumeration literal with a specified name, execution control is transferred to the next command, otherwise execution control is passed to <labName>.
19.1. **existsEnumElem** (<strVarEnumNameIn>**).(**<strVarEnumElemNameIn>**)**
else <labName>**;**

20. **existsAttr** <clName>**.**<attrName>**.**<typeName> **else** <labName>**;**
If an attribute with specified properties exists, execution control is transferred to the next command, otherwise execution control is transferred to <labName>.
20.1. **existsAttr**
(<strVarClNameIn>**).(**<strVarAttrNameIn>**).(**<strVarAttrTypeNameIn>**)**
else <labName>**;**

21. **existsAssoc** <clName>**.**<roleName>**.**<clName> **else** <label>**;**
If an association with specified properties exists, execution control is transferred to the next command, otherwise execution control is transferred to <labName>.
21.1. **existsAssoc**
(<strVarClNameIn>**).(**<strVarRoleNameIn>**).(**<strVarClNameIn>**) else**
<labName>**;**

22. **existsCompos** <clName>**.**<roleName>**.**<clName> **else** <label>**;**
If a composition with specified properties exists, execution control is transferred to the next command, otherwise execution control is transferred to <labName>.

22.1. **existsCompos**
(<strVarClNameIn>**).(**<strVarRoleNameIn>**).(**<strVarClNameIn>**)**
else <labName>**;**

23. **existsRel** <subClName>**.subClassOf.**<superClName> **else** <label>**;**
If there is a generalization relationship between specified classes,execution control is transferred to the next command, otherwise execution control is transferred to <labName>.
23.1. **existsRel (**<strVarSubClNameIn>**).subClassOf. (**<strVarSuperClNameIn>**)**
else <labName>**;**

4 Implementation of L0+

4.1 Selection of the Runtime Environment

The implementation of a transformation language starts from the selection of a runtime environment. The selection of the run-time environment is not limited to the selection of a target language, because we have to provide a way of storing and accessing persistent data (metamodel and its instances) while implementing a model transformation language.

Quite natural choices in this situation are in-memory metamodel-based data stores (repositories) [17, 18, 19].

For the implementation of L0 and L0+, the in-memory data store developed at the IMCS was selected [19]. This repository proved to be reasonably efficient. For instance, in [19] it is shown, that this data store is at least as efficient in typical instance selection tasks as one of the most popular open-source RDF data stores Sesame [20].

The API of the chosen repository is implemented as a library of C++ functions. This library provides the following possibilities:

- a set of functions for creation and deletion of metamodel elements, as well as iteration through them;
- model processing functions that can be divided into two subcategories:
  - functions for creation and deletion of instances (objects and links) and functions for getting/setting the value of an attribute, for example:
    ```
    long CreateObject(long ObjectTypeId);
    int DeleteObjectHard(long ObjectId);
    int CreateLink(long LinkTypeId, long ObjectId1, long ObjectId2);
    int DeleteLink(long LinkTypeId, long ObjectId1, long ObjectId2);
    ```
  - efficient searching functions (these functions are based on sophisticated indexing mechanisms):
    ```
    int GetObjectNum(long ObjectTypeId);
    long GetObjectIdByIndex(long ObjectTypeId, int Index);
    int GetLinkedObjectNum(long ObjectId, long LinkTypeId);
    ```

long GetLinkedObjectIdByIndex(long ObjectId, long LinkTypeId, int Index);

The main advantage of the use of this repository is that there are close counterparts for a substantial part of L0 and L0+ commands in the repository API. It means that it will be quite easy to implement these commands, and there will be no substantial difference (at least in the aspect of efficiency) between a hand-written and a compiler-generated code.

Taking into account the fact that the selected repository provides a C++ API, C++ was chosen as a target language for L0 and L0+ compilation. Since effective C++ compilers are known, it is possible to generate a C++ code without thinking of its extensive optimization, because all optimizations of the C++ code will be done by the C++ compiler. Thus we can omit final phases of traditional compilers i.e. intermediate code generation and optimization.

4.2 Compilation Schema

L0 and L0+ compilation process consists of four phases:

- Preprocessing phase, when compiler directives (for example, "useMM") are analyzed;
- Lexical analysis phase, when transformation program is divided into lexemes;
- Syntactical analysis phase, when the list of lexemes of the program is divided into groups of lexemes corresponding to definite commands;
- Code generation phase, when C++ code is generated from a group of lexemes, corresponding to a definite command.

L0 and L0+ languages do not contain recursive constructs. Moreover, the start and the end of every command are easily identifiable. Because of these two facts, syntactical and lexical analysis is quite simple and will not be described further.

C++ code generation seems to be more interesting. Let us start with some general principles. Firstly we have to understand how to compile general constructs: subprograms (with corresponding parameters passing mechanisms), control flow commands, elementary variables and pointers to class instances. It is not difficult to spot similarities between C++ functions and L0 subprograms, C++ elementary variables and L0 elementary variables, C++ control flow possibilities, and L0 control flow possibilities.

However, the situation with L0 pointers (references to class instances) is somewhat more special, because in C++ there is no direct way of simulating them. To represent L0 pointers in C++ program, we define a C++ class L0_Var_2 with operations corresponding to L0 commands. This class has the following operations:

- bool moveNext();
- bool isNull();
- bool setFirstToRoot(const string & className),;
- bool setFirstFrom(const L0_Var_2 & rhs , const string & assocName);
- bool setStringAttributeValue(const string & attrName, const string & newValue);

etc.

The implementation of the class L0_Var_2 is based on model processing functions from the Repository API.

Now, when it is clear how individual constructs are compiled, an overall compilation schema can be given:

- Every L0 subprogram is compiled to a corresponding C++ function;
- Every elementary L0 variable is compiled to a corresponding C++ variable;
- Every L0 pointer is compiled to an object of the C++ class L0_Var_2;
- Every L0 command call on a given L0 pointer is compiled to a corresponding C++ method call on a C++ object.

The situation with L0+ commands is similar. Every L0+ command is mapped to a C++ function that relies on the metamodel processing functions from the Respository API.

4.3 Elementary Tracing Facilities

If a program flow is specified with conditional and unconditional control flow transfer operators (i.e. structured programming constructs are not used), then it becomes rather difficult to trace program execution flow (as a consequence, it is difficult to debug these programs). L0 does not provide structured programming constructs. That is why typical errors in L0 programs are related to incorrectly specified control flows. To simplify L0 transformation debugging, L0 compiler can generate code in debugging mode. When a program generated in this mode runs, it logs execution path and other significant information. For example, the following program finds the least of three numbers.

```
transformation traceDemo;

DEBUG ON;
main procedure p();
  var i1 : Integer;
  var i2 : Integer;
  var i3 : Integer;
  var min: Integer;
begin;
  setVar i1 = 10;
  setVar i2 = 8;
  setVar i3 = 6;

  var i1 < i2 else i2LessThani1;
    var i1 < i3 else i3LessThani1;
     setVar min = i1;
     return;

    label i3LessThani1;

     setVar min = i3;
```

```
        return;

      label i2LessThani1;

      var i2 < i3 else i3LessThani2;
        setVar min = i2;
        return;

        label i3LessThani2;

        setVar min = i3;
        return;
    end;

    endTransformation;
```

When running this program in debug mode, it will produce the following output:

```
     12 : procedure p

     18 : setVar i1 = 10
       i1 = 10
     19 : setVar i2 = 8
       i2 = 8
     20 : setVar i3 = 6
       i3 = 6
     22 : var i1 < i2 else i2LessThani1
     32 : label i2LessThani1
     34 : var i2 < i3 else i3LessThani2
     38 : label i3LessThani2
     40 : setVar min = i3
       min = 6
     41 : return
   Return from p
```

To implement this functionality, generated C++ code is appended with some code fragments logging activities of the program.

For example, when L0 compiler receives a command "**setVar** `i1 = 10;`", it emits the following C++ code: "`elemVar___p_i1_ = 10;`". But if L0 compiler is generating debug code, it will emit substantially different code for the same L0 command:

```
"
Logger::inst().wrtLineNum( 17 );
  Logger::inst().wrtLine( "setVar i1 = 10" );
elemVar___p_i1_ =  10 ;
Logger::inst().wrtPref( );  Logger::inst().wrt( "i1 =
");
```

```
  Logger::inst().wrtInt(elemVar___p_i1_ );
  Logger::inst().newLine();
  ”.
```

5 Conclusions

This paper was devoted to the problems of effective implementation of model transformation languages. It was stated that direct implementation of high level transformation languages is a difficult and costly process that does not guarantee effectiveness of the obtained implementation. It is believed that a more optimal way to implement high level model transformation language is to use bootstrapping. Bootstrapping in its turn is not possible without an effective base language.

One of the main results of this paper is the definition of a new low level model transformation language L0+, that can be used as a base language in bootstrapping process. L0 is called a base language, because:

- it contains minimal, but sufficient model transformation constructs;
- an effective implementation for this language does exist.

One more reason justifying L0+ usage in bootstrapping process is the increased portability of a high level language being compiled to L0+. L0+ naturally becomes a kind of a repository abstraction layer, meaning that if we want to port an implementation of a high level language to another target environment (that uses L0+ as a target language), it is sufficient to port only the implementation of L0+.

The second notable result of this paper is principles of an effective implementation of this language. According to these principles, an effective implementation of L0+ was obtained.

L0+ was designed to be of as low level as possible to simplify its implementation, and it does not contain some the of constructs (mainly, patterns) found in more advanced languages. Nevertheless, this language is also used for a direct development of model transformations.

References

1. J. Barzdins, A. Kalnins, E. Rencis, S. Rikacovs, Model Transformation Languages and their Implementation by Bootstrapping Method, Pillars of Computer Science: Essays Dedicated to Boris (Boaz) Trakhtenbrot on the Occasion of His 85th Birthday, Arnon Avron, Nachum Dershowitz, and Alexander Rabinovich, editors, Lecture Notes in Computer Science, vol. 4800, Springer-Verlag, Berlin, 2008.
2. A.Kleppe, J. Warmer, W. Bast, MDA Explained: The Model Driven Architecture -- Practice and Promise, Addison Wesley, 2003.
3. Request for Proposal : MOF 2.0 Query / Views / Transformations RFP, URL: http://www.omg.org/docs/ad/02-04-10.pdf
4. OMG, MOF 2.0 Query/View/Transformation Specification. URL: http://www.omg.org/docs/ptc/07-07-07.pdf

5. Agrawal A., Karsai G, Shi F.: Graph Transformations on Domain-Specific Models. Technical report, Institute for Software Integrated Systems, Vanderbilt University, ISIS-03-403, November 2003.
6. A.Kalnins, J. Barzdins, E.Celms. Basics of Model Transformation Language MOLA. - ECOOP 2004 (Workshop on Model Transformation and execution in the context of MDA), Oslo, Norway, June 14-18, 2004.
7. ATL. URL: http://www.sciences.univ-nantes.fr/lina/atl/
8. MTF. URL: http://www.alphaworks.ibm.com/tech/mtf
9. Tefkat.URL: http://www.dstc.edu.au/Research/Projects/Pegamento/tefkat/
10. E.Rencis. Model Transformation Languages L1, L2, L3 and their Implementation, Scientific Papers. University of Latvia, "Computer Science and Information Technologies", 2008.
11. 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.
12. VIATRA2 URL: http://dev.eclipse.org/viewcvs/indextech.cgi/gmt-home/subprojects/VIATRA2/index.html
13. Meta Object Facility (MOF) Specification Version 2.0 URL : http://www.omg.org/docs/formal/06-01-01.pdf
14. Meta Object Facility (MOF) Specification Version 1.4.1 URL : http://www.omg.org/docs/formal/05-05-05.pdf
15. Deborah J. Armstrong, The quarks of object-oriented development, Communications of the ACM, 2006.
16. A.Sostaks., A.Kalnins. The implementation of MOLA to L3 compiler, Scientific Papers University of Latvia, "Computer Science and Information Technologies", 2008.
17. Budinsky, F., Steinberg, D., Merks, E., Ellersick, R., Grose, T.: Eclipse Modeling Framework. Addison Wesley, 2003.
18. Metadata Repository (MDR). URL: http://mdr.netbeans.org/
19. 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 , 2006), pp. 203-218.
20. Sesame. URL: http://www.openrdf.org/