Model Transformation Languages and Their Implementation by Bootstrapping Method

Janis Barzdins, Audris Kalnins, Edgars Rencis, and Sergejs Rikacovs

University of Latvia, IMCS, 29 Raina boulevard, Riga, Latvia
janis.barzdins@mii.lu.lv, audris.kalnins@mii.lu.lv, edgars.rencis@lumii.lv, sergejs.rikacovs@lumii.lv

Dear Boris, You are the father of Computer Science in Latvia.
Thank you for this.

Abstract. In this paper a sequence of model transformation languages L0, L1, L2 is defined. The first language L0 is very simple, and for this language it is easy to build an efficient compiler to C++. The next language L1 is an extension of L0, and it contains powerful pattern definition facilities. The last language L2 is of sufficiently high level and can be used for implementation of traditional pattern-based high level model transformation languages, as well as for the development of model transformations directly. For languages L1 and L2 efficient compilers have been built using the bootstrapping method: L1 to L0 in L0, and L2 to L1 in L1. The results confirm the efficiency of model transformation approach for domain specific compiler building.

1 Introduction

A well known fact is that the heart of the most advanced software engineering technology MDA [1] is model transformation languages. In recent years the main emphasis has been on the development of industrial transformation languages [2,3,4,5,6,7]. For most of the transformation languages there is an implementation. However, there has been no thorough research on transformation language implementation, especially on the efficiency aspects. On the other hand, there have been only a few attempts to use transformation languages for defining their compilers (to use bootstrapping) [5,7,8]. It is a little bit strange taking into the account that the main idea of MDA is to use transformation languages for transforming formal design models also a sort of language. Most of the MDA success stories are related to Domain Specific Languages there the corresponding transformations are in fact compilers. One of the goals of this paper is demonstrate the usability and efficiency of transformation languages namely for defining compilers for transformation languages.

The other goal is to propose a very simple, but at the same time sufficiently high level, transformation language L2 which can be used in practice for direct development of model transformations.

A. Avron et al. (Eds.): Trakhtenbrot/Festschrift, LNCS 4800, pp. 130–145, 2008.

The main results of this paper are the following:

- a sequence of transformation languages L0, L1, L2 is offered and each language is obtained from the previous one by adding some features. The final language L2 is of pretty high level (it contains a kind of patterns, loops, etc.)
- the first of languages L0 is very simple. It contains only the basic transformation facilities and its complete description can be given in less than two pages (see Section 2). For this language it is easy to build an efficient compiler to C++
- a compiler from L_{i+1} to L_i (i = 0,1) can be easily specified in L_i (this can be done also in L0). This acknowledges the efficiency of using transformation languages for building their compilers as long as an appropriate for bootstrapping language sequence has been found
- the last language in the sequence L2 is of sufficiently high level for traditional pattern-based high level model transformation languages (such as MOLA [6]) to be compiled to it in a natural way, with the compiler also being easily definable in L2.

The language L0 and henceforth also L_i include also the basic facilities for defining metamodels, in order to make these languages self-contained.

2 The Base Language L0

The purpose of this section is to give a brief overview of the transformation language **L0**. This language is a rather low level procedural textual language, with control structures mostly taken from assembler-like languages (and syntax influenced by C++). The basic setting of L0 is as for any transformation language - we process a model, which is an instance of metamodel (MOF style). But the language constructs which are specific to model transformations have been chosen to be as simple as possible.

Basically these constructs give the programmer the ability to:

- iterate through instances (both links and objects),
- create/delete objects and links,
- read/write (change) object attribute values.

An elementary unit of L0 transformation program is a **command** (an imperative statement). L0 transformation program itself is a **transformation**, which contains several parts:

- global variable definition part
- native subprogram (function or procedure) declaration part (used C++ library function headers)
- L0 subprogram definition part. It is expected that exactly one subprogram in this part is labeled with the reserved word **main**. The subprogram labeled with **main** defines the entry point of the transformation. An L0 subprogram definition also consists of several parts:

- Subprogram header
- Local variable definitions
- Keyword **begin**;
- Subprogram body definition
- Keyword **end**;

L0 contains the following kinds of commands:

1. **transformation** <transformationName>; This command starts a transformation definition.
2. **endTransformation**; The command ends a transformation definition.
3. **pointer** <pointerName> : <className>; Defines a pointer to objects of class <className>.
4. **var** <varName> : <ElementaryTypeName>; <ElementaryTypeName> is one of Boolean, Integer, Real, String. Defines a variable of elementary type.
5. **procedure** <procName>(<paramList>); Subprogram header, the (formal) parameter list can be empty. 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 or a class from the metamodel), 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>(<paramList>): <returnType>; Return type name can be an elementary type name or class name.
7. **begin**; Starts subprogram body definition.
8. **end**; Ends subprogram body definition.
9. **return**; Returns execution control to caller.
10. **return** <identifier>; Return the value of <identifier> to the caller, the type must coincide with the function return type. <identifier> is an elementary variable name or pointer name.
11. **call** <subProgName>(<actPrmList>); The actual parameter list, which can be empty, consists of identifiers separated by ",". An identifier can be a variable name, a pointer name, or a subprogram parameter name.
12. **setVarF** <variable>=<funcName>(<actPrmList>); This command can be used to obtain the value of the function result. The result is of an elementary type and is assigned to a variable. The variable type must coincide with the function return type.
13. **first** <pointer> : <className> **else** <label>; Positions <pointer> to an arbitrary (the first one in an implementation dependent ordering) object of <className>. Typically, this command in combination with the **next** command is used to traverse all objects of the given class (including subclass objects). If <className> does not have objects, <pointer> becomes **null**, and execution control is transferred to the <label>. The <className> in this command must be the same as (or a subclass of) the class used in pointer definition; if it is a subclass, then the pointer value set is narrowed (for the subsequent executions of **next**).

14. **first** <pointer1> : <className> **from** <pointer2> **by** <roleName> **else** <label>; Similar to the previous command, the difference is that it positions <pointer1> to an arbitrary class object, which is reachable from <pointer2> by the link <roleName>. Similarly, this command in combination with the **next** command is used to traverse all objects linked to an object by the given link type.
15. **next** <pointer> **else** <label>; Gets the next object, which satisfies conditions, formulated during the execution of the corresponding **first** and which has not been visited (iterated) with this variable yet. If there is no such object, the <pointer> becomes **null**, and execution control is transferred to <label>.
16. **goto** <label>; Unconditionally transfers control to <label>, <label> should be located in the current subprogram.
17. **label** <labelName>; Defines a label with the given name.
18. **addObj** <pointer>:<className>; Creates a new object of the class <className>.
19. **addLink** <pointer1>.<roleName>.<pointer2>; Creates a new link (of type specified by <roleName>) between the objects pointed to by the <pointer1> and <pointer2> , respectively.
20. **deleteObj** <pointer>; Deletes the object, which is pointed to by <pointer>.
21. **deleteLink** <pointer1>.<roleName>.<pointer2>; Deletes link, whose type is specified by <roleName>, between objects pointed to by <pointer1> and <pointer2>, respectively.
22. **setPointer** <pointer1>=<pointer2>; Sets <pointer1> to the object, which is pointed to by <pointer2>; in place of <pointer2> the **null** constant can be used.
23. **setPointerF** <pointer>=<funcName>(<actPrmList>); Sets <pointer> to the object, which is returned by <funcName>.
24. **setVar** <variable> = <binExpr>; Sets <variable> to <binExpr> value. <binExpr> is a binary expression consisting of the following elements: elementary variables, subprogram parameters (of elementary types), literals, object attributes and standard operators (+, -, *, /, &&, ||, !).
25. **setAttr** <pointer>.<attrName>=<binExpr>; Sets the value of attribute <attrName> (of the object, pointed to by <pointer>) to the <binExpr> value.
26. **type** <pointer> == <className> **else** <label>; If the type of the pointed object is identical to the <className>, then control is transferred to the next command, else control is transferred to <label>. In place of the equality symbol == an inequality symbol != can be used. This command is used for determining the exact subclass of an object.
27. **var** <variable>==<binExpr> **else** <label>; If the condition is not true then control is transferred to <label>. In place of equality symbol other (<, <=, >, >=, !=) relational operators compatible with argument types can be used.
28. **attr** <pointer>.<attrName> == <binExpr> **else** <label>; If condition is not true then control is transferred to <label>. Other relational operators (<, <=, >, >=, !=) can be used too.

29. **link** <pointer1>.<roleName>.<pointer2> **else** <label>; Checks whether there is a link (with the type specified by <roleName>) between the objects pointed to by <pointer1> and <pointer2>, respectively.
30. **pointer** <pointer1>==<pointer2> **else** <label>; Checks whether the objects pointed to by <pointer1> and <pointer2>, respectively, are identical. Instead of <pointer2> **null** can be used, and the inequality symbol can be used too.

Actually L0 contains also commands for building the relevant metamodel; for details see `http://Lx.mii.lu.lv/`.

It is easy to see that the language L0 contains only the very basic facilities for defining transformations. At the same time, it obviously is **complete** in the sense of its functional capabilities. This is confirmed by the fact that high level transformation languages such as MOLA can be successfully compiled to it. We omit this result in the form of a theorem, but all informal justifications of this thesis are in place. Namely this is why we call L0 the basic transformation language. We start our bootstrap approach with this language.

We conclude this section with a very simple example of L0 - a transformation which builds a representation B of a directed graph (where edge connection points are also objects) from the simplest one A (where only nodes and edges are present). Figure 1 presents the metamodel for both representations.

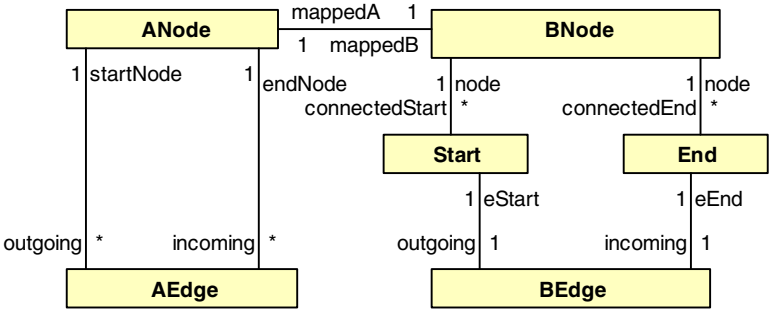


Fig. 1. Metamodel for the example

The L0 program performing the transformation:

```
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 aEdgesProc;
label loopAEdge;
addObj bEd : BEdge;
addObj edgeStart : Start;
addObj edgeEnd : End;
addLink bEd.eStart.edgeStart;
addLink bEd.eEnd.edgeEnd;
//quit if not found;
first aEdgeStNode:ANode from aEd by startNode else aEdgesProc;
first mapBNode:BNode from aEdgeStNode by mappedB else
aEdgesProc;
addLink edgeStart.node.mapBNode;
first aEdgeEnNode:ANode from aEd by endNode else aEdgesProc;
first mapBNode:BNode from aEdgeEnNode by mappedB else
aEdgesProc;
addLink edgeEnd . node. mapBNode;
next aEd else aEdgesProc;
goto loopAEdge;
label aEdgesProc;
end;
endTransformation;
```

3 Implementation of L0

The language L0 can be implemented in several ways. The first problem is how to store and access the persistent data the metamodel and its instances. Obviously, a kind of data store is required for this. A traditional relational database could be used, but they typically have no adequate low level API. Another alternative could be an in-memory data store, such as RDF-oriented Sesame [9] or an MOF-oriented one (EMF [10], MDR [11]). However, for this approach we have selected our own metamodel-based in-memory repository [12], which has an appropriate low level API. Being developed over many years for other goals - generic metamodel based tool building [13], this repository occurred to be efficient enough for implementing L0.

The API of this repository is implemented as a C++ function library. This library offers: a) a system of low-level data retrieval functions that is complete for low-level data query programming; b) a selected set of more complicated widely usable data searching functions. By means of a sophisticated indexing mechanism, these more complicated functions are also efficiently implemented.

The API of this repository includes two groups of functions:

1. Meta-model management functions for creating, modifying, and deleting classes, attributes and associations, querying about their properties, class inheritance, etc. However, meta-model management functions are used relatively seldom, the most heavily used functions belong to the next group.
2. Instance management. This group of functions, in its turn, also can be subdivided in two groups:
   (a) functions for creating instances, assigning attribute values, creating links between instances, modifying and deleting instances and links, querying about instance attributes and links. For example:
   ```
   long CreateObject(long ObjTypeId); // returns objId
   int DeleteObjectHard(long ObjId);
   int CreateLink(long LinkTypeId, long ObjId1, long ObjId2);
   int DeleteLink(long LinkTypeId, long ObjId1, long ObjId2);
   ```
   (b) efficient searching functions (internally these functions use sophisticated indexing mechanisms):
   ```
   int GetObjectNum(long ObjTypeId);
   long GetObjectIdByIndex(long ObjTypeId, int Index);
   int GetLinkedObjectNum(long ObjId, long LinkTypeId);
   long GetLinkedObjectIdByIndex(long ObjId, long LinkTypeId,
                                 int Index);
   ```

If a repository with such API is available, then building an L0 compiler (to C++) is quite a straightforward job. Such a compiler has been built by one of the authors of this paper (S. Rikacovs) in two months (not including L0 debugging facilities). The main advantage of using this repository is that the instance management functions in L0 (**first** and **next**, including the **by** link options) have close counterparts in the repository API.

The implementation efficiency is also sufficiently high. First, some experiments show that efficiency loss with respect to the same transformation manually coded in C++ is no more than 1.5 times. Another aspect is efficiency of the selected repository for typical transformations, where another group of experiments [12] show that the selected repository is at least as efficient as Sesame [9] data store for typical instance retrieval operations.

4 The Language L1

The crucial component of any advanced transformation language is some sort of pattern definition facilities. This way, the transformation language L1 is obtained from L0 by adding pattern definition facilities of a specific new form. In selecting

the pattern definition method we were guided by two conflicting requirements. On the one hand, the pattern concept must be practically usable. On the other hand, it must have a simple and efficient implementation by compiler (traditional patterns, e.g. in [4,5,6] not always have this property). One of the main results of this paper is the proposed pattern definition facility, which satisfies both requirements. The main component of pattern specification is a facility for defining expressions over environments of model objects. Our approach is based on a new kind of expressions - **begin-end expressions**, which are defined as **command blocks** of the kind:

begin <commandSequence> **end**.

Namely, if we execute the block on the given object environment, and reach the **end** command, then the expression value is defined to be true, otherwise it is false.

For example, the expression (block):

```
begin
  attr p.age==23;
  attr p.occupation=="Student";
end
```

has the value true if and only if the pointer p (of type Person) points to an instance, whose attribute age has the value 23 and the attribute occupation has the value "Student".

Some more comments on begin-end expressions must be given - what is meant by not reaching the **end**. If during the block execution we reach an undefined **else** branch of a command (there is no **else** keyword or it is not followed by a label, this is permitted for all **else**-containing commands of L0) then the expression is defined to have the value false. A similar way is to use a **goto** command without label (but it is forbidden to use a label not defined in the block).

Now, when the begin-end expressions are described, it is possible to define the language L1 precisely.

The language **L1** differs from L0 in commands **first** and **next** extended by **suchthat** part containing a begin-end expression:

```
first <pointer> : <className>  suchthat <BeginEndExpression>
else <label>;

next <pointer>  suchthat <BeginEndExpression>
else <label>;
```

Now we will explain in some details the role of begin-end expressions for pattern definition and compare them to other facilities for pattern definition. Let us assume that we have the class diagram ("metamodel") in Figure 2. Such a class diagram can be treated also as a signature for formula definition in many-sorted first order logic (MS FOL) - an association corresponds to a binary predicate and an attribute to a function. We want to define certain patterns

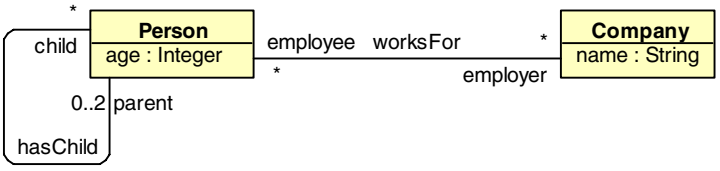


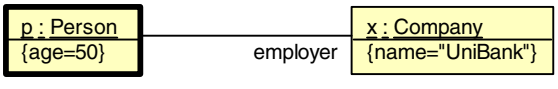
Fig. 2. Metamodel for pattern examples

for p:Person, i.e., constraints which should be satisfied by appropriate Person instances.

To get a deeper insight into the situation, we will define these patterns in several languages, starting from the natural language. Let us consider an example:

p is a Person, whose age is 50 and who works for (i.e., its employer is) the Company "UniBank".

The same pattern can be specified graphically in the MOLA transformation language:



(in other transformation languages this can be done in a similar way).

In MS FOL the same pattern can be represented by the following formula F(p) (the free variable p has the type Person):

$$p.age = 50 \quad \& \\ \exists x : Company(x.name = \text{“}UniBank\text{”} \quad \& \quad worksFor(p, x)) \ . \qquad (1)$$

The same pattern can be specified also by a begin-end expression, where p is a pointer variable with the type Person:

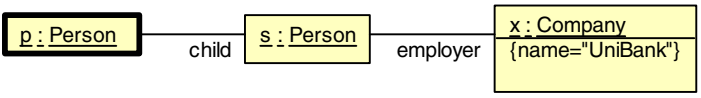
```
begin
  attr p.age==50;
  first x:Company suchthat
    begin
    attr x.name=="UniBank";
    link p.employer.x;
  end;
end;
```

Let us note that in this context "**first** x: **suchthat** " is equivalent to "**exists** x: **suchthat** ".

Now let us consider a more complicated example:

p is a Person, who has a child working for the Company "UniBank".

This corresponds to the following MOLA pattern:



The corresponding MS FOL formula F(p) is:

$$\exists s : Person(hasChild(p, s) \quad \& \\ \exists x : Company(x.name = \text{“}UniBank\text{”} \quad \& \quad worksFor(s, x))) \ . \qquad (2)$$

The corresponding begin-end expression is:

```
begin
  first s:Person suchthat
  begin
    link p.child.s;
    first x:Company suchthat
    begin
      attr x.name=="UniBank";
      link s.employer.x;
    end;
  end;
end;
```

Now let us consider a significantly more complicated example:

p is a Person, whose all adult (not younger than 18) children work for the Company “UniBank”.

It is difficult to specify such a pattern in a graphical pattern definition language. At the same time it can be specified quite easily as a MS FOL formula and also as a begin-end expression.

The corresponding MS FOL formula F(p) is:

$$\forall s : Person(s.age >= 18 \quad \& \quad hasChild(p, s) \quad \supset \\ \exists x : Company(x.name = \text{“}UniBank\text{”}) \quad \& \quad worksFor(s, x)) \ . \qquad (3)$$

The corresponding begin-end expression is:

```
begin
  first s:Person suchthat
  begin
    link p.child.s;
    attr s.age>=18;
    first x:Company suchthat
    begin
      attr x.name=="UniBank";
      link s.employer.x;
    end
    else L1;
```

```
    goto;
    label L1;
  end else L0;
  goto;
  label L0;
end;
```

It is easy to check that we can reach the final **end** iff p points to a Person, which satisfies the abovementioned constraint. This begin-end expression actually corresponds to the following MS FOL formula (which is equivalent to the formula above):

$$\neg\exists s : Person(hasChild(p, s) \quad \& \quad s.age >= 18 \quad \& \\ \neg\exists x : Company(x.name = \text{“}UniBank\text{”} \quad \& \quad worksFor(s, x))) \ . \qquad (4)$$

MS FOL apparently is one of the most universal languages for defining patterns. However, existing transformation languages avoid the use of MS FOL formulas for pattern definition. The reason is that for such a universal pattern specification no satisfactory (non-exponential) pattern matching algorithm is known (most probably, such an algorithm does not exist). Therefore existing transformation languages limit in a natural way their pattern definition mechanisms in accordance with their graphical capabilities.

A natural question arises about the relation between our begin-end expressions and MS FOL formulas in the context of pattern definition. The answer is that for pattern definition the power of begin-end expressions is not less than that of MS FOL formulas. We will not go into details of this problem. Let us note only that the proof of this assertion (after the corresponding concepts are made precise enough) is not complicated - it is sufficient to trace the inductive definition of MS FOL formulas.

However, in order to give a deeper insight into begin-end expressions, we explain a small fragment of this proof. Let $F(p)$ and $G(p)$ be MS FOL formulas with p as the free variable. We assume that we have already built begin-end expressions $E_{F(p)}$ and $E_{G(p)}$ which define the same patterns. Namely,

$E_{F(p)} \equiv$ **begin** <commandSequence for F> **end**

and

$E_{G(p)} \equiv$ **begin** <commandSequence for G> **end**.

Let us consider the formula $F(p)\&G(p)$. It is easy to see that the following begin-end expression defines an equivalent pattern:

begin <commandSequence for F> <commandSequence for G> **end**

Now let us consider the formula $\neg F(p)$. The corresponding begin-end expression can be obtained in the following way. Those else-branches inside $E_{F(p)}$ which

have no label are completed by a certain fixed label, let's say L. The same action is applied to goto's without label (such commands are permitted in L1). This action is not applied to begin-end expressions which are inside nested suchthat parts. Let us denote the transformed begin-end sequence by <commandSequence for $\neg F$>. The sought for begin-end expression has the following form:

begin <commandSequence for $\neg F$> **goto**; **label** L; **end**.

It is easy to see that we can reach the label L (which is the last one in this block and therefore reaching it means that the whole expression assumes the value true) iff the original expression for F had the value false.

The other inductive steps for MS FOL formula definition can be treated in a similar way.

In reality begin-end expressions are even more powerful than pure MS FOL, since begin-end expressions can contain also operations on elementary variables.

A question arises why our begin-end expressions are superior to MS FOL for specifying patterns. There are three essential reasons for this:

1. A begin-end expression specifies the command execution order during the pattern matching (i.e., the order in which the instances are traversed).
2. When a pattern is matched all its elements are assigned an identity which can be used further for referencing these elements (a similar approach is used in all graphical pattern languages).
3. Begin-end expressions can be easily compiled to L0 (the obtained L0 fragment directly implements the pattern matching for the expression).

5 The Final Language L2 and Its Usage

The language **L2** is obtained from L1 by extending it with a **foreach** command (loop) and the **if-then-else** command:

```
foreach <loopVariable> : <className> suchthat
<BeginEndExpression> do <L2commSequence> end;

if <BeginEndExpression> then do <L2commSequence> end else do
<L2commSequence> end;
```

The loop semantics is quite natural: the loop variable traverses all instances of the class, which satisfy the suchthat condition, for each such instance the do-end block is executed (explicit jumping out of the loop body is prohibited). The foreach command may be used also inside a suchthat block.

The metamodel of L languages is given in Figure 3 (dashed association corresponds to element of L1, bold classes/associations to L2).

The language L2 has at least two important usage areas. On the one hand, it can be used as a practical model transformation language. On the other hand, practical high level model transformation languages can be adequately compiled to it, and the compiler itself can be written in L2 (we consider this kind of usage the main one). Currently such a schema has been successfully applied for building an efficient implementation of MOLA [6], but the same approach could be applied also for implementing MOF QVT [2] and other transformation languages. The main issue for such compilations is how to map "completely declarative" traditional patterns to patterns with the specified search order in L languages. In some sense the basic idea for such a mapping is given in [14].

6 Implementation of L1 and L2

The languages L1 and L2 have been implemented according to the bootstrapping principles described in the introduction.

A compiler from L1 to L0 has been implemented in L0 (as a set of recursive procedures). It contains about 200 lines of L0 and has been written in one month (by E. Rencis). Though L1 includes a pattern definition mechanism even more powerful than that of MS FOL, implementation of L1 patterns is relatively simple since the search order of pattern elements is precisely specified in the language. Actually the command sequence defining a begin-end expression can quite easily be transformed into an equivalent sequence of L0 commands, using recursion for nested expressions.

To illustrate the idea, we will show briefly the schema how the L1 command

```
first <pointer> : <className>  suchthat <BeginEndExpression>
else <label>;
```

can be compiled to L0 commands. By means of **first**, **next** and **goto** commands a simple loop is organized which scans all instances of the given class. The "body" of this loop contains slightly modified commands form the begin-end expression commands with missing (or empty) else-branch are "redirected" to a new label in the else-case. Then reaching this new label would mean that this **suchthat** fails on the given instance and the next instance must be tried. If, on the contrary, the end of the loop body is reached, the given instance satisfies the whole **suchthat** and the job is done. If a command within the expression body is not an L0 command, but a true L1 command, the same procedure is applied recursively. This compilation schema is illustrated in Table 1.

The compiler from L2 to L1 is also relatively simple (about 560 lines of L0).

Both L1 and L2 compilers rely on the metamodel of L languages (Figure 3). Compilation of L_{i+1} to L_i actually converts into a transformation of models (i.e., L_{i+1} programs) corresponding to the given metamodel. As it was already mentioned, this transformation occurrs to be relatively simple.

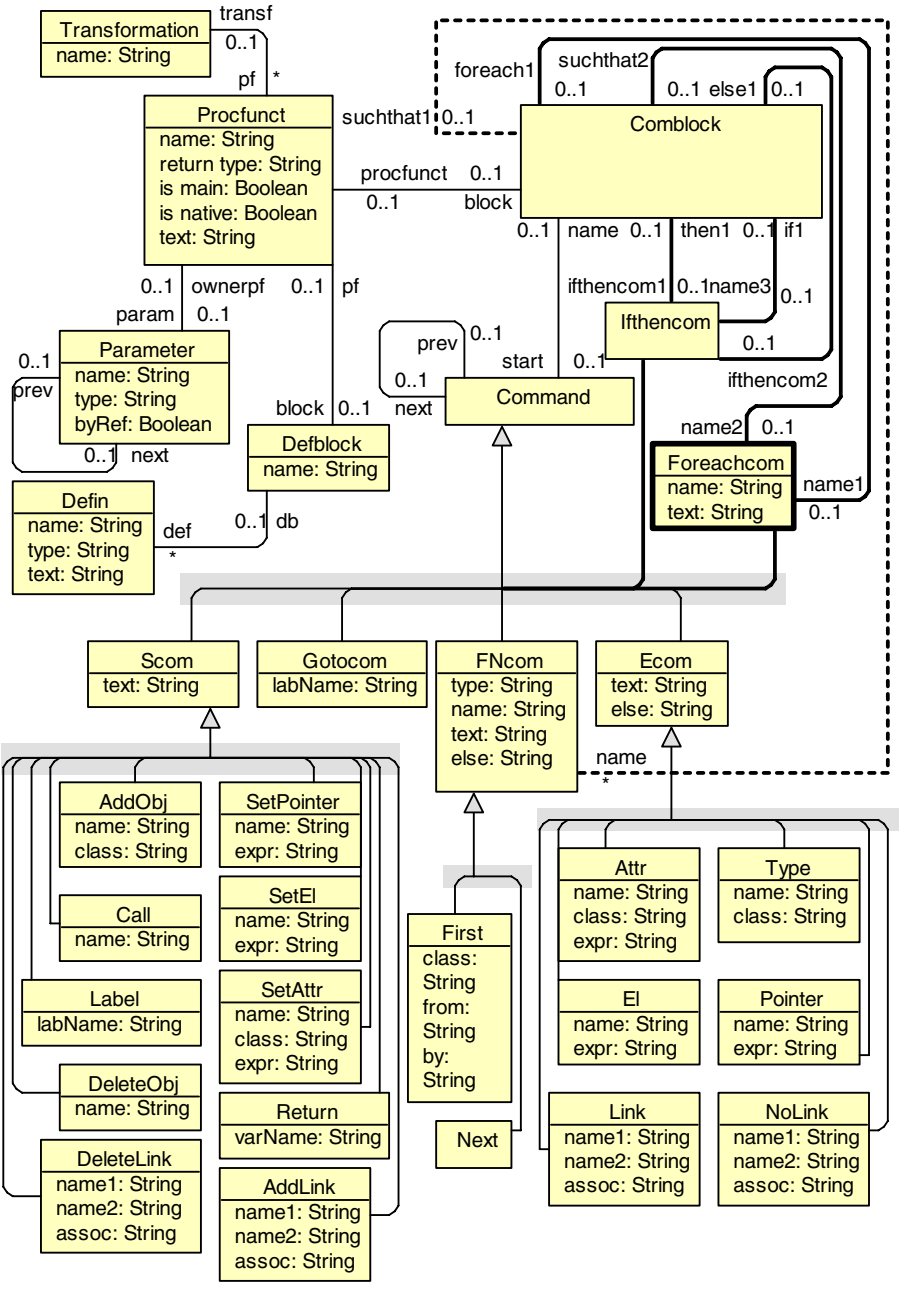


Fig. 3. Metamodel of L languages

Table 1. Compilation schema from L1 to L0

L1	**L0**
first <objName> : <className>	**first** <objName> : <className>
suchthat	**else** <labelName>;
begin	**label** _L_i;
command_1;	command_1 [else _L_i+1];
command_2;	command_2 [else _L_i+1];
...	...
command_n;	command_n [else _L_i+1];
end	**goto** _L_i+2;
else <labelName>;	**label** _L_i+1;
	next <objName> **else** <labelName>;
	goto _L_i;
	label _L_i+2;

7 Conclusions

It was many years ago, when the first author of this paper was a PhD student of B. A. Trakhtenbrot and studied the most general concept of automata (so called growing automata [15]), based on Kolmogorov-Uspenskii machines [16]. The Kolmogorov-Uspenskii machine, in contrast to Turing machine, can process arbitrary constructive objects ("colored graphs"), which can change their topology during the processing. At that time it was merely an instrument for studying theoretical capabilities of algorithms and automata.

Several decades have passed until similar ideas have reified into a powerful software engineering tool, now named Model Transformation Languages. Transformation languages can be regarded also as practical languages for programming Kolmogorov-Uspenskii machines.

References

1. MDA Guide Version 1.0.1. OMG, document omg/03-06-01 (2003)
2. MOF QVT Final Adopted Specification, OMG, document ptc/05-11-01 (2005)
3. Jouault, F., Kurtev, I.: Transforming Models with ATL. In: Bruel, J.-M. (ed.) MoDELS 2005. LNCS, vol. 3844, pp. 128–138. Springer, Heidelberg (2006)
4. Agrawal, A., Karsai, G., Shi, F.: Graph Transformations on Domain-Specific Models. Technical report, Institute for Software Integrated Systems, Vanderbilt University, ISIS- 03-403 (2003)
5. Willink, E.D.: UMLX - A Graphical Transformation Language for MDA. In: 2nd OOPSLA Workshop on Generative Techniques in the context of Model Driven Architecture, OOPSLA 2003, Anaheim (2003)
6. Kalnins, A., Barzdins, J., Celms, E.: Model Transformation Language MOLA. In: Aßmann, U., Aksit, M., Rensink, A. (eds.) MDAFA 2003. LNCS, vol. 3599, pp. 62–76. Springer, Heidelberg (2005)
7. Clark, T., et al.: Language Driven Development and MDA, BPTrends. MDA Journal (October 2004)

8. Bezivin, J., et al.: The ATL Transformation-based Model Management Framework. Research Report No 03.08, IRIN, Universite de Nantes (2003)
9. Broekstra, J., Kampman, A., Harmelan, F.V.: Sesame: A Generic Architecture for Storing and Querying RDF and RDF Schema. In: Proc. International Semantic Web Conference, Sardinia, Italy (2002)
10. Budinsky, F., et al.: Eclipse Modeling Framework. Addison-Wesley, Reading (2003)
11. Metadata Repository (MDR), `http://mdr.netbeans.org/`
12. Barzdins, J., et al.: Towards Semantic Latvia. Communications of the 7th International Baltic Conference on Databases and Information Systems (Baltic DB&IS 2006), Vilnius, pp. 203–218 (2006)
13. Kalnins, A., Barzdins, J., Celms, E., et al.: The first step towards generic modeling tool. In: Proceedings of the 5th International Baltic Conference on Databases and Information Systems, Tallin, vol. 2, pp. 167–180 (2002)
14. Kalnins, A., Barzdins, J., Celms, E.: Efficiency Problems in MOLA Implementation. In: 19th International Conference OOPSLA 2004, Workshop "Best Practices for MDSD", Vancouver, Canada (October 2004)
15. Barzdin, J.M.: Universality problems in the theory of growing automata. Dokl. Akad. Nauk SSSR (in Russian) 157(3) (1964) (English translation in: Soviet Math. Dokl. 9, 535–537 (1964)
16. Kolmogorov, A.N., Uspensky, V.A.: To the Definition of an Algorithm. Uspekhi. Mat. Nauk (in Russian) 13(4), 3–28 (1958) (English translation in: AMS Translations, ser. 2, 21, 217–245 (1963))