LATVIJAS UNIVERSITĀTES RAKSTI. 2008, 733. sēj.:
DATORZINĀTNE UN INFORMĀCIJAS TEHNOLOĢIJAS 140.–178. lpp.

The Implementation of MOLA to L3 Compiler

Agris Sostaks[1], Audris Kalnins[2]
[1,2] University of Latvia, Institute of Mathematics and Computer Science,
Raina blvd 29,LV-1459 Riga, Latvia
[1]Agris.Shostaks@gmail.com, [2]Audris.Kalnins@mii.lu.lv

Abstract. The implementation of the model transformation language MOLA compiler to the L3 language is described in the paper. It is shown that L3 is a suitable low-level model transformation language for efficient implementation of pattern matching in MOLA. A rationale for the chosen compiler architecture is offered. The detailed description of mappings from MOLA to L3 is also given. Some general approach to the graphical language compiler development, such as model-driven compiling and debugging, is also sketched.

Keywords: Graphical model transformation language, MOLA, L3, Lx, compiler, model-driven compiling.

1 Introduction

Model transformations play an important role in the Model-Driven Software Development (MDSD) [1]. The main idea of MDSD is a systematic use of **models** as primary software engineering artefacts throughout the software development lifecycle. Model-Driven Development refers to a range of development approaches that are based on the use of software modelling. A model expresses a particular aspect of a software system in a certain level of detail. A code of the software system is generated from models built by a system developer. The generated code varies ranging from a system skeleton to a complete product. It depends on the abstraction level of models used as a source for the generator. If the created models are at high level of abstraction, then **model transformations** are applied to create more detailed models that can be used for code generation. The model transformation is the automatic generation of a target model from a source model, according to a transformation definition [2]. Model transformation languages are used to define model transformations. Models that are used by model transformations must conform to **metamodels**. A metamodel defines a language which specifies a model. A model transformation language uses metamodels to define the model transformation. A meta-language specifies the metamodels. The general architecture of model transformations is shown in Fig.1.

The best known Model-Driven Software Development initiative is the Object Management Group (OMG) [3] Model-Driven Architecture (MDA) [4], which is a registered trademark of OMG. The OMG has developed the set of standards related to

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

MDA, including the Meta-Object Facility (MOF) [5] (a meta-language), Object Constraint Language (OCL) [6], Unified Modelling Language (UML) [7] (a software development language), and MOF Queries/Views/Transformations (MOF-QVT) [8] (a model transformation language).

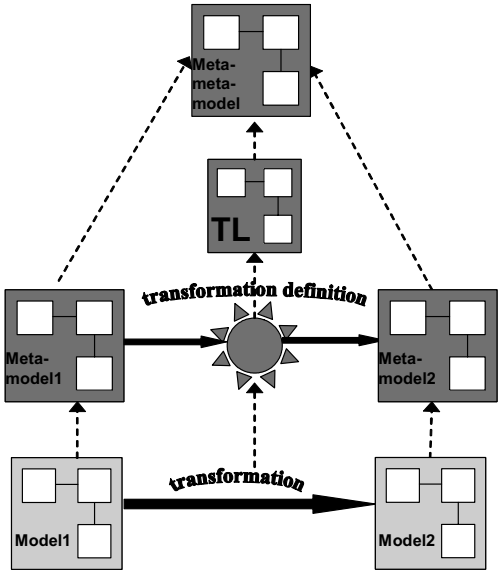


Fig. 1 Model transformation

The MDA approach defines system functionality using a platform-independent model (PIM) that is written in an appropriate modelling language (for example, UML). Then the PIM is transformed to one or more platform-specific models (PSMs), which include platform- or language-specific details. For example, the UML Profile for Java [9] can be used to specify the PSM. Then the PSM is translated to the code written in the language appropriate to the PSM.

Today the application area for model transformation languages is much broader. One such area is generic meta-model-based modelling tool building. The model transformation languages can be used (and are used [10, 11, 12]) as a much more effective domain specific substitute for the general purpose languages that are used for tool building up to now. This paper shows that model transformation languages also become appropriate facilities for compiler building. Thus, domains for applications of model transformation languages are quite different, but the typical language constructs used for model processing in all these domains are quite similar.

The OMG was the first to state precisely the requirements what should be a model transformation language [13]. The MOF-QVT language, which is an answer by OMG itself to these requirements, becomes the OMG standard for model transformations [8]. In MOF-QVT source and target meta-models conform to the MOF. There are two variants of MOF defined – the EMOF (Essential MOF) and the CMOF (Complete MOF). The MOF can be viewed as a general standard to write metamodels, but, more specifically, EMOF is used for metamodel definition in MOF-QVT. The MOF-QVT standard defines two languages of transformation development – the *Relations* and the

Operational Mappings. The *Relations* language is at the highest level of abstraction and uses patterns and a declarative transformation definition style whenever possible. This language has two semantically equivalent concrete syntaxes – a graphical and a textual one. The *Operational Mappings* language is an imperative textual language. The syntax of the *Operational Mappings* provides constructs commonly found in imperative languages (loops, conditions, etc), while the management of model elements is based on extended OCL constructs. Actually, the MOF-QVT specification [8] also contains the third language – the *Core*. The role of this language is to serve for semantic definition of the first two OMG languages and also for possible implementation of these languages. There are several realizations of the MOF-QVT language. The *Relations* textual language is implemented in the *medini QVT* [14]. The *Operational Mappings* language is implemented in the *SmartQVT* [15], several less complete implementations are also available.

There are many other model transformation languages which also satisfy the OMG requirements. There are textual model transformation languages – ATL [16], VIATRA2 [17], the Lx language family (L0-L3) [18] and also graphical model transformation languages – Fujaba [19], GReAT [20], MOLA [21]. In fact, model transformation languages existed even before the OMG coined this concept. These were the graph transformation languages, which were used to transform a source graph to a target graph in a rule-based manner. The structure of both graphs was defined by means of graph grammars which, in fact, are the same metamodels. There are several such graph transformation languages that are now being used as the model transformation languages, for example, AGG [22] and PROGRES [23].

Most of the model transformation languages rely on an EMOF-compatible meta-language for defining metamodels. For example, Fujaba and GReAT use class diagram notations close to EMOF, and ATL uses KM3 [24] (a certain extension of EMOF). Sometimes meta-languages are used that are much more expressive than EMOF, for example, VTML [25] for the VIATRA2 language. An implementation of a metamodelling language is closely related to the specific repository used for storing models.

An efficient implementation of model transformation languages is still a topical issue. There are several possibilities of implementation. A direct compilation to a general purpose programming language is a common approach (AGG, Fujaba, GReAT). The result of the compilation contains invocations of the API of the repository used to manage models and the corresponding metamodel. Another possibility is a compilation to an intermediate "very low-level" transformation language, for example, ATL uses the so called ATL byte-code [26]. It is also possible to build a direct interpreter of a model transformation language, as it is done for the VIATRA2 language.

The model transformation language MOLA is developed by the University of Latvia, Institute of Mathematics and Computer Science. This paper describes the implementation of the MOLA compiler. The MOLA compiler uses a different approach by compiling MOLA to L3, which is a lower-level textual model transformation language, but still has features typical of a transformation language. The L3 language is an imperative language which also includes imperative facilities for pattern definition; therefore, the compilation of declarative patterns in MOLA is the only complicated part of MOLA to L3 compiler realization. The L3 language is

efficient regarding implementation [27], and it is also developed by UL, IMCS. The L3 language is also used for the development of MOLA compiler. In other words, the compiler itself is built as a model transformation. Therefore, the chosen implementation is relatively simple and at the same time guarantees efficiency of implementation.

A brief introduction to the MOLA language is given in chapter 2. The experience gained in building the previous MOLA realizations is described in chapter 3. The language family Lx is introduced in chapter 4. The general architecture of the MOLA compiler and a brief overview of the model-driven compiling are given in chapter 5. Mappings from MOLA to L3 are described in details in chapter 6. Chapter 7 contains MOLA environment problem descriptions and possible solutions that are not directly related to the compiling process.

2 MOLA Language

MOLA is a graphical model transformation language, which is used for transforming an instance of a source metamodel (the source model) into an instance of the target metamodel (the target model). A transformation definition in MOLA consists of the source and target metamodel definitions and one or more MOLA procedures.

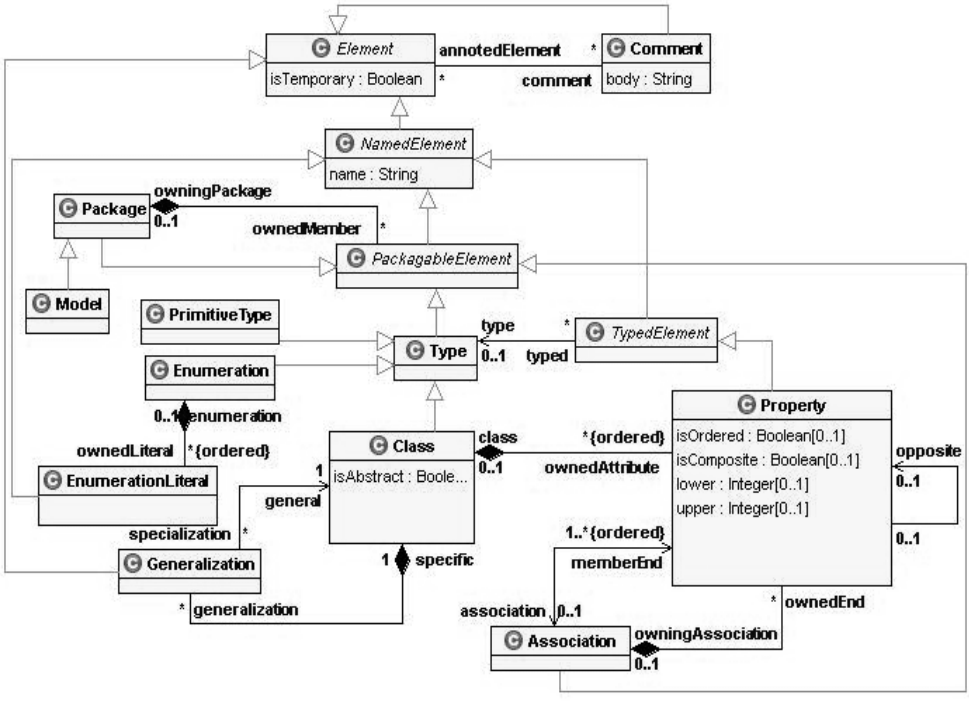


Fig. 2. The metamodel of the MOLA metamodelling language

Source and target metamodels are jointly defined in the MOLA metamodelling language, which is quite close to the OMG EMOF specification [8]. These metamodels are defined by means of one or more class diagrams, packages may be used in a standard way to group the metamodel classes. Actually, the division into source and target parts of the metamodel is quite semantic, as they are not separated syntactically (the complete metamodel may be used in transformation procedures in a uniform way). Typically, additional mapping associations link the corresponding classes from source and target metamodels; they facilitate the building of natural transformation procedures and document the performed transformations. The source and target metamodel may be the same – that is the case for in-place model update transformations. The MOLA metamodelling language is defined formally in the Kernel package of the MOLA metamodel (see Fig. 2).

MOLA procedures form the executable part of a MOLA transformation. One of these procedures is the main one, which starts the whole transformation. MOLA procedure is built as a traditional structured program, but in a graphical form. Similarly to UML activity diagrams (and conventional flowcharts), control flow arrows determine the order of execution of MOLA statements. Call statements are used to invoke sub-procedures. However, the basic language statement of MOLA procedures is specific to the model transformation domain – it is the **rule**. Rules embody the pattern match paradigm, which is typical of model transformation languages. Each rule in MOLA has the pattern and the action part. Both are defined by means of **class-elements** and **-links**. A class-element is a metamodel class, prefixed by the element ("role") name (graphically shown in a way similar to UML instance). An association-link connecting two class-elements corresponds to an association linking the respective classes in the metamodel. A pattern is a set of class-elements and -links which are compatible to the metamodel for this transformation. A pattern may simply be a metamodel fragment, but a more complicated situation is also possible – several class-elements may reference the same metamodel class – certainly, their element names must differ (these elements play different roles in the pattern, e.g., the start and end node of an edge). A class-element may also contain a constraint – a Boolean expression in a simplified subset of OCL. The main semantics of a rule is in its pattern match – an instance set in the model must be found, where an instance of the appropriate class is allocated to each class-element so that all required links are present in this set and all constraints evaluate to true. If such a match is found, the action part of the rule is executed. The action part also consists of class-elements and links, but typically these are create-actions – the relevant instances and links must be created. An end of a create-link may also be attached to a class-element included in pattern. Assignments in class-elements may be used to set the attribute values of the instances. Instances may also be deleted and modified in the action part. Thus a rule in MOLA typically is used to locate some construct in the source model and build a required equivalent construct in the target model. If several instance sets in the model satisfy the rule pattern, the rule is executed only once (on an arbitrarily chosen match). Such a situation should be addressed by another related construct in MOLA – the loop construct. In addition, the reference mechanism (a class-element may be a reference to an already matched or created instance in a previous rule) is used to restrict the available match set. Thus, rules are typically used in MOLA in situations where at most one match is possible. Certainly, there may be a situation when no

match exists – then the rule is not executed at all. To distinguish this situation, a rule may have a special *ELSE*-exit (a control flow labelled *ELSE*), which is traversed namely in this situation. Thus, a rule plays in MOLA the role of an *if-then-else* construct as well.

Another essential construct in MOLA is the **loop** (more concretely, for-each loop). The loop is a rectangular frame, which contains one special rule – the **loophead**. The loophead is a rule which contains one specially marked (by a bold border) element – the loop variable. The semantics of a for-each loop is that it is executed for all possible matches for the loophead, which differ by instances allocated to the loop variable (possible variations for other loop head elements are not taken into account). In fact, a for-each loop is an iterator which iterates through all possible instances of the loop variable class that satisfy the constraint imposed by the pattern in the loophead. With respect to other elements of the pattern in the loop head, the "existential semantics" is in use – there must be a match for these elements, but it does not matter whether there are one or several such matches. Thus a for-each loop is the main MOLA construct, which is used to code a situation: "for each instance of . . . which satisfies . . . perform the following transformation. . . ". Namely such situations in informal descriptions of model transformations are frequently called transformation rules, but in MOLA they must be formalised as for-each loops. In addition to the loophead, a loop typically contains the loop body – other MOLA statements whose execution order is organised by control flows. The loop body is executed for each iteration of the loop. Since the loop head is a rule, it may also contain create actions, thus simple transformations of source model elements may be coded in MOLA by loops consisting of the loop head only. For nested loops the main organising feature is the possibility to reference the loop variable (and other elements) of the main loop in the pattern of the nested loop head, thus specifying an iteration over all related instances (to the current instance in the main loop).

There also are other available constructs in MOLA procedures. Procedures may have **parameters** (of type of a metamodel class or a primitive type) and local **variables** (also of both types). These elements may be used in MOLA rules, in addition, **text-statements** (consisting of a constraint and assignments) may be used to process these elements more directly. For primitive-typed variables the text statement is the only option. A text statement containing a constraint (a Boolean expression) may also have an *ELSE*-exit and serve as an *if-then-else* construct (in addition to rule). Besides MOLA procedures, external (coded in an OOPL) procedures can also be invoked; this feature is used for low-level data processing (e.g., model data import). It should be noted that MOLA has no built-in UI support (MOLA is oriented towards behind-the-scenes transformations), therefore diagnostic messages and similar situations should be addressed via a library of external procedures. All MOLA procedure elements are defined formally in the MOLA package of the MOLA metamodel (see Fig. 3).

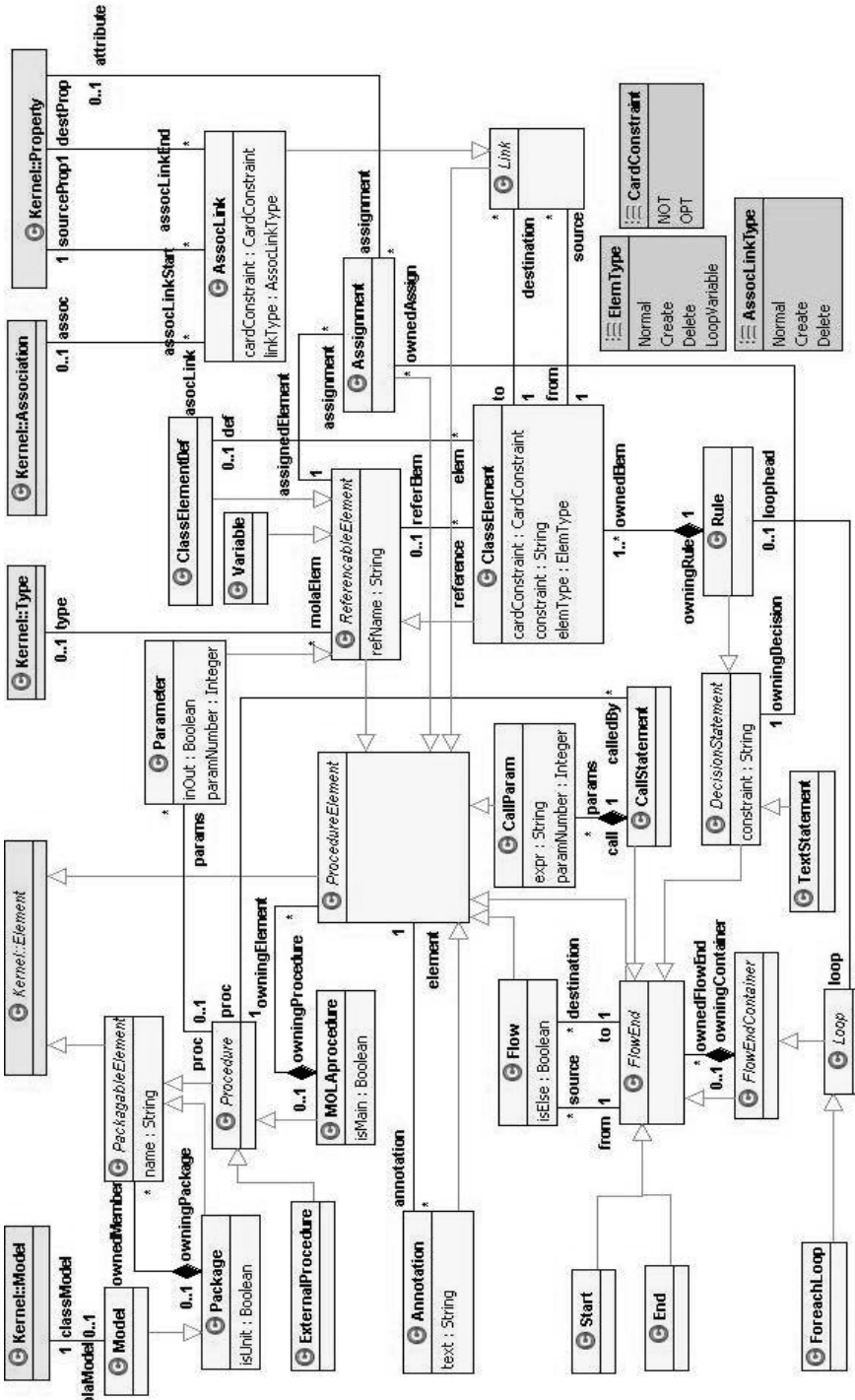


Fig. 3. The metamodel of the MOLA procedure elements

The execution of a MOLA transformation on a source model starts from the main procedure. A loop is executed while there are instances to iterate over, then the next construct according to the control flow is executed. If a rule without a valid match is to be executed, and this rule has no *ELSE*-exit, then the current procedure is terminated (if this occurs outside a loop) or the next iteration of the loop is started (within a loop body). When the main procedure reaches its end, the transformation is completed.

3 Previous Realizations of MOLA

The most critical part of the implementation of a pattern-based transformation language is the implementation of the pattern matching. It has been already shown [28] that an efficient MOLA pattern matching implementation is possible. This realization is based on only few specific low-level operations needed to iterate over a model. They are:

- `getNext(Class Cl)` – returns the next instance of a metaclass `Cl` upon each call. There is also an initialization for it – `initializeGetNext(Class Cl)`
- `getNextByLink(Association as, Cl1 inst, Class Cl2)` – returns one by one instances of a metaclass `Cl2` that can be reached by links corresponding to association `as` from a fixed instance `inst`. There is also an initialization for it, with similar parameters – `initializeGetNextByLink(Association as, Cl1 inst, Class Cl2)`
- `checkLink(Cl1 inst1, Cl2 inst2, Association as)` – checks whether a link of the required type is between these instances
- `eval(Cl inst, Expr exp)` – evaluates a local constraint on attributes

Thus, the target language of the MOLA compiler or the API of a repository that is used for realization of the MOLA interpreter (Virtual Machine) must contain similar operations. This approach requires the implementation of the pattern matching algorithm using such low-level constructs. That is a sufficiently complicated task.
Another approach that can be used for pattern matching is to rely on some powerful high-level pattern matching language and build mappings from MOLA to it. An appropriate model repository must also be chosen.

The previous realization of MOLA [29] used SQL queries as a pattern matching language and a relational database as the model repository. A fixed database schema had been defined in the most natural way by storing the metamodel in tables which correspond to the EMOF metamodel classes. The storage of model elements – instances of metamodel classes, associations, and attributes was completely straightforward in the corresponding tables. A MOLA program was also naturally stored in tables according to the MOLA metamodel. The main idea was to map a MOLA pattern to a single SQL statement. SQL queries generated by this realization were large self-join queries that are non-typical of standard database applications. The database engines were performing efficiently for queries if the number of class

elements in a MOLA pattern did not exceed a certain number. Experiments and benchmark tests had shown that the implemented MOLA Virtual Machine performed satisfactorily and MOLA is a suitable transformation language for typical MDSD tasks. However, for an industrial usage of MOLA a special in-memory repository and a compiler/interpreter that implements the principles described in [28] is required.

The next step in the realization of the model transformation language MOLA was to search for a solution which satisfies the requirements mentioned above.

4 Lx Language Family

The search for a suitable solution for the MOLA realization revealed that an appropriate language and also a repository could be found nearby. The model transformation languages Lx [18] (the so called Lx language family) fulfil the requirements mentioned in the previous chapter. Textual model transformation languages Lx contain the base transformation language L0 and its related transformation languages L0', L1, L2 and L3. Each of these languages is based on the previous language of this family by adding some extra features.

The model transformation language L3 has been chosen as a target language for the MOLA compiler. A more detailed description of the Lx language family is available in [32] and [27]; however, a brief overview of all these languages is given in this chapter in order to make this paper understandable without reading the papers mentioned above.

4.1 Lx Metamodelling Facilities

The Lx language family, as any other model transformation language, uses some sort of metamodelling language. It is quite close to the OMG EMOF specifications. The main difference is that multiple generalization is not allowed and there are no packages in this metamodelling language. The metamodel of this language is shown in Fig. 4.

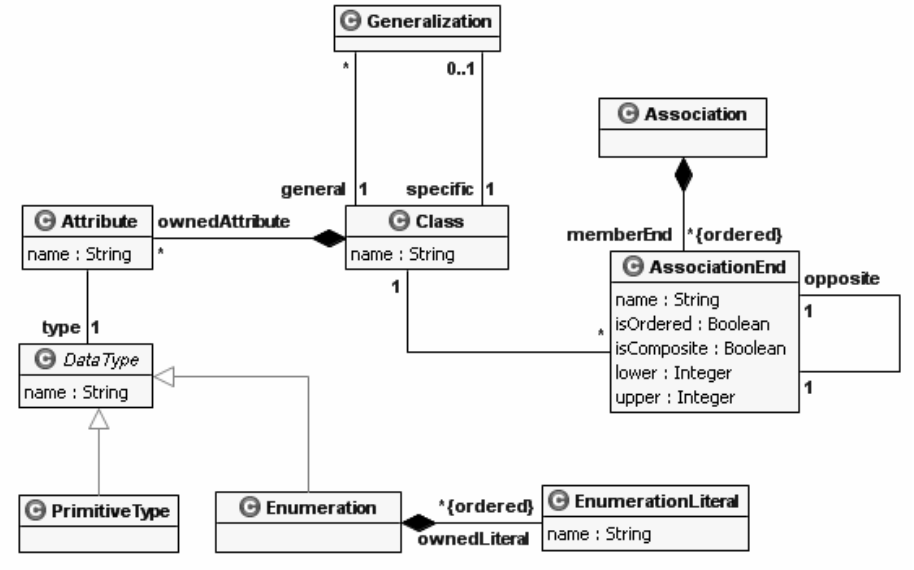


Fig. 4. The metamodel of Lx metamodelling language

Classes and binary associations are core elements of this language. Classes can have attributes which can be primitive or enumeration-typed. There are four pre-defined primitive types – *String*, *Integer*, *Boolean*, and *Real*. There are no possibilities to define new ones.
The basic commands (constructs for a textual definition of a metamodel) of the Lx family metamodelling language are the following:

- **class** <className>**;** – defines class with a given name.
- **attr** <className>**.**<attrName>**:**<ElementaryTypeName>**;** – defines attribute with a given name and type.
- **assoc** <className>**.{ordered}**<card><roleName>/<roleName><card>**{ordered}.** <className>**;** – defines association with corresponding properties.
- **compos** <compositeClassName>**.{ordered}**<card><roleName>/<roleName><card> **{ordered}.**<partClassName>**; –** defines compositions with corresponding properties.
- **rel** <subClassName>**.subClassOf.**<superClassName>**; –** defines a generalization relationship between given classes.
- **enum** <enumName>**:{** <enumLiteral1> **,** < enumLiteral2>**,** … **}; –** defines enumeration with given elements.

4.2 Language L0

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

- global variable definition part;
- native subprogram (function or procedure) declaration part (used C++ library function headers);
- L0 subprogram definition part. Exactly one subprogram in this part is the **main**. The main subprogram defines the entry point of the transformation. An L0 subprogram definition also consists of several parts:
  - Subprogram header
    - **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.
    - **function** <funcName>**(**<paramList>**):** <returnType>**; –** return type name can be an elementary type name or class name.
  - Local variable definitions

- **pointer** <pointerName> **:** <className>**;** – defines a pointer to objects of class <className>.
- **var** <varName> **:** <ElementaryTypeName>**;** – defines a variable of elementary type. <ElementaryTypeName> is one of elementary types.

  - Keyword **begin** – starts subprogram body definition
  - Subprogram body definition
  - Keyword **end -** ends subprogram body definition.

The subprogram body definition may contain the following commands:

1. **return;** – returns execution control to caller procedure or function.
2. **call** <subProgName>**(**<actPrmList>**);** – calls a subprogram. Actual parameters list can be empty. Actual parameter list consists of binary expressions separated by ",".
3. **label** <labelName>**;** – defines a label with the given name.
4. **goto** <label>**;** – unconditionally transfers control to <label>. <label> should be located in the current subprogram.
5. **first** <pointer> **:** <className> **else** <label>**;** – positions <pointer> to an arbitrary 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**).
6. **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.
7. **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>.
8. **addObj** <pointer>**:**<className>**;** – creates a new object of the class <className>.
9. **addLink** <pointer1>**.**<roleName>**.**<pointer2>**;** – creates a new link (of type specified by <roleName>) between the objects pointed to by the <pointer1> and <pointer2> , respectively.
10. **deleteObj** <pointer>**;** – deletes the object, which is pointed to by <pointer>.
11. **deleteLink** <pointer1>**.**<roleName>**.**<pointer2>**;** – deletes link whose type is specified by <roleName> between objects pointed to by <pointer1> and <pointer2>, respectively.

12. **setPointer** <pointer1>=<pointer2>**;** – sets <pointer1> to the object which is pointed to by <pointer2>. Instead of <pointer2> the *null* constant can be used.
13. **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* (+,-,*,/,&&,||,!).
14. **setAttr** <pointer>**.**<attrName>=<binExpr>**;** – sets the value of attribute <attrName> (of the object, pointed to by <pointer>) to the <binExpr> value.
15. **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>. Instead of the equality symbol == an inequality symbol != can be used. This command is used for determining the exact subclass of an object.
16. **var** <variable>**==**<binExpr> **else** <label>**;** – if the condition is *true*, then control is transferred to the next command, else control is transferred to <label>. Instead of equality symbol other (<, <=, >, >=, !=) relational operators compatible with argument types can be used.
17. **attr** <pointer>**.**<attrName> **==** <binExpr> **else** <label>**;** – if the condition is *true,* then control is transferred to the next command, else control is transferred to <label>. Other relational operators (<, <=, >, >=, !=) can be used too.
18. **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.
19. **pointer** <pointer1>==<pointer2> **else** <label>**;** – checks whether the objects pointed to by <pointer1> and <pointer2> are identical. Instead of <pointer2> *null* constant can be used. The inequality symbol (!=) can be used too.

It is easy to see that the language L0 contains only the very basic facilities for defining transformations [32].

4.3 Languages L0' – L3

Language L0' – model transformation language L0' is based on the language L0. The new feature of L0' is the possibility to make long arithmetic expressions (in L0, only unary and binary expressions were allowed).
Language L1 – is supplemented with an imperative pattern matching feature, so that it is possible to search for instances that match some condition. Any L1 pattern can contain conditions on values of variables or attributes, links between instances and other. In fact, all L1 commands can be used to specify pattern condition.
The textual syntax for the pattern (*such-that* block) is as follows:

```
suchthat
begin
<L1Commands>
end;
```

The condition holds if it is possible to successfully [27] reach the end of the block (i.e., successfully execute its last command). The "conditional" commands in L0 (commands that have an **else** branch) may be used without the **else** branch in the *such-that* block. If in such a command the undefined **else** branch is to be executed, then the condition defined by the pattern fails.
The *such-that* block may be used with **first** and **next** commands.
Language L2 – has the possibility to make loops. A special command exists in L2 with which it is possible either to visit all instances of the specified class or just those instances of the class that match the given pattern. The textual syntax for the loop is as follows:

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

Language L3 – has the branching command – a standard *if-then-else* construct can be used. The textual syntax of the branching command is as follows:

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

The L3 metamodel (the Lx language family metamodel) is shown in Fig. 5.

4.4 MOLA and L3

The main reasons why the Lx model transformation language family and the L3 language, particularly, have been chosen are described in this section.

One of the main requirements that must be met is the compatibility of metamodeling languages. In our case metamodelling languages are EMOF-based for both MOLA and Lx language family. There are no significant differences between both languages, but such minor issues like absence of packages in Lx family metamodeling language can be resolved using name prefixes for class names. Thus, we can claim that MOLA and Lx metamodeling languages are fully compatible.

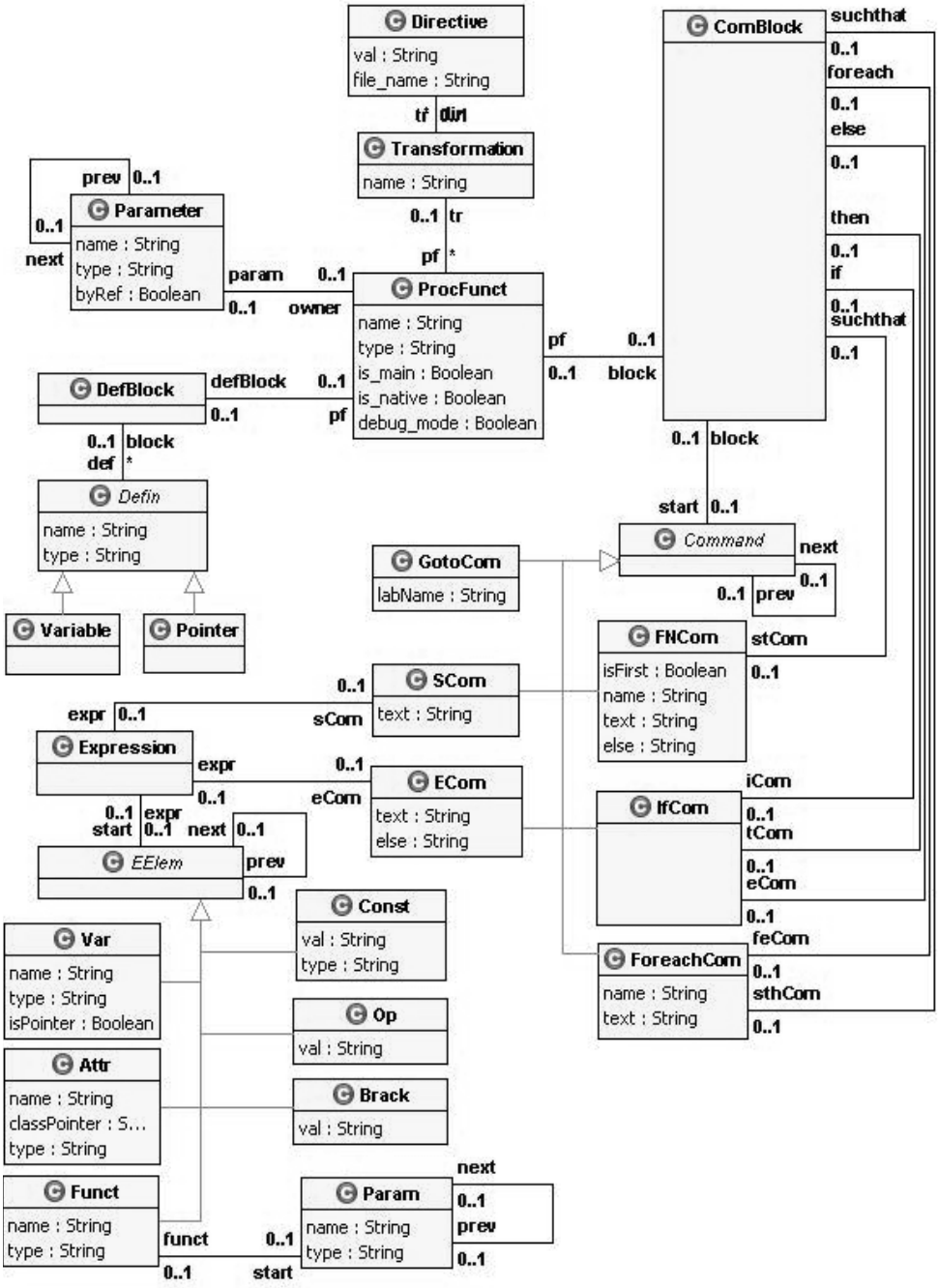


Fig. 5. The metamodel of L3 language

It has already been shown [28] that MOLA language can be implemented efficiently using a set of low-level operations for patterns. There is a direct mapping from the required operations to the commands of Lx model transformation family.

- `initializeGetNext(Class Cl)` and `getNext(Class Cl)` operations can be mapped to **first** *c:Cl* and **next** *c* commands. These commands return all instances of a given meta-class. In the beginning the **first** *c:Cl* command must be called to initialize the iteration through required instances and afterwards the **next** *c* must be called to iterate through.
- `initializeGetNextByLink(Association as, Cl1 inst, Class Cl2)` and `getNextByLink(Association as, Cl1 inst, Class Cl2)` operations can be mapped to the **first** *c:Cl2* **from** *inst* **by** *as* and **next** *c* commands. These commands return all instances of a given meta-class navigable by links of the given type from a fixed instance. The iteration must be done similarly to the previous case.
- `checkLink(Cl1 inst1, Cl2 inst2, Association as)` operation can be mapped to the **link** *inst1.as_rolename.inst2* command. The semantics of this command is the same as the semantics of this operation – check the existence of a link of the given type between two fixed instances.
- `eval(Cl inst, Expr exp)` operation is an expression interpreter and the MOLA realization to L3 must implement a generator of sequences of L3 commands that interprets the given expression. The core elements of such expressions are attribute or variable value checks. These operations can be mapped to **attr** *inst.*<*attrname*><*relation*><*expression*> and **var** <*varname*><*relation*><*expression*> commands accordingly. Arithmetic expressions can be mapped to expressions introduced by the L0' language. Constraints that are complex (Boolean) expressions where conjunction, disjunction and negation are used can be mapped to a sequence of commands which interprets the given expression.

MOLA operations that create update and delete instances and links can be mapped to **addObj**, **addLink**, **setAttr**, **deleteObj**, **deleteLink** commands. The control flows in MOLA can be mapped to **label** and **goto** commands in L3 language. L3 language as well as MOLA has such concepts as *procedure*, *parameter*, *variable*, *sub-procedure call*. These concepts can be mapped directly from MOLA to L3 language. Thus L3 language provides all necessary features that allow us to build an efficient MOLA compiler.

These basic features are included in the L0' language, but commands introduced in the following languages L1-L3 (pattern matching, looping, and branching commands) allow much easier implementation of the MOLA compiler. That is possible because these commands are at an abstraction layer much closer to MOLA concepts, such as for-each loop and rule, than basic, L0 and L0', commands.

A detailed description of the mapping from MOLA to L3 is given in chapter 6 of this paper.

5 Architecture of MOLA Compiler

This chapter describes the general architecture of the MOLA compiler. It includes the chain of compilers from MOLA to L3, L3 to L0, L0 to C++, and C++ to executable code. An introduction to the model-driven compiling is also included in this chapter.

5.1 Implementation of the Lx Language Family

An efficient compiler has been already built [18] for the Lx language family. Actually, an efficient realization of the L0 language has been built, and a compiler for each next language is built using the bootstrapping method [30]. It means that the previous language in the family is used to build the compiler for the next one (L0 for L0' compiler, L0' for L1 compiler, and so on).

The metamodel-based in-memory repository [31] developed by the UL IMCS has been chosen to store metamodel and its instances for the implementation of L0 language. This repository has an appropriate low-level API implemented as a C++ function library. Therefore, the intermediate result of the L0 compilation is a C++ program. The final result of the L0 compilation is a dynamic link library (DLL file) that can be executed over a repository instance which contains the appropriate metamodel and model and must be loaded into memory. The experiments have shown that the repository itself and the selected way of compilation to the API [32] are efficient for the implementation of a model transformation language.

The bootstrapping method used to build compilers for the rest of the Lx family languages requires that programs written in L0' to L3 must be stored in the repository that is used by L0 language. Thus, the metamodel of the L3 language is required. All other languages of the Lx family are described by the same metamodel because each next language is derived from the previous one by adding some new features; therefore, the metamodel of the last language in the chain (L3) also describes all the previous languages.

The first step in the compilation of an L3 program is to obtain a model – an instance of the L3 metamodel. It is a representation of the L3 program in the metamodel-based repository. This step is a separate step in the whole process of the compilation which requires parsing of the text file and building a model. It is implemented using a traditional programming language (C++). Obtained lexemes [33, chapter 3] are stored in the repository as a very simple lexeme model [27]. Next, the transformation language L0 is used to obtain the L3 program model from the lexeme model.

When a program model has been built, the actual compilation is being performed. The L3 (also L2, L1, L0') compiler actually is a model transformation. In this case, an in-place transformation is used – the L3 program model is overwritten by the semantically equivalent L2 program model (also L2 by L1, etc.). The final result of the chain of compilation steps is an L0 program model which is semantically equivalent to the initial L3 program given as the input file. The chain of compilation steps (from L3 to L0) can be treated as one step (the corresponding transformations are invoked one after another).

The last step in the compilation process is the code generation (a model to text transformation). An L0 language text file is generated. This step is also carried out

using the L0 language extended with native functions for file handling written in C++. Actually, only one write to file function is needed.

5.2 MOLA Compiler

Since the whole L3 compilation process has been divided into three separate steps, there is a possibility to start with any step if the appropriate model has been prepared. This fact is used by MOLA to L3 compiler – MOLA program is being compiled directly to an L3 model. This allows to decrease significantly the complexity of the implementation of MOLA to L3 compiler. Actually, it allows to use transformation language L3 to build MOLA to L3 compiler.

The first MOLA Transformation Definition Environment (MOLA Editor) [34] was built on the basis of Generic Modelling Framework [35] – a domain-specific modelling framework, developed by the UL IMCS together with the company *Exigen Services DATI*. The models (MOLA program and metamodel) were stored in a compatible format to the repository used by the L0 language. Thus, the input for the MOLA to L3 compiler, a model of a MOLA transformation, already could be obtained. In fact, no other natural representation of a MOLA program than a model can be obtained because MOLA is a graphical transformation language. The most appropriate way to implement MOLA compiler to any suitable language is by using model transformations. Thus, the first MOLA compiler was implemented using L3 language.

Since the MOLA Editor required more sophisticated features than the GMF domain specific modelling framework could offer, the next MOLA Editor – MOLA2 Tool – has been built. MOLA2 Tool uses the METAclipse framework [10], which is based on Eclipse platform [36] and model transformations. It should be noted that METAclipse uses the same repository as the L0 realization. Therefore it was possible to develop transformations for MOLA2 Tool using MOLA itself and the first MOLA compiler. The second version of MOLA to L3 compiler has been built for MOLA2 Tool, also using L3 language.

Although there are two implementations of MOLA to L3 compiler, there are no significant differences in the architecture and general ideas of the implementations of both compilers. The main difference between these two implementations is the MOLA metamodel. The MOLA metamodel for MOLA2 Tool was improved by eliminating metamodel restrictions enforced by GMF and making it more suitable for compilation. The experience and a significant part of the code from the first version of MOLA to L3 compiler is reused in the second version. This paper is based on the second version of MOLA to L3 compiler.

Compilation of a MOLA transformation is divided into four steps. Each of them is performed by a separate component – compiler. These components are:

- MOLA to L3 compiler,
- L3 to L0 compiler,
- L0 to C++,
- C++ to executable file.

The general architecture of MOLA compiler is shown in Fig. 6.

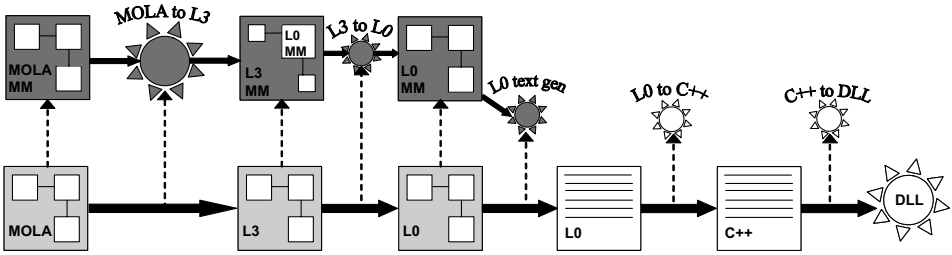


Fig. 6. The general architecture of MOLA compiler

A question may arise – why such a large number of compilers are used? Why do not use direct compilation from MOLA to C++? The answer is in the low complexity and reusability of each step. Each compiler transforms a higher-level language to a lower-level language. It is much easier to build compiler to a language that is at a closer abstraction level to the source language. Especially it is so if the general concepts of both languages are similar. This is the reason why L3 (and not L0) is used as the target language for MOLA. Another issue is the reusability. The compiler of L3 language was already built and this implementation was efficient. The efficiency of the generated code does not suffer if MOLA compiler is built on top of the compiler chain. In addition, if we will decide to implement MOLA on another EMOF compatible repository, for example, EMF [37] or Gralab [38], then only L0 compiler must be rewritten. Even less, only the actual code generator in L0 compiler must be rewritten – the lexical and syntax analyzers can be reused. The last compiler (L0 to code) is dependent on the programming language that implements the API of the model repository, but for most programming languages it is already built and free, or open-source versions are available. For example, there are free compilers for Java [39] and C++ [40]. The only disadvantage of a long compiler chain is longer compilation time, but it is not a significant problem in areas where transformation languages are used.

5.3 Model-Driven Compiling

The usage of models and transformation languages in the process of compilation is not new. The ATL model transformation language [16] has already been used to compile CPL to SPL [41] and FIACRE to LOTOS [42]. The ATL language itself is also compiled using a domain-specific language created only for this purpose – ACG (ATL Code Generation language) [43]. All of these are textual languages and the model-to-model transformation is used for actual compilation similarly to the way it was used in the example of the L3 to L0 compilation [27]. A similar idea is also used in the SmartQVT [15] implementation. The QVT code is parsed to obtain the model representation of a QVT transformation, and the actual compilation to the Java file is performed using this model.

A similar pattern of compilation is used in all examples. Three basic steps are performed:

- parse an input program and obtain the model of it,
- compile the model of the input program to the model of an output program,

- generate the code of the output program from the model.

This approach may be called **model-driven compiling** – models are used as core elements of the compilation process (see Fig. 7).

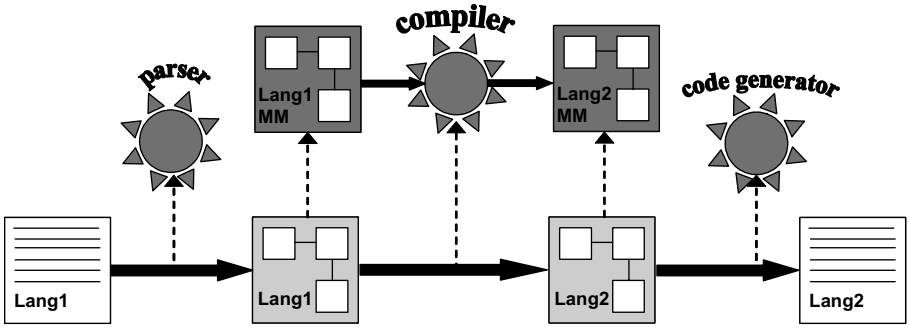


Fig. 7. Model-driven compiling – general architecture

These steps are similar to the phases of compilation in the traditional compilation technique [33, chapter 1]. The lexical and syntactical analyses are performed by the parser. The semantic analysis, intermediate code generation (target program model), and optimization are performed by compiler (model transformation). The code generation is done in the last step. The model of a source program is stored according to the language metamodel. Actually, the parse trees used in the traditional compilation technique can be treated as sort of models. Thus, the similarity is obvious.

All three steps of the model-driven compiling require appropriate metamodels already built for both input and output languages and transformation written using a model transformation language suitable for the compilation tasks. Actually, text-to-model (T2M), model-to-model (M2M), and model-to-text (M2T) languages are needed. An exporter or importer written in the general purpose programming language can be used instead of the T2M and M2T transformations. Certainly, the choice of the programming language depends on the repository used to store models.

The model-driven compiling is even more appropriate for graphical languages such as MOLA. Since programs of graphical languages are stored as models, the first step can be omitted – the model-to-model transformation that implements a compiler can be applied directly.

The main advantages of using model-driven compiling:

- The higher level of abstraction that is provided by model transformation languages allows reducing the complexity of compiler implementation.
- This is the most appropriate way to compile graphical languages because they are mostly implemented using some metamodel [37] or graph-based [38] repository. Actually, programs (diagrams) of such languages are models and the usage of a model transformation language is the most natural approach.

- If the concrete syntax of the input language is based on some general "coding" language, like XML [44], then model transformations can be applied to obtain the model of the program from its "coding". In this case, a standard parser can be used to obtain the model of the "coding". Next, the model transformation can be used to obtain the model conforming to the input language metamodel. A similar approach is also applicable for the output language.
- Since attribute grammars have been used to specify the semantics of programming languages [45], a precise definition of the model transformation between source language and target languages can be used to define the semantics of the source language in even more readable way.

The first experience in using **model-driven compiling** was quite promising. The MOLA to L3 and L3 to L0 [27] compilers have been developed. The implementation of both compilers has shown that using transformation language for compilation tasks reduces the complexity of the implementation. However, the best practice of model-driven compiling has yet to be developed, and comparison to the traditional compilation techniques [33] must be drawn.

6 Mapping from MOLA to L3

This chapter contains detailed description of the mapping from MOLA to L3. That includes mapping of metamodeling language constructs and mapping of MOLA procedure and its elements to constructs of the L3 language.

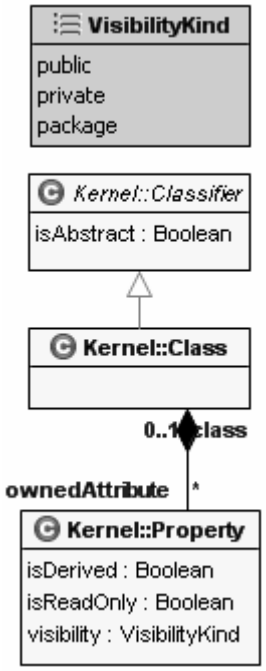
6.1 Mapping of Metamodelling Languages

Both MOLA metamodelling language and the Lx family metamodelling language are based on EMOF. So the mapping is straightforward. For the description of this mapping, we will use the meta-class names from MOLA and Lx family metamodelling language metamodels shown in Fig. 2 and Fig. 4. The MOLA related meta-class names are prefixed by the *Kernel* prefix, but the Lx related meta-class names are prefixed by the *Lx* prefix.

- Each *Kernel::Class* instance is transformed to *Lx::Class* with the same name, but since there are no packages in Lx, the *Lx::Class* name is prefixed by all parent package names. For example, the *Kernel::Class* "Lifeline", which is owned by the package "Interactions", which is in package "UML", is transformed to *Lx::Class* named "UML::Interactions::Lifeline"
- Both languages have pre-defined primitive types. All the primitive types that are in MOLA – *String*, *Integer, Boolean* – are also in Lx.
- Each *Kernel::Enumeration* instance is transformed to *Lx::Enumeration* instance and each *Kernel::EnumerationLiteral* instance is transformed to *Lx::EnumerationLiteral* instance owned by the appropriate enumeration.
- Each *Kernel::Generalization* instance is transformed to *Lx::Generalization* instance. Of course, *general* and *specific* links are set to the appropriate classes. This implementation of the L0 does not allow multiple generalization; thus, it cannot be used in MOLA either.

- Each *Kernel::Association* instance is transformed to *Lx::Association*, and appropriate association ends that are represented as *Kernel::Property* instances linked by *memberEnd* link to the association are transformed to *Lx::AssociationEnd* instances. They are linked to the appropriate class instances. Multiplicity, ordering and composition information of association ends are also transformed directly to Lx.
- Each *Kernel::Property* instance that is an attribute is transformed to an *Lx::Attribute* instance. Since MOLA allows only primitive or enumeration-typed attributes, the correspondence is direct.

An example of the transformation is given in Fig. 8.



class Kernel::Classifier;
class Kernel::Class;
class Kernel::Property;
enum VisibilityKind : {public,private,package};
compos Kernel::Class**.[0..1]**class/
ownedAttribute**[*].**Kernel::Property;
attr Kernel::Classifier.isAbstract:Boolean;
attr Kernel::Property.isDerived:Boolean;
attr Kernel::Property.isReadOnly:Boolean;
attr Kernel::
Property.AggregationKind:AggregationKind;
attr Kernel::Property.VisibilityKind:VisibilityKind;
rel Kernel::Class.**subClassOf**.Kernel::Classifier;

Fig. 8. An example of MOLA to Lx metamodelling language

6.2 Mapping of the Procedure Headers

MOLA procedures form the executable part of a MOLA transformation. The L3 language also has procedures. Both MOLA and L3 procedures may have parameters that may be *in* (passed by value) or *in-out* (passed by reference). Both languages may have variables declared. In L3, the class-typed variables and parameters are called *pointers* and have a different syntax, so compiler must distinguish class-typed variables from enumeration and primitive-typed variables. Each non-reference class-element that is used in rules in a MOLA procedure is transformed to a pointer declaration. Actually, the transformation of procedure header is straightforward and does not need detailed description. An example of the transformation of a MOLA procedure header is shown in Fig. 9 (the L3 code in all examples is used to better

illustrate the result of compilation. Actually, the compiler produces instances of the model of an L3 program)

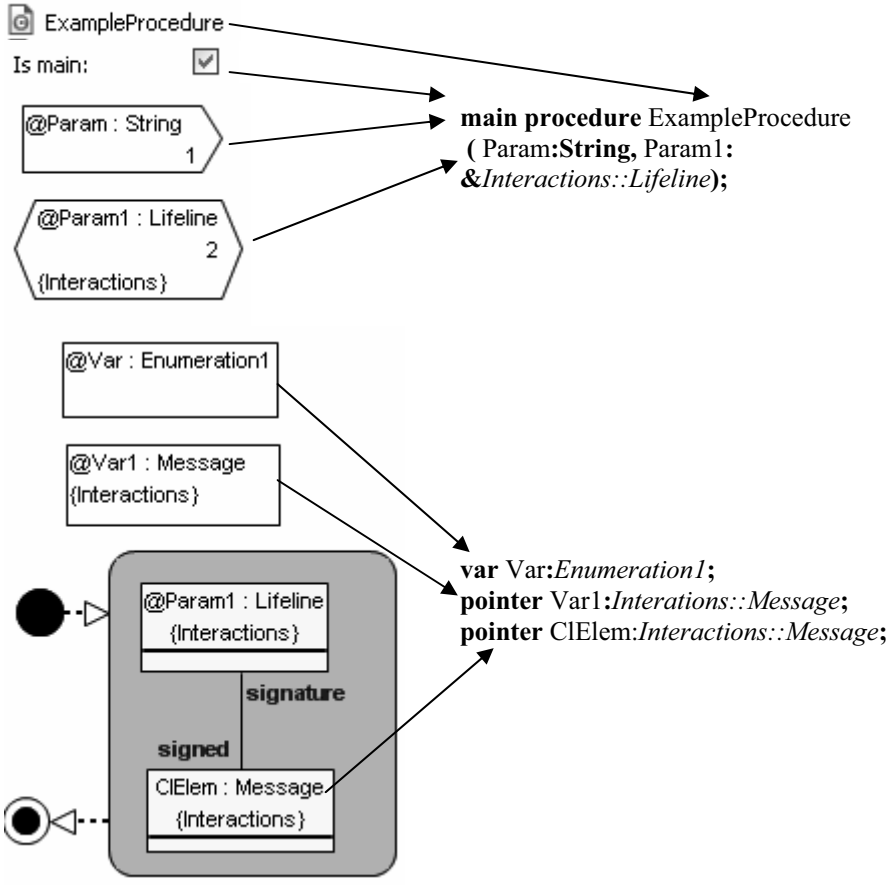


Fig. 9. Procedure header to L3

6.3 Mapping of the Execution Control Flows

The basic statements of MOLA are rule and for-each loop. There also are other MOLA statements – text-statement, call-statement, etc. Control flows are used to determine the order of execution of MOLA statements within one MOLA procedure.

There is exactly one start-statement in a MOLA procedure. It defines the entry point of the MOLA procedure. Other statements may pass the execution control to another statement or terminate the execution of the procedure. End-statements are used to terminate the execution of the procedure. They define the exit points of the MOLA procedure. The execution of the procedure may also be terminated by a text-statement or a rule if the corresponding control flow is not present. Actually, a text-statement and a rule are used as traditional branching constructs (they may have two outgoing control flows, one of them labelled *ELSE*). A for-each loop contains nested MOLA statements (loop-body) that are executed during each iteration. It has a special

statement – loop header (rule-based loophead), which defines the entry point to the loop-body. There may be any other MOLA statement in the loop (except start-statement) – nested loops are also allowed. A statement that has no outgoing control flow terminates the current iteration of the loop. A branching statement may also terminate the current iteration of the loop if one of outgoing control flows is not present. Other statements (call-statement, etc) just pass the execution control to the next statement. Control flows in MOLA procedure may connect statements in an almost arbitrary way, there are only few restrictions. Incoming control flows are not allowed to the start-statement and loophead. Outgoing control flows are not allowed from end-statements. It is not allowed to "jump" into a loop from an outside statement either (it is allowed to "jump" out).

Control flows and MOLA statements form a directed graph, where some nodes (loops) may contain a nested graph. This graph is the control flow graph (CFG) of a MOLA procedure. The control flow graph is a data structure used by traditional compilers for analysis and optimization of program execution [33, chapter 10].

The most natural way to code the control flow graph in a textual language is to use a labelled block of code for every node and a "jump" command for every edge. Thus each node of the MOLA control flow graph will compile to the block of L3 code. The block of code must start with a **label** command that unambiguously identifies the block. The execution control is passed to another code block using a **goto** command. If the execution of the MOLA procedure must be terminated, then a **return** command is used.

According to the different types of statements described above, we can distinguish five types of nodes in the control flow graph of the MOLA procedure and define the mapping to L3 language for these types:

- Entry node (start-statement) is a unique and mandatory node. Here we do a little optimization – no L3 code block is created for start-statement. The outgoing control flow determines the first MOLA statement that in turn determines the first code block of the procedure.
- Exit node (end-element) is compiled to the following code block (in what follows, a simple template language is used – L3 keywords are bolded, other parts of code are shown in angular braces containing an intuitive description):

  label <label name>**;**
  return;
- Simple node (call-statement) may not have an outgoing *ELSE* control flow. It is compiled to a simple code block – a sequence of commands depending on the actual type of MOLA statement and the **goto** command to the **label** command of the code block that is created from the MOLA statement connected by the outgoing control flow.

  label <label name>**;**
  <sequence of commands>**;**
  goto <next label name>**;**
- Branching node (rule, text-statement) may have two outgoing control flows, where one of them may be an *ELSE* control flow. It is compiled to an **if-then-else** command. The *if-block* contains the condition, *then-block* contains the action part of the MOLA rule or text-statement and *else-block* contains a

goto command to the **label** command of the code block that is created from the MOLA statement connected by the outgoing *ELSE* control flow. The last command in the main code block is the **goto** command to the **label** command of the code block that is created from the MOLA statement connected by the other (*non-ELSE*) outgoing control flow.

```
label <label name>;
if
        begin
                <condition commands>;
        end
then
        begin
                <action commands>;
        end
else
        begin
                goto <next else label name>;
        end;
goto <next label name>;
```

- Loop node (for-each loop) contains a nested control flow graph. Since a loop and its loophead can not be used separately, a common L3 code block is created for both nodes. A loop is compiled to a **foreach** command. The *such-that* block contains the condition, the *do* block contains the action part of the loophead. The *do* block also contains a **goto** command to the **label** command of the code block that is created from the MOLA statement connected by the outgoing from the loophead control flow. The last command in the *do* block is a **label** command. This label is used to receive back the execution control from the code blocks that terminate an iteration of the loop. Thus, a MOLA statement which terminates the execution of the current iteration of the loop passes the execution control to this **label** command instead of terminating the execution of the whole procedure. In fact, the execution control is passed away from the *do* block of a **foreach** command, but it is received back just at the end of an iteration. Thus, the code blocks that are created from MOLA statements within the loop body are included in the corresponding L3 loop body indirectly – using **goto** and **label** commands. The last command in the main code block is a **goto** command to the **label** command of the code block that is created from the MOLA statement connected by the outgoing control flow of the loop.

```
label <label name>;
foreach <loop variable name> suchthat
        begin
                <loophead condition commands>;
        end
do
        begin
                label <loophead label name>;
                <loophead action commands>;
                goto <loophead next label name>;
                label <loop iteration end label name>;
        end
goto <next label name>;
```

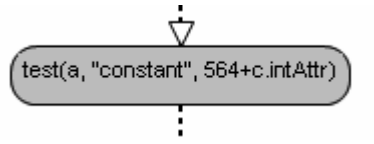
The complete code of the procedure is assembled using code blocks obtained in the way just described. The first code block is determined by the start-statement. All other code blocks may be added to the procedure in an arbitrary order because the order of execution is determined only by **label** and **goto** commands – not by the order in which command blocks are added to the procedure.
The result will be likely a sort of "spaghetti code" [46], but this causes no danger because the L3 code is just an intermediate code which is compiled further. This code is not read by a transformation developer. The wide usage of the **goto** commands does not cause any loss in the overall performance.

6.4 Mapping of MOLA Statements

The control structure aspect of the mapping of MOLA statements to L3 commands has already been described in the previous section. This section contains a detailed description of the mapping for each MOLA statement including data processing and pattern matching aspects.
The mapping for start and end statements has already been described. The start-statement is used to determine the first MOLA statement and end-statement is transformed to the **return** command.

6.4.1 Call-Statement

The **call-statement** is transformed to the **call** command. Since the mapping from a MOLA procedure to L3 procedure is one-to-one, the called L3 procedure is the same that is mapped from the MOLA procedure called by the MOLA call-statement. The L3 language allows only binary expressions to be used as actual parameters of the **call** command. MOLA allows arbitrary expressions (of appropriate type) to be used as actual parameters (the same problem is for calling functions in an expression). Our solution is to use temporary variables or pointers (depending on the actual type of a parameter) and **setVar** or **setPointer** commands to calculate the values of expressions. These commands must be executed before the **call** command. If the actual parameter is a MOLA variable, parameter, or class element identifier, then a temporary variable is not used. An example of the compilation is shown in Fig. 10.




```
var temp_var1:String;
var temp_var2:Integer;
begin
…
label id_lab1;
setVar temp_var1=”constant”;
setVar temp_var2=564+c.intAttr:Integer;
call test(a,temp_var1,temp_var2);
goto id_labx;
…
```

Fig. 10. The compilation of the call-statement

6.4.2 Text-Statement

As it was described before, the **text-statement** is transformed to the **if-then-else** command. MOLA text-statement has two main parts – a condition (constraint), which is expressed using OCL-style expression, and a list of assignments. The condition holds if the expression evaluates to *true*. The condition is compiled to the *if* block of the **if-then-else** command. Assignments are compiled to the *then* block of the **if-then-else** command.

Assignments are used in the text statement to assign values to elementary variables and pointers. The L3 commands that are used for this task are **setVar** and **setPointer**. In MOLA the value that is being assigned is expressed using a *simple expression* of an appropriate type. A simple expression of *Integer* type may contain *Integer-typed* variable, parameter or attribute specifications, *Integer* constants, pre-defined functions (*size, indexOf, toInteger*) and arithmetic operations (addition, subtraction, multiplication). A simple expression of *String* type may contain *String-typed* variable, parameter or attribute specifications, *String* constants, pre-defined functions (*toLower, toUpper, substring, toString*), and a concatenation operation. A simple expression of *Boolean* type may contain *Boolean-typed* variable, parameter or attribute specifications, *Boolean* constants (*true* and *false*), or pre-defined function (*isTypeOf, isKindOf, toBoolean*). A simple expression of *enumeration* type may contain *enumeration-typed* variable, parameter or attribute specification, *enumeration* literals or a pre-defined function *toEnum*. A simple expression of *class* type may contain a *class-typed* variable or parameter specification (pointer), *null* constant or typecast.

In L3 similar expressions are allowed, but there are a few differences: there is no direct typecast of a pointer, actual parameters in a function call may be only a binary expression of an appropriate type. The list of pre-defined functions in L3 does not match all the pre-defined functions of the MOLA language either. The solutions to these problems are rather simple. In addition, some kinds of expressions in L3 allow more features than in MOLA, but these features are not relevant for MOLA compiler. The complete table of correspondence is shown in Table 1.

Table 1. Correspondence of elements used in expressions in MOLA and L3

MOLA	**L3**
String, Integer, Boolean, enumeration-typed constants, NULL constant	+
elementary variables, pointers	+
attribute specification	+
+,-,*,concatenation	+
direct typecast (class-typed)	temporary variable and extra **setPointer** command used
function call	temporary variables and extra **setVar** commands for complex parameters used

pre-defined functions	extended library of native functions used
toEnum, toInteger, toString, toBoolean	+
indexOf, toLower, toUpper	extended library used
size, substring	+
isTypeOf, isKindOf	temporary variable and **type** command used

The left column describes features used in MOLA expressions and the right column shows the correspondence in L3. The plus sign (+) means that the mapping is direct. If there is no direct mapping, the basic principles of a solution are shown. It may be the usage of a temporary variable (typecast and function call) or the usage of an extended library of native functions (*indexOf, toLower, toUpper* functions).

Though L3 expressions allow Boolean operations, they cannot be used with relations. Relational operators (<, >, etc) may be used only in **var** and **pointer** commands. That makes the compilation of *Boolean* expressions used in MOLA more difficult.

In MOLA the simplest condition is a simple expression of the *Boolean* type. Then it is compiled using a temporary variable and a **var** command in the following way:

Condition:	**if** **begin**
<simple boolean expression>	[<extra commands>] **setVar** temp_var=<simple boolean expression>**;** **var** temp_var**==true**; **end…**

Usually a condition also contains a relation (>, <, >=, <=, =, <> operators can be used). Since the left and the right operands may be arbitrary expressions of the same type, the value of each expression is computed and stored in a temporary variable. Then these variables are compared using a **var** or **pointer** command depending on the type of expressions.

Condition:	**if** **begin**
<expression1><relation> <expression2>	[<extra commands>] **setVar/setPointer** temp_var1=<expression1>**;** [<extra commands>] **setVar/setPointer** temp_var2=<expression2>**;** **var/pointer** temp_var1<relation>temp_var2**;** **end** **...**

A condition in MOLA may also contain Boolean operations – conjunction (**and**), disjunction (**or**), and negation (**not**) – together with relational operators. The L3 has no such features, but it is shown [18, chapter 4] that it is possible to construct L3 code that implements the Boolean operations. The algorithm implemented in MOLA to L3 compiler uses the same principles.

Our template language will be used to explain this algorithm. An extension of the template language is required – let us define a function

PrintBooleanExpression(variable_name,boolexpression) that returns **the block of L3 code** that calculates the value of the Boolean expression *boolexpression* and stores it in the variable whose name is passed by the parameter *variable_name*. The use of this function means that the code block returned by the function replaces the function call. We will also need an auxiliary procedure *CreateBooleanVariable(varname)*, which adds the declaration of a new *Boolean* variable whose name is passed by the parameter *varname*. Variable and label names having a prefix *unique* are considered to be unique within the procedure.

If the parameter *boolexpression* is a simple expression of type *Boolean* or a relation, then the function *PrintBooleanExpression* will return the following code:

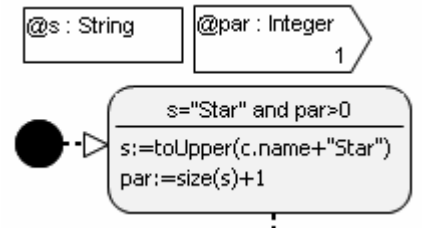
boolexpression=<simple boolean expression>	[**<**extra commands**>**] **setVar** *variable_name* = <simple boolean expression>**;**
boolexpression= <*expression1*><*relation*> <*expression2*>	**setVar** *variable_name* =**false**; [<extra commands>] **setVar** unique_temp_var1=<expression1>; [<extra commands>] **setVar** unique_temp_var2=<expression2>; **var** unique_temp_var1<relation> unique_temp_var2 **else** unique_label; **setVar** *variable_name* =**true**; **label** unique_label;

If the parameter *boolexpression* contains Boolean operators **and, or, not,** then the function will return the following code

boolexpression= *boolexpression*1 **or** *boolexpression2*	*CreateBooleanVariable ("unique_temp_var1")* *CreateBooleanVariable ("unique_temp_var2")* *PrintBooleanExpression("unique_temp_var1", boolexpression*1*)* *PrintBooleanExpression("unique_temp_var2", boolexpression2)* **setVar** *variable_name*=**true;** **var** *unique_temp_var1*==**false else** unique_label**;** **var** *unique_temp_var2*==**false else** unique_label**;** **setVar** *variable_name*=**false;** **label** unique_label**;**
boolexpression= *boolexpression*1 **and** *boolexpression2*	*CreateBooleanVariable ("unique_temp_var1")* *CreateBooleanVariable ("unique_temp_var2")* *PrintBooleanExpression("unique_temp_var1", boolexpression*1*)* *PrintBooleanExpression("unique_temp_var2", boolexpression2)* **setVar** *variable_name*=**false;** **var** *unique_temp_var1*==**true else** unique_label**;** **var** *unique_temp_var2*==**true else** unique_label**;** **setVar** *variable_name*=**true;** **label** unique_label**;**

boolexpression= **not** *boolexpression1*	*CreateBooleanVariable ("unique_temp_var1")* *PrintBooleanExpression("unique_temp_var1", boolexpression1)* **setVar** *variable_name*=**true***;* **var** *unique_temp_var1*==true **else** unique_label; **setVar** *variable_name*=**false**; **label** unique_label;

An example of the compilation of a MOLA text-statement is shown in picture Fig. 11.




```
if begin
    setVar _mvar_6=false;
    setVar _mvar_9=s;
    setVar _mvar_10="Star";
    var _mvar_9==_mvar_10 else
    _mlabel_8;
    setVar _mvar_6=true;
    label _mlabel_8;
    setVar _mvar_7=false;
    setVar _mvar_12=par;
    setVar _mvar_13=0;
    var _mvar_12>_mvar_13 else
    _mlabel_11;
    setVar _mvar_7=true;
    label _mlabel_11;
    setVar _mvar_4=false;
    var _mvar_6==true else _mlabel_5;
    var _mvar_7==true else _mlabel_5;
    setVar _mvar_4=true;
    label _mlabel_5;
    var _mvar_4==true;
end then begin
    setVar _mvar_14=
    c.name:String+"Star";
    setVar s= toUpper(_mvar_14);
    setVar par= Length(s)+1;
end else begin
    return;
end;
```

Fig. 11. The compilation of the text-statement

6.4.3 Rule

Another, and the most important, decision statement in MOLA is a **rule**. It is also compiled to the **if-then-else** command. The condition of the rule is expressed using a pattern. The implementation of pattern matching typically is the most demanding component to implement and also the key factor determining the implementation efficiency. The efficiency of the implementation of the pattern matching is not the

central theme of this paper. The chosen realization of the pattern matching implements some ideas that have been already described in [28]. This approach guarantees sufficient efficiency of the pattern matching for typical MOLA use cases.

The basic elements of the pattern are class-elements and association-links. A class-element represents the instance of the particular class. There are several types of class-elements, but only *normal* and *delete* class-elements are used to specify a pattern. Let us call them *pattern elements*. In addition, only *normal* and *delete* association-links are used to specify a pattern. Let us call them *pattern links*. Pattern elements and pattern links form the *pattern graph*. Pattern elements that are linked by pattern links form the *pattern fragment* (connected component of the pattern graph). A pattern may contain several pattern fragments that can be treated as separate patterns. All pattern fragments must match for the whole pattern to match. The main goal of the pattern matching is to find particular instances that match the given pattern. The sought instances are represented by non-reference pattern elements. The pattern links, reference class elements, and constraints on class elements form the *pattern constraint*. Actually, *such* a set of instances is sought *that* matches the pattern constraint.

The pattern is compiled to a block of L3 code which is placed in the *if* block of the **if-then-else** command. Several pattern fragments are compiled to separate L3 code blocks following each other. Natural constructs in L3 language that implement patterns are **first-suchthat** and **first-from-by-suchthat** commands. A pattern fragment is thus compiled to a nested **first-suchthat** or **first-from-by-suchthat** command.

To achieve this goal, the pattern graph must be traversed and appropriate commands built. The classical graph traversing techniques are used – a recursive algorithm that marks already traversed nodes and edges [47].

The first task is to decide which pattern element will be processed first – let us call it a *root node*. This is an important task because this decision affects the overall performance of the pattern matching. The main idea is to reduce the number of instances that must be examined to match or fail the pattern. If the pattern fragment contains a reference element, then the traversing of the pattern graph must be started from this element. This version of MOLA language also allows to denote the root element manually, using special compiler-related annotations.

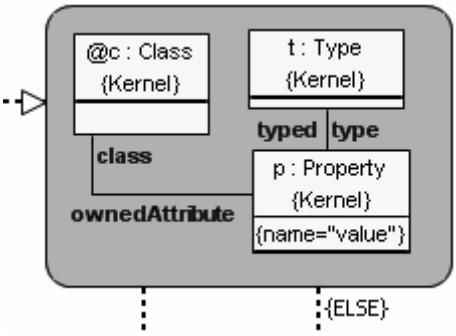
The algorithm starts the processing of the graph with the root node:

- *root node* – is marked as *traversed*.
    - If it is a non-referenced class-element, then the **first-suchthat** command is created. The *such-that* command block of the command is selected as the *current command block*. L3 commands that are obtained from the local constraint of the class-element are placed in the *such-that* block of the created command.
    - If it is a referenced class element, then L3 commands that are obtained from the local constraint of the class element are placed in the *if* block of the **if-then-else** command.
    - All nodes connected by adjacent edges (pattern links that have not yet been traversed) are processed.
- Other (non-root) *nodes* are processed in the following way – the edge which is used to reach this node is marked as *traversed*.

- If the node has been already traversed, then a **link** command is added to the *current command block*.
- If the node has not been traversed, then it is marked as *traversed*.
  - If it is a reference class-element, then a **link** command is added to the *current command block*. L3 commands that are obtained from the local constraint of the class element are placed in the *if* block of the **if-then-else** command.
  - If it is a non-reference class-element, then the **first-from-by-suchthat** command is added to the *current command block*. The *such-that* command block of the this command is selected as the *current command block*. L3 commands that are obtained from the local constraint of the class element are placed in the *such-that* block of the created command.
  - All nodes connected by adjacent edges that have not yet been *traversed* are processed.

The local constraints of pattern elements are processed in the same way as the condition of the text-statement.

An example of the compilation of a pattern is given in Fig. 12.




```
if begin
    first p:Kernel::Property from c
    by ownedAttribute suchthat
    begin
      setVar _mvar3=p.name:String;
      setVar _mvar4=”value”;
      var _mvar3==_mvar4;
      first t:Kernel::Type from p by
      type;
    end;
end
then
….
```

Fig. 12. The compilation of the rule-pattern

Actually, the algorithm described above realizes the principles of MOLA Virtual Machine described in [28]. This algorithm builds an efficient L3 code if MOLA language constructs are used in a natural way. The practical usage of MOLA compiler has also shown that the natural use of MOLA constructs leads to an efficient pattern matching. Thus, the current implementation is sufficient enough for typical tasks (MDA, tool building). However, the algorithm can be enhanced in order to achieve a better performance in less typical situations. For example, if the pattern does not contain a reference pattern-element or annotated pattern-element, then a more detailed analysis of the pattern graph should be performed. The multiplicities of the associations that correspond to the association-links used in the pattern could be analyzed. The direction of traversing the graph should be chosen so that the “going” along an association in the direction of ‘*’ multiplicity is minimized. More complicated algorithms for the pattern matching have been used typically in rule-

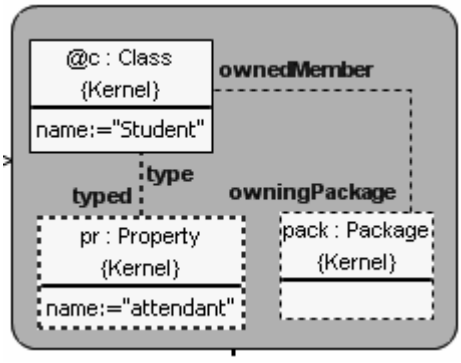
based transformation languages, for example, VIATRA [48]. This problem (the pattern matching efficiency) is not the main topic of this paper; therefore, it is not discussed in-depth.

The action part of a rule consists of class-elements, association links, and attribute assignments that are included in class elements. The *create* and *delete* class-elements are used to create and delete particular instances. The *create* and *delete* association-links are used to create and delete links. The assignment is used to assign the value of the attribute of a particular instance. The value is specified by using expressions that have been already described in previous sections. The correspondence between MOLA and L3 constructs is shown in Table 2.

Table 2. Correspondence of constructions used in the action part of the rule

MOLA	**L3**
create, delete class-elements	**addObj**, **deleteObj** commands
create, *delete* association-links	**addLink**, **deleteLink** commands
attribute value assignments	**setAttr** commands

The L3 code that is created for the action part of the rule is placed in the *then* block of the **if-then-else** command. An example of the compilation of the action part of a rule is shown in Fig. 13.




```
If begin …end
then begin
    addObj pr:Kernel::Property;
    addLink pr.type.c;
    setAttr c.name="Student";
    setAttr pr.name="attendant";
    deleteLink
    c.owningPackage.pack;
    deleteObj pack;
end else
…
```

Fig. 13. The compilation of the rule – action part

6.4.4 For-each loop

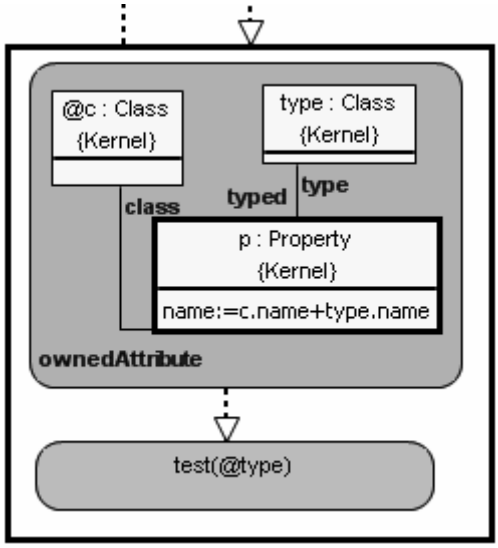
The last MOLA statement described in this chapter is the **for-each loop**. The implementation of a loop is one of the crucial issues in the implementation of the MOLA compiler. An incorrectly chosen search structure may cause serious efficiency problems.

The condition of a loop is expressed by using the pattern of the loophead, which contains a special class-element – the *loop variable*. The iteration is performed over all instances that correspond to the loop variable.

The loop is compiled to the **foreach** command. The condition of the loop is compiled to the *such-that* block of the **foreach** command. The compilation of the loophead pattern is similar to the compilation of the rule pattern. The pattern match starts from the loop variable (it is chosen as the *root node*). Usually there is a *restriction-path* – a path from a referenced class element to the loop variable where all multiplicities of the corresponding associations are **'0..1'** or **'1'.** Then for this path, **first-from-suchthat** commands are created and added to the code block before the **foreach** command. The loop variable is used as the loop variable in the **foreach** command. All nodes and edges that have been already processed (appropriate commands built for the loop variable and class-elements in the restriction path) are marked *traversed*, and the algorithm used for the compilation of a rule is executed.

This algorithm is not the most optimal either, but it is suitable for most of typical examples – usually there is a restriction path. Further optimization of the algorithm is not addressed in this paper.

The action part of the loophead is compiled in the same way as the action part of a rule. The created code is added to the *do* block of the **foreach** command. Fig. 14 illustrates an example of the compilation of a loop.




```
foreach p:Kernel::Property from c
by ownedAttribute suchthat
begin
   first type:Class from p by
   type;
end do begin
   setAttr p.name=c.name:String
   + type.name:String;
   goto _mlabel_10;
   label _mlabel_9;
end;
goto _mlabel_23;
...
label _mlabel_10;
call test(type);
goto _mlabel_9;
...
```

Fig. 14. The compilation of the loop

The mapping of the most important MOLA constructs to L3 has been defined in this chapter.

7 The surrounding of the MOLA compiler

This chapter introduces the problems that have been discovered during the implementation of the MOLA compiler. The compiler is the most important part of the implementation of a programming or transformation language. However, there are other parts needed in a proper development environment.

7.1 Error handling in MOLA

The compiler detects syntax errors in a program. Usually a development environment of a textual programming language provides the possibility to navigate to errors in a code. A list of errors is shown and the appropriate "problematic" line of code is highlighted. Similar requirements can also be applied to the MOLA development environment. Since MOLA is a graphical language, there are no "lines of code", as it is in textual languages. Each element that has a visual representation (MOLA statement, class-element, etc) can be treated as a "line of code". The MOLA compiler must detect errors in a program and point to the appropriate element. Actually, MOLA compiler does not "know" anything about the visual representation of a MOLA element. Thus, the visualization of an error is done by the development environment.

Our solution is to store the error information in the *error model*. The *error metamodel* is very simple (see Fig. 15).

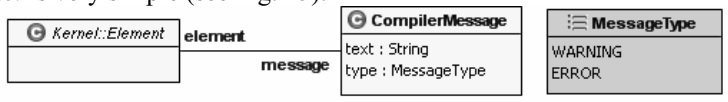


Fig. 15. The error metamodel

In fact, there is only one class (*ErrorMessage*). It represents a particular error. There are two attributes – the attribute *text* contains the textual information and *type* determines whether it is a warning or an error. The association *element* represents an "error pointer" to the appropriate element in a MOLA transformation (any MOLA element inherits from the *Kernel::Element*, see Fig. 3). The MOLA compiler deletes the existing error model and creates a new one in the process of compilation. The MOLA2 Tool reads the error model and visualizes it. An example of the error visualization is shown in Fig. 16.

The list of errors is shown in the properties tab. It is possible to navigate to the corresponding MOLA procedure from there. The elements pointed by the compiler are highlighted. This is an adequate way to treat the error handling problem in a graphical language.

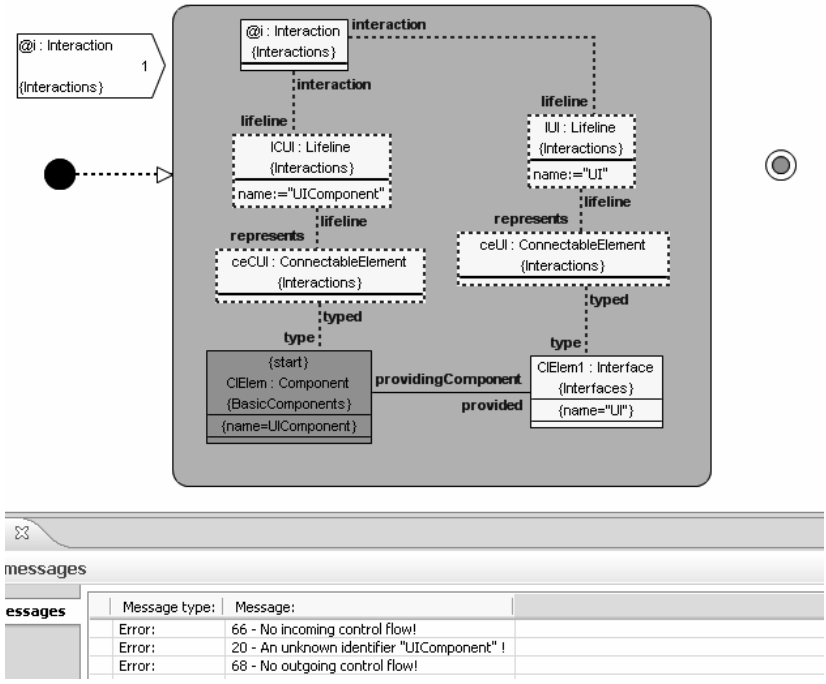


Fig. 16. The visualization of errors in a MOLA procedure.

7.2 Structuring a program in MOLA

Another feature provided by modern development environments is the possibility to compile only part of the code if the whole program has already been compiled. This is needed for large programs, when a compilation takes a significant amount of time. To achieve this goal, the program has to be structured. The most common approach is to use code units. Each unit is compiled to a separate object. Next, a linker is used to obtain a single executable.

A similar idea is also used in the MOLA2 Tool. Packages are used to structure a MOLA program. A package may be defined as a *MOLA unit*. It means that all MOLA procedures that are contained by the unit are compiled to a separate L0 unit. This allows using L0 compiler as a linker that assembles all L0 units into one C++ project. Thus, model transformations (MOLA and L3-L0'compilers) can work with smaller models that helps to improve the overall performance of the compilation process.

7.3 Debugging in MOLA

If a program is successfully compiled, it means that it is syntactically correct, but it does not mean that the program is semantically correct. Testing is a common approach used by a program developer. If a bug is found, then it must be fixed. This

process is called *debugging*. The debugging process requires a tool support to ease this process. Tools used for debugging are called *debuggers*.

Typically, debuggers offer functions such as running a program step by step and pausing the program to examine the current state of the program to track the values of some variables. Some debuggers have the ability to modify the state of the program while it is running. The importance of a good debugger is very high. The existence of such a tool can often be the deciding factor in the use of a language, even if another language is more suited to the task.

However, a debugger for the MOLA2 Tool has not yet been developed. There are examples of a debugger of a graphical language, for example, the UML Model Debugger [49]. There are differences between the debugger of a textual language and the debugger of a graphical language. The main difference is in the representation of the single-stepping approach. Since graphical languages are usually represented in diagrams, an animation of the program execution is required. Other representations could also be used, but they would be rather far from the concepts of the language.

An interpreter or instrumentation by an additional code in the compilation result may be used for the debugging purposes. The execution of a single MOLA statement could be considered as one step in the step-by-step debugging process. The result of the compilation of a MOLA program is L3 code. Since this code consists of code blocks that correspond to one MOLA statement, these blocks could be supplemented with a debugging code in a rather simple way.

There is another widely used but not so sophisticated way of the debugging. The trace (log) files can be used to trace the execution of a program. The current version of the MOLA compiler uses the L0 debugging feature – the L0 trace file. It logs an execution of every L0 command. However, the L0 tracing operates with L0 concepts. Therefore, a tracing that is at a closer abstraction level to the MOLA is needed.

8 Conclusions and Future Work

A sufficiently efficient implementation of the MOLA to L3 compiler has been described in this paper. The MOLA compiler has already been used practically in the area of tool building. The transformations that are used for implementation of the MOLA2 Tool within the METAclipse framework are developed using the MOLA to L3 compiler. The MOLA2 Tool that includes the second version of the MOLA compiler is successfully being used in the European IST 6th framework project ReDSeeDS [50]. Traditional MDA tasks are being implemented in MOLA there. These tasks include transformations from formalized software requirements to an architecture model of the system to be built and then to a detailed design model. Thus, the efficiency of the chosen architecture has been approved by practical usage. In both cases, non-trivial MOLA transformations have been developed and applied to sufficiently large models.

On the one hand, the future work is related to the problems discussed in chapter 7. The practical usage of MOLA has shown that the problem of debugging is quite significant. It should be noted that building both a user-friendly and sufficiently high-level debugger for model transformation languages, especially for graphical ones, is quite a challenging task. On the other hand, improvements in the implementation of the MOLA compiler are also expected – a more advanced algorithm of pattern

matching for MOLA will be developed. These improvements should ensure more efficient execution for less typical MOLA transformations. In addition, the model-driven compiling briefly sketched in this paper also deserves a more detailed research.

References

1. Volter M. and Stahl T., Model-Driven Software Development. John Wiley & Sons, 2006.
2. A.G. Kleppe, J.B. Warmer, & W. Bast, MDA explained: The model driven architecture: Practice and promise (Boston: Addison-Wesley, 2003)
3. The Object Management Group (OMG) URL: http://www.omg.org/
4. OMG Model-Driven Architecture URL: http://www.omg.org/mda/
5. Meta Object Facility (MOF) 2.0 Core Specification URL: http://www.omg.org/docs/ptc/04-10-15.pdf
6. OCL 2.0 Specification Version 2.0 URL: http://www.omg.org/docs/ptc/05-06-06.pdf
7. OMG Unified Modelling Language (UML), version 2.1.1 URL: http://www.omg.org/technology/documents/formal/uml.htm
8. Meta Object Facility (MOF) 2.0 Query/View/Transformation Specification URL: http://www.omg.org/docs/ptc/07-07-07.pdf
9. Metamodel and UML Profile for Java and EJB Specification URL: http://www.omg.org/docs/formal/04-02-02.pdf
10. Kalnins, A., Vilitis, O., Celms, E., Kalnina, E., Sostaks, A., Barzdins, J.: Building Tools by Model Transformations in Eclipse. Proceedings of DSM'07 workshop of OOPSLA 2007, Montreal, Canada, Jyvaskyla University Printing House, 2007, pp. 194–207.
11. I. Rath, D. Varro. Challenges for advanced domain-specific modelling frameworks. Proc. of Workshop on Domain-Specific Program Development (DSPD), ECOOP 2006, France.
12. Ermel, C., Ehrig, K., Taentzer, G., Weiss, E.: Object Oriented and Rule-based Design of Visual Languages Using Tiger. Proceedings of GraBaTs'06, 2006, pp. 12
13. Request for Proposal: MOF 2.0 Query / Views / Transformations RFP URL: http://www.omg.org/docs/ad/02-04-10.pdf
14. ikv++ - mediniQVT URL: http://www.ikv.de/index.php?option=com_content&task=view&id=75&Itemid=77
15. SmartQVT URL: http://smartqvt.elibel.tm.fr/index.html
16. ATL. URL: http://www.eclipse.org/m2m/atl/
17. VIATRA2 URL: http://dev.eclipse.org/viewcvs/indextech.cgi/gmt-home/subprojects/VIATRA2/index.html
18. 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.
19. T. Fischer, J. Niere, L. Torunski, and A. Zundorf. Story diagrams: A new graph rewrite language based on the Unified Modelling Language. In G. Engels and G. Rozenberg, editors, Proc. of the 6th International Workshop on Theory and Application of Graph Transformation, volume 1764 of LNCS, pages 296–309. Springer Verlag, 1998.
20. 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

21. Kalnins, A., Barzdins, J., Celms, E.: Model Transformation Language MOLA. Proceedings of MDAFA 2004, Vol. 3599, Springer LNCS, 2005, pp. 62–76.
22. C. Ermel, M. Rudolf, and G. Taentzer. The AGG Approach: Language and Tool Environment. In H. Ehrig, G. Engels, H. J. Kreowski, and G. Rozenberg, editors. Handbook of Graph Grammars and Computing by Graph Transformation, Vol. 2: Applications, Languages and Tools, pages 551–603. World Scientific, 1999
23. Schürr, A., Winter, A., Zündorf, A.: The PROGRES approach: Language and environment. In Ehrig, H., Engels, G.,Kreowski, H.J., Rozenberg, G., eds.: Handbook on Graph Grammars and Computing by Graph Transformation: Application, Languages, and Tools. Volume 2. World Scientific (1999) pp. 487–550
24. F. Jouault and J. Bézivin. KM3: a DSL for Metamodel Specification. In Procs. FMOOD'06, volume 4037 of LNCS, pages 17–185
25. Balogh, A., Varro, D. Advanced Model Transformation Language Constructs in the VIATRA2 Framework, ACM SAC2006, Dijon, France, 2006
26. ATL: Atlas Transformation Language Specification of the ATL Virtual Machine URL: http://www.eclipse.org/m2m/atl/doc/ATL_VMSpecification%5Bv00.01%5D.pdf
27. E. Rencis, Model Transformation Languages L1, L2, L3 and Their Implementation, Articles of the University of Latvia, "Computer Science and Information Technologies" 2008.
28. Kalnins A., J. Barzdins, E. Celms. Efficiency Problems in MOLA Implementation. 19th International Conference, OOPSLA'2004 (Workshop "Best Practices for Model-Driven Software Development"), Vancouver, Canada, October 2004
29. A. Kalnins, E. Celms, A. Sostaks. Simple and Efficient Implementation of Pattern Matching in MOLA Tool. Proceedings of the 7th International Baltic Conference on Databases and Information Systems (Baltic DB&IS'2006), Vilnius, Lithuania, July 3–6, 2006, pp. 159–167.
30. B. Efron, R.J. Tibshirani, "An Introduction to the Bootstrap", Chapman & Hall/CRC, 1994, 436 p
31. Barzdins, J., Barzdins, G., Balodis, R., Cerans, K., Kalnins, A., Opmanis, M., Podnieks, K.: Towards Semantic Latvia. Proceedings of Seventh International Baltic Conference on Databases and Information Systems, Communications, Vilnius, Lithuania, O. Vasileckas, J. Eder, A. Caplinskas (Eds.), Vilnius, Technika, 2006, pp. 203–218.
32. S. Rikacovs, The base transformation language L0+ and its implementation, Articles of the University of Latvia , "Computer Science and Information Technologies", 2008
33. A. Aho, R. Sethi, J. Ullman, Compilers: Principles, Techniques, and Tools. Bell Laboratories, 1986
34. A. Kalnins, E. Celms, A. Sostaks. Tool support for MOLA. Fourth International Conference on Generative Programming and Component Engineering (GPCE'05). Proceedings of the Workshop on Graph and Model Transformation (GraMoT), Tallinn, Estonia, September 2005, pp. 162–173
35. Celms E., A. Kalnins, L. Lace. "Diagram definition facilities based on metamodel mappings". Proceedings of the 18th International Conference, OOPSLA'2003 (Workshop on Domain-Specific Modeling), Anaheim, California, USA, October 2003, pp. 23–32
36. Eclipse – an open development platform. URL: http://www.eclipse.org/
37. Eclipse Modelling Framework (EMF, Eclipse Modelling subproject), http://www.eclipse.org/emf/
38. Peter Dahm and Friedbert Widmann. GraLab - Das Graphenlabor. Projektbericht 4.3.0, University of Koblenz-Landau, Institute for Software Technology, 07 2003.

39. Java Technology URL: http://java.sun.com/
40. GCC, the GNU Compiler Collection URL: http://gcc.gnu.org/
41. Jouault, F., Bezivin, J., Consel, C., Kurtev, I., Latry, F. Building DSLs with AMMA/ATL, a Case Study on SPL and CPL Telephony Languages. In: Proceedings of the 1st ECOOPWorkshop on Domain-Specific Program Development (DSPD), July 3rd, Nantes, France. (2006)
42. ATL Use Case – Compiling a new formal verification language to LOTOS (ISO 8807) URL: http://www.eclipse.org/m2m/atl/usecases/FIACRE2LOTOS/
43. F. Jouault, and F. Allilaire, An introduction to the ATL Virtual MachineV1.0 draft URL: http://www.eclipse.org/m2m/atl/doc/ATL_VM_Presentation_%5B1.0%5D.pdf
44. Extensible Markup Language (XML) 1.1 (Second Edition) URL: http://www.w3.org/TR/xml11/
45. Slonneger, K. and B. Kurtz. Formal Syntax and Semantics of Programming Languages. A Laboratory Based Approach, Addison-Wesley Publishing Company, 1995.
46. E. W. Dijkstra, GOTO Statement Considered Harmful, Letter of the Editor, Communications of the ACM, March 1968, pp. 147–148.
47. Cormen, Thomas H.; Leiserson, Charles E.; Rivest, Ronald L. (1990). Introduction to Algorithms, first edition, MIT Press and McGraw-Hill.
48. G. Varro, D. Varro and K. Friedl. Adaptive graph pattern matching for model transformations using model-sensitive search plans. In G. Karsai and G. Taentzer editors, Proc. of Int. Workshop on Graph and Model Transformation (GraMoT'05), volume 152 of ENTCS, pages 191–205, Tallinn, Estonia, September 2005.
49. D. Dotan, A. Kirshin, Debugging and Testing Behavioral UML Models, Proceedings of OOPSLA 2007, Montreal, Canada
50. ReDSeeDS. Requirements Driven Software Development System. European FP6 IST project. http://www.redseeds.eu/, 2007.