Model Transformation Approach Based on MOLA

Audris Kalnins, Edgars Celms[1], Agris Sostaks

University of Latvia, IMCS, 29 Raina boulevard, Riga, Latvia
{Audris.Kalnins, Edgars.Celms}@mii.lu.lv, agree@os.lv

Abstract. This paper provides a solution to the mandatory transformation example specified in MOLA – a graphical model transformation language developed at the University of Latvia. The solution is validated by executing it via the MOLA execution environment on several examples. In addition, a solution to one of the optional examples – determinization of a non-deterministic automaton is provided.

1 Introduction

The idea of model transformations as the main support for model driven software development is already gaining some maturity now. First and foremost, it appears in the area of model transformation languages. The emerging OMG standard model transformation language, QVT-Merge [1], most probably will reach its final shape at the end of this year. But while waiting for this, various independent model transformation languages gain their maturity too. Most of the languages use some sort of the pattern concept (to be matched in the source model) and rules controlling the application of patterns.

According to a very rough grouping, model transformation languages can be divided into textual and graphical languages. The QVT-Merge language fits into both groups since it has both textual and graphical form. Textual languages such as ATL[2], MTF[3], Tefkat[4], MT[5] and many other, though very different in details, typically use recursion as the main control structure.

Graphical transformation languages are significantly less in number. Besides QVT-Merge, Fujaba Story diagrams (SDM) [6] and GME-based GReAT [7] notation should be mentioned. The MOLA transformation language, which is the topic of this paper is namely in this category. In addition, graph transformation languages (such as AGG [8]), though originally built for different goal, actually have similar characteristics. It should be noted, that many characteristics of the graphical languages are somewhat similar too.

An unbiased comparison of qualities of transformation languages is not so easy to obtain, because there are so many different subjective viewpoints. Therefore this workshop, where very precisely defined requirements for a mandatory transformation example are given in its CFP [9], could provide the first such impartial comparison.

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

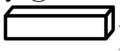
This paper presents a solution to this transformation task specified in MOLA – a graphical language developed at the University of Latvia, IMCS. Though the description of the same transformation in graphical languages is longer than in textual ones, authors consider the provided solution to be optimal from the readability and clarity point of view (though this is subjective too). The solution is validated on test models, by executing it via the MOLA tool. The paper describes how a problem specific modeling environment (for building test models) linked to the MOLA execution environment can be built using GMF – a generic modeling framework also developed at the University of Latvia (unfortunately, a name clash has occurred – an Eclipse project also named GMF [10] has been recently started).

Sections 2 and 3 provide a brief introduction to MOLA and its tools. The section 4 presents the solution of the mandatory transformation example, but section 5 – its validation via MOLA tool. Section 6 provides MOLA solution for one of the optional examples – the determinization of a non-deterministic automaton.

2 Brief Description of MOLA Language

The MOLA model transformation language has been developed at the University of Latvia, IMCS [11,12,13,14], the most complete description is given in [11]. MOLA is a graphical procedural transformation language. Its main distinguishing features are advanced graphical pattern definitions and control structures taken from the traditional structural programming. In this section we briefly remind the main concepts of MOLA. Later on in the examples sections example diagrams will be annotated by comments, which will allow easily to follow the notation.

Like most of the model transformation languages, MOLA is based on source and target metamodels, which describe the source and target models respectively. The used metamodeling language is EMOF [15] (with some slight restrictions). In MOLA source and target metamodels are combined in one class diagram, but packages may be used for structuring. The source and target metamodels may coincide. Special mapping associations linking the corresponding classes in source and target metamodels may be added to the metamodel. Their role is similar to relations in other transformation languages – for structuring the transformation and documenting the transformation traceability. If necessary, temporary classes and/or associations for storing intermediate data may be added.

The transformation itself is defined by one or more MOLA diagrams. A MOLA diagram is a sequence of graphical statements, linked by arrows. The most used statement in a MOLA diagram is the **FOREACH loop** – a bold-lined rectangle(▭). A loop has a **loop head** (a grey rounded rectangle - ▢), which contains the **loop variable** (a bolded element – e.g., **at:Attribute**) – a class, instances of which the loop has to iterate through. In addition, the loop head contains a **pattern**, which specifies namely which of the instances qualify for the given loop. A pattern is a metamodel fragment, but in instance notation – it contains **elements**, e.g., cl:Class, therefore classes may be repeated. Pattern links just correspond to metamodel associations. A pattern element may contain a **constraint** – an expression in OCL subset, which must be true for an instance to qualify. The semantics of loop is quite natural – the loop must be executed for all instances of the loop variable for which there exist instances of other pattern elements satisfying their constraints and linked by the specified links (pure existence semantics). Loops may be nested, the instance of the loop variable (and other elements) matched in the parent loop may be referenced in the nested loop by the **reference** notation – the element name prefixed by @ character. Besides the FOREACH loop, there is also the less used WHILE loop (), which is executed while there is at least one instance of loop variable for which the pattern matches, i.e., the same instance may be processed several times.

Another kind of graphical statement is the **rule** (a grey rounded rectangle too), which also contains a pattern but without loop variable. A rule typically contains actions – element or association building (red dotted lines) and deletion (dashed lines). A rule is executed once in its control path (if the pattern matches) or not at all – thus it plays the role of an if-statement too. A loop head may also contain actions. MOLA subprograms are invoked by the **call** statement (possibly with parameters), recursive calls are permitted. The parameters may be references to elements or primitive values.

One year experimental usage of MOLA, mostly in academic environment, has suggested few extensions with respect to the original definition of MOLA in [11]. Firstly, the use of **NOT** constraint in patterns has been clarified and extended. A MOLA element in a pattern may have the NOT constraint, e.g., {NOT} at:Attribute. The meaning is that the whole pattern matches, if there is **no** instance of the given class, which satisfies the local OCL constraint (if any) and has the specified links with the other ("positive") elements of the pattern. In addition, there may be a NOT constraint on a pattern link (no such link may exist between the matched instances) and a NOT-region – a rectangle containing several pattern elements (then there may be no properly linked match for the whole subpattern). Since the last two cases are not used in this paper, we present no more details of semantics for them.

Other extensions are related to control flows – now there are graphical equivalents for most of structured control constructs of modern programming languages. A rule may have two exits – one unmarked and the other one marked {**ELSE**} (any of them may be absent). If the rule pattern matches (and the rule actions are performed), the unmarked exit is taken. Otherwise, the ELSE exit is taken. If the required exit is absent, there is a default transition – if inside a loop body, then to the next iteration ("implicit continue"), if at the top level of a MOLA program, then it means the program end ("implicit return"). Thus a true if-then-else construct is provided. Branched control flows may merge again, but it is forbidden to build a "proper goto" – to branch backwards. Elements matched in a rule may be referenced only in its "positive path". In the context of a loop, some more options are available. A flow may reach the loop rectangle from inside – it means an "explicit continue". A flow may also cross the loop border – this is an "explicit break" (or "explicit return", if the target is an end symbol). In any case, no backward loops are permitted this way.

3 MOLA support tools

A MOLA tool supporting the MOLA transformation language has been built at the University of Latvia (see the first report on it in [16]). MOLA tool has two parts – the **Transformation Definition Environment** (**TDE**) and the **Transformation Execution Environment** (**TEE**). Both environments use a common **runtime repository**, which currently is a relational database. There transformations, metamodels and models all are stored.

The definition environment (TDE) is at the metamodel level (M2 in the MOF classification). Since MOLA is a graphical language, TDE is a set of graphical editors, built on the basis of GMF [17] – a generic metamodel based modeling framework, developed by University of Latvia, IMCS together with the Exigen company. It contains graphical editors for class diagrams (EMOF level) and MOLA diagrams. Both the source and target metamodels currently are shown in the same class diagram, together with possible mapping associations. A transformation is typically described by several MOLA diagrams, one of which is the main. In addition to editors, TDE contains the MOLA compiler which performs the syntax check and converts both the combined metamodel and MOLA diagrams from the GMF repository format to the MOLA runtime repository format. All MOLA examples in this paper have been taken from the MOLA TDE.

MOLA TEE is based on the MOLA Virtual Machine (VM) – an interpreter performing the model transformation, with instance data kept in the runtime repository (RDB). MOLA VM performs MOLA statements by converting them to SQL queries. It should be noted, that the most complicated element of MOLA – a pattern in a loop head or rule can be converted to a single SQL query. Thus the given implementation of MOLA is sufficiently simple (see more details in [16]). At the same time the experience with MOLA tool shows that it is also efficient enough – models with hundreds of instances may be transformed in seconds, if an appropriate RDB is used for the repository (currently – MSDE [19], the free version of MS SQL).

There are several ways how a complete MOLA TEE can be built because it must have close links with the supplier/consumer of models – a modeling environment. One of the ways is to use MOLA TEE as a plug-in for a modeling tool, with model data being exchanged in XMI format. It is sufficiently easy in the case of Eclipse and EMF [18] based tools. In [16] it is described in sufficient detail, how MOLA TEE can be used as a plug-in for the commercial IBM Rational modeling tool RSA. It should be noted that this approach requires at least one of the models (source or target) to be in standard UML 2.0.

Another approach, which is more relevant to the goals of this paper, is to use a generic modeling environment where an arbitrary graphical modeling notation can be supported. Since the GMF environment [17] fulfills these requirements, a reasonable solution is to link MOLA TEE to this environment. In GMF it is possible to define the graphical presentation of a domain model as a sort of transformation (though not very universal, see more in [17]), therefore for many modeling notations usable graphical editors can be defined without proper programming at all. In addition, Eclipse EMF style model tree browsers/editors, but more flexible ones (e.g., with several instances combined in one tree node), can be built very easily with GMF. Thus a readable visual representation of a model (source or target) can be obtained. This approach is adequate for domain specific notations, including non-UML ones, where frequently standard editing facilities simply are not available. Since the examples of this paper are in this category, namely such an approach is used. It should be noted that a somewhat similar approach is used for GReAT transformation language, combined with the generic GME modeling environment [7].

To apply the approach, two visual editors (diagrammatic or model tree based) must be defined in GMF for the source and target models respectively (if the source and target is different). They are based on the same metamodels which are used to define the model transformation in MOLA. Currently these metamodels must be ported manually to the GMF environment (GMF metamodels are in a slight variation of EMOF notation), but in the near future an automatic support will be provided. Then the editor definitions must be provided (e.g., which "domain metamodel pattern" maps to a presentation class, which pattern maps to a tree node etc., see more in [17]). The GMF-based MOLA TEE contains universal metamodel-controlled instance export and import components from/to GMF repository. The relevant MOLA transformation can be invoked directly from the GMF environment (MOLA VM is used as a GMF plug-in). The general schema of GMF based MOLA TEE is shown in Fig. 1.

The outlined here approach will be demonstrated in section 5 for the mandatory example – both tree-form and diagrammatic editors for source and target models will be shown. The convenient graphical facilities for building source models are used to test the correctness of defined MOLA transformations (see more in section 5).

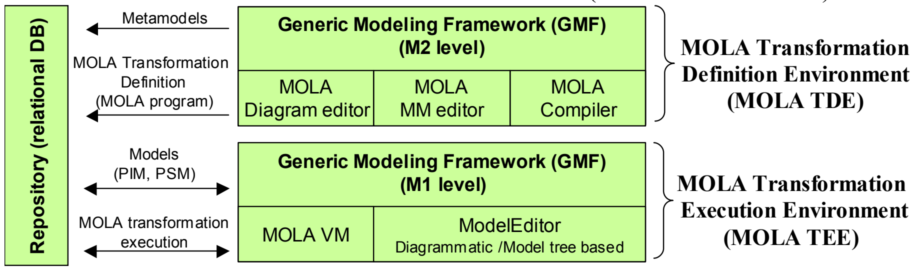

Repository (relational DB)
Metamodels
MOLA Transformation Definition (MOLA program)
Models (PIM, PSM)
MOLA transformation execution
Generic Modeling Framework (GMF) (M2 level)
MOLA Diagram editor
MOLA MM editor
MOLA Compiler
Generic Modeling Framework (GMF) (M1 level)
MOLA VM
ModelEditor Diagrammatic /Model tree based
MOLA Transformation Definition Environment (MOLA TDE)
MOLA Transformation Execution Environment (MOLA TEE)

Fig. 1. MOLA tool schema.

4 The mandatory example in MOLA

In this section we provide the MOLA solution for the mandatory model transformation example. The example is taken literally as specified in the workshop call for papers (CFP) [9]. However, the lately added to FAQ comment that subclasses of persistent classes do not add new elements to the primary key is not used – we permit primary attributes to be merged up to the persistent class. All diagrams of the proposed MOLA solution are shown in Fig. 2 – 12.

4.1 Metamodel of the example

Fig. 2 shows the metamodel of the example. In MOLA source and target metamodels (if different) must be combined in one class diagram. The upper region in Fig. 2 is the source metamodel (simplified UML) and the lower one is the target (simplified SQL). The regions are just graphical comments. All black associations are the original ones.

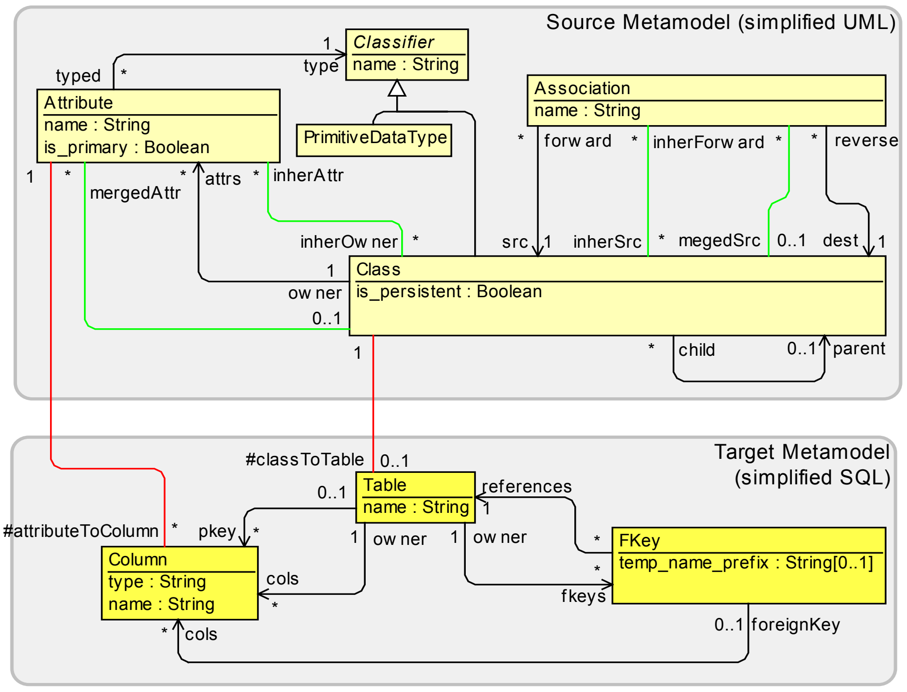


Fig. 2. The combined source and target metamodel in MOLA.

MOLA uses a slightly simplified EMOF syntax for metamodels. Association multiplicities must be explicit in MOLA, therefore the default ones have been added. Some role names for non-navigable ends also have been added (they are not mandatory for transformations, but ease the instance management in MOLA environment).

Associations in colors other than black have a special meaning in MOLA. The green ones are temporary – they are not present in the source model, but built by MOLA programs to store some intermediate relations. They are not also included in the resulting model. The red ones are the mapping associations, typically they link classes in source metamodel to target ones. They are built by MOLA programs, and their role is similar to relations, e.g., in QVT-Merge language – to transfer the results of high level transformations to subordinated ones and to facilitate the definition of inverse transformations (they are retained in the resulting model).

Fig. 2 contains two intermediate relations between `Class` and `Attribute` and between `Class` and `Association` – they are used to relate all (transitively) inherited elements (according to the standard UML semantics) and all "transitively merged-up" elements – as specified by the example requirements. See the section 4.2 and 4.3, how their use makes the transformations more readable. There are also two mapping associations – from `Class` to `Table` and from `Attribute` to `Column`. They serve as a "backbone" for defining the correspondence between the source and target models, e.g., it is very convenient to find easy, whether a table for a class has been built and namely which. A temporary attribute `temp_name_prefix` is also added to `Fkey` class (certainly, with multiplicity 0..1) – to store a temporary string. Actually, the role of all these additional metamodel elements is clearly visible when transformations themselves are discussed, and normally they are added "on the fly" during the transformation program design.

4.2 The main program of transformation

Now the transformation itself as a set of MOLA programs is being described. We start with the description of the main program, where the main principles of the proposed solution can be seen. Fig. 3 shows the main MOLA program.

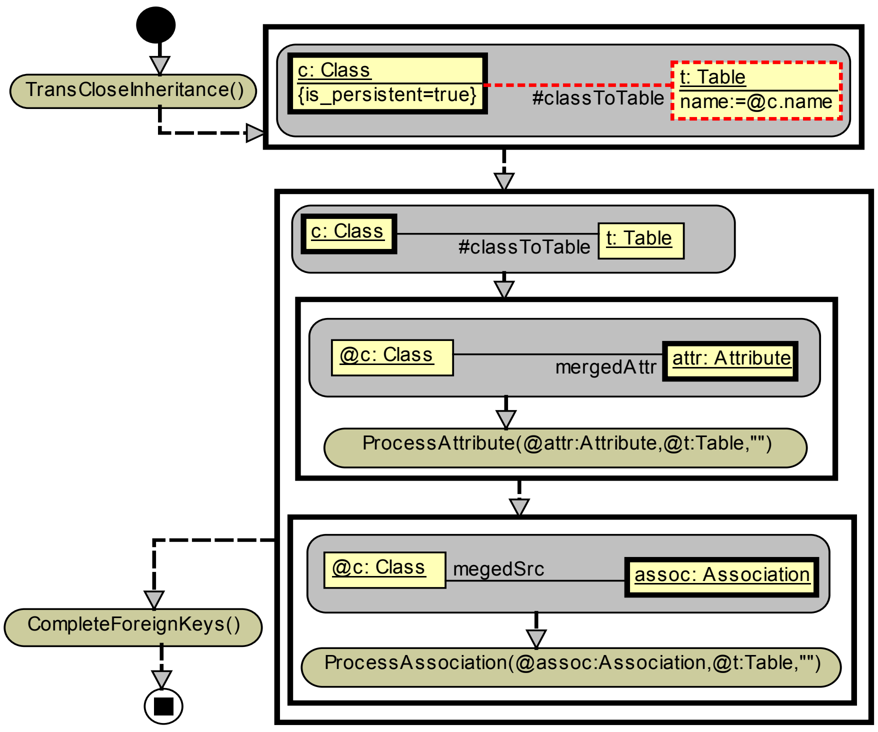


Fig. 3. The main MOLA program of the transformation.

We start with some comments on the transformation algorithm. Inheritance-related items 6 and 7 of the requirements specification [9], together with the specified precondition on inheritance (persistent classes are topmost parents), suggest that it would be convenient to process transitively the inheritance as the first step. More precisely, for non-persistent classes the traditional UML inheritance semantics must be applied, while for persistent classes the "transitive merge up" semantics must be used. The results of this transitive closure for a non-persistent class can be stored by means of temporary associations `inherAttr` (to all inherited attributes – including the direct ones) or `inherSrc` (to exiting associations), and `mergedAttr/mergedSrc` for persistent classes respectively. Namely this inheritance processing is performed in the subprogram `TransCloseInheritance`. In all the follow-up activities the appropriate temporary associations are used instead of the original ones (`attrs` or `src`). It should be noted that many "classical" UML tools (including Rose by IBM Rational) process the inheritance namely this way – you can always see all inherited attributes/associations of a class directly.

Now the comments on the MOLA program are given. We remind that MOLA control flows have some similarity to UML activity diagram – the same Start/End symbols are used. After the subprogram call for inheritance processing, the first FOREACH loop starts. This loop builds an equally named table for each persistent class – note the simple pattern consisting only of the loop variable (`c:Class`) itself (with the attribute constraint expressing the persistence). An assignment expression in MOLA can contain attributes from all elements in the same loop head (or rule), prefixed by the element name. In addition to the `Table` instance, an instance of the mapping association is also built.

The next loop actually again iterates over all persistent classes, but it has a different pattern – formally, loop over all `Class` instances which have a link to a `Table` instance (which is the same since such a link and instance have been built in the previous loop). The reason why we use the other pattern now is that we want to reference both the class (`@c:Class`) and its table (`@t:Table`) in the loop body. And in turn, we couldn't insert all the actions in this loop body into the first loop – we want to build also foreign keys (in the nested subprograms), which reference another table, and during the first loop it could happen that the target table is not yet built.

The body of this loop does the main job in the whole transformation. At the top level, it consists of two nested loops – for each merged up `Attribute` (i.e., having the temporary `mergedAttr` link to the current `Class` instance) invoke the `ProcessAttribute` subprogram with appropriate parameters and for each merged up exiting `Association` invoke the `ProcessAssociation`. Namely, the use of `mergedAttr` and `mergedSrc` links (built by the `TransCloseInheritance` subprogram) ensures the fulfilment of item 7 in the requirements specification – "the resultant table should contain the merged columns from all of its subclasses". The subprograms `ProcessAttribute` and `ProcessAssociation` are recursive – they invoke themselves (indirectly), thus implementing the recursive definition of names for target columns (and the recursive drill-down as such). The third (string) parameter of these subprograms is the currently cumulated up name prefix – for the top level invocation it is just empty string. The second parameter is the `Table` instance to which the generated `Column` (if any) or `FKey` must be attached. These subprograms actually implement rules 2, 3, 4, 5 of the requirements specification [9].

When the main job is done, there still remains something to do – foreign keys have no columns. The reason, why we couldn't fill them up "on the fly" again is – an FK must have columns corresponding to all columns of the referenced PK, and that PK could yet be undefined. So a separate subprogram `CompleteForeignKeys` completes the job.

4.3 The principal subprograms of the transformation

In this section we analyze the principal subprograms of the transformation: `ProcessAttribute`, `ProcessAssociation`, `BuildColumn`, `BuildForeignKey` and `ProcessNonPersistent`, which jointly perform the recursive drill-down of attributes and associations for a class. We start with the `ProcessAttribute` (Fig. 4). It has three parameters – the attribute to be processed, the table to which to add the result and the cumulated name prefix (string).

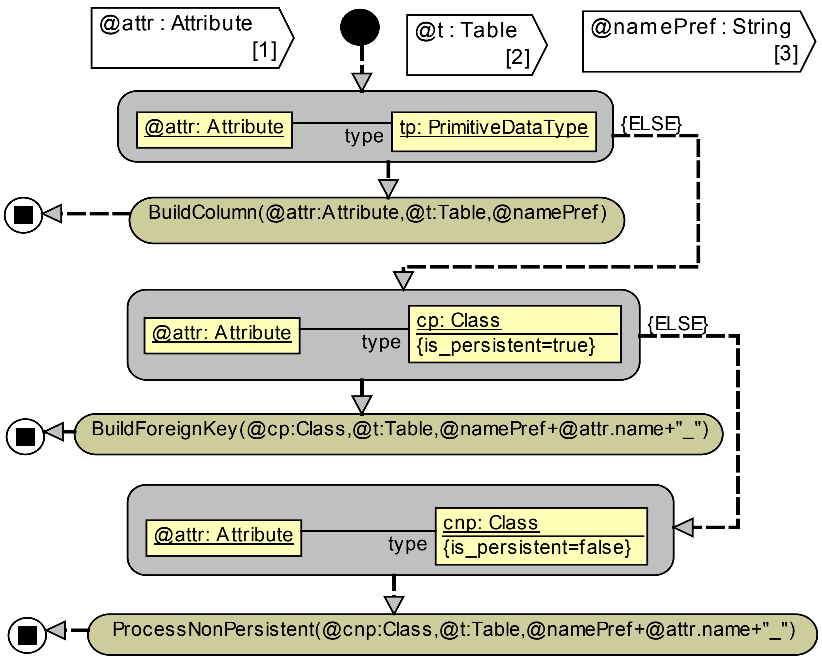
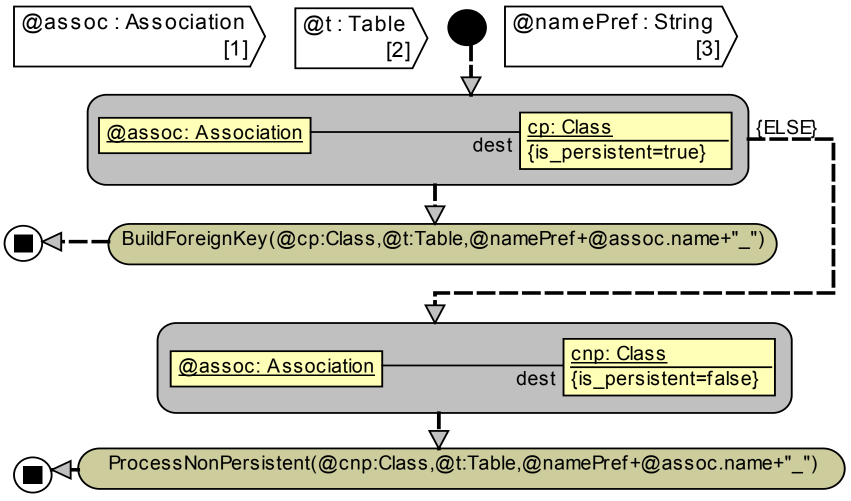


Fig. 4. ProcessAttribute subprogram.

This relatively straightforward subprogram implements items 3, 4 and 5 of the specification [9], by invoking the relevant subprograms. It contains no loops, but only rules. The first rule acts as a precondition for the item 3 – "an attribute has a primitive data type", therefore its unmarked (positive) exit leads to `BuildColumn` with appropriate parameters. If the pattern fails (the attribute's type is not primitive) the ELSE exit is taken. Similar graphical if-then-else constructs implement the other two cases (build foreign key if the type is a persistent class, invoke recursive processing of a non-persistent class). In both these cases the name prefix is prolonged – current attribute name added to it.

The `ProcessAssociation` subprogram (Fig. 5) is quite similar, except that only two cases are possible (there is no direct column generation from an association).



@assoc : Association [1]
@t : Table [2]
@namePref : String [3]
@assoc: Association
dest
cp: Class {is_persistent=true}
{ELSE}
BuildForeignKey(@cp:Class,@t:Table,@namePref+@assoc.name+"_")
@assoc: Association
dest
cnp: Class {is_persistent=false}
ProcessNonPersistent(@cnp:Class,@t:Table,@namePref+@assoc.name+"_")

Fig. 5. ProcessAssociation subprogram.

The `BuildColumn` (Fig. 6) subprogram is also quite simple, it contains only rules for building instances (the ELSE exit of the first rule is semantically impossible; if the pattern does not match for the second rule the default program end is used).

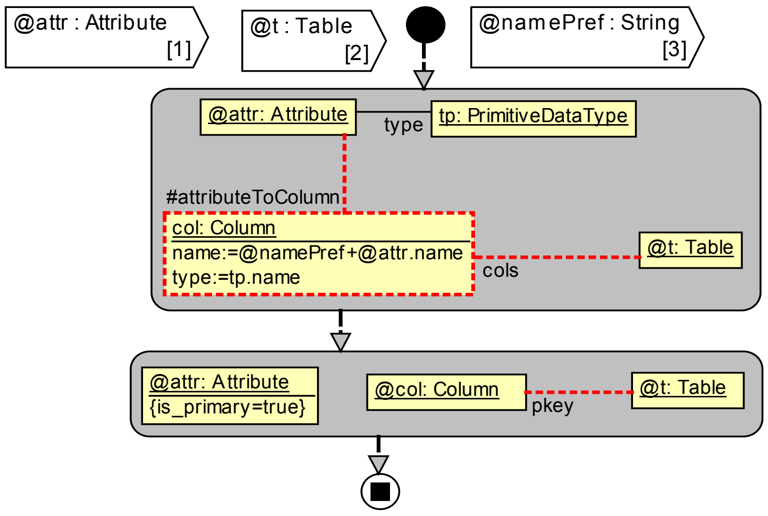


Fig. 6. BuildColumn subprogram.

In addition to building a column (using both the prefix and the current attribute), a primary attribute enforces the column to be included into the PK list.

Similarly, the `BuildForeignKey` subprogram (Fig. 7) contains a rule for building a foreign key, together with its reference to the target (note that the required `dt:Table` instance now exists for sure).

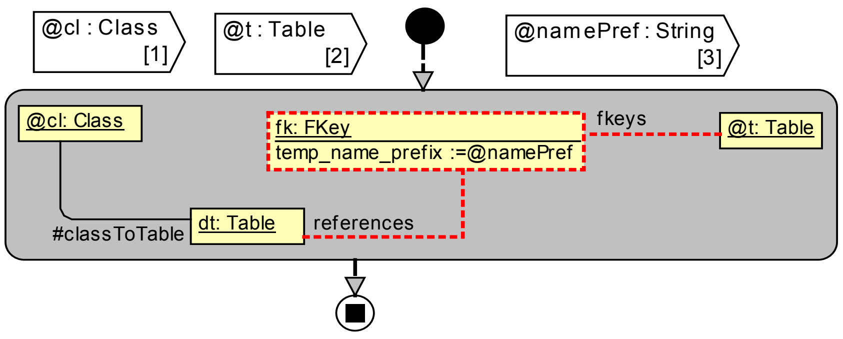


Fig. 7. BuildForeignKey subprogram.

The final subprogram in this set is `ProcessNonPersistent` (Fig.8), which completes the recursion (item 2 in the requirements [9]) for a non-persistent class (by processing all its inherited attributes and exiting associations).

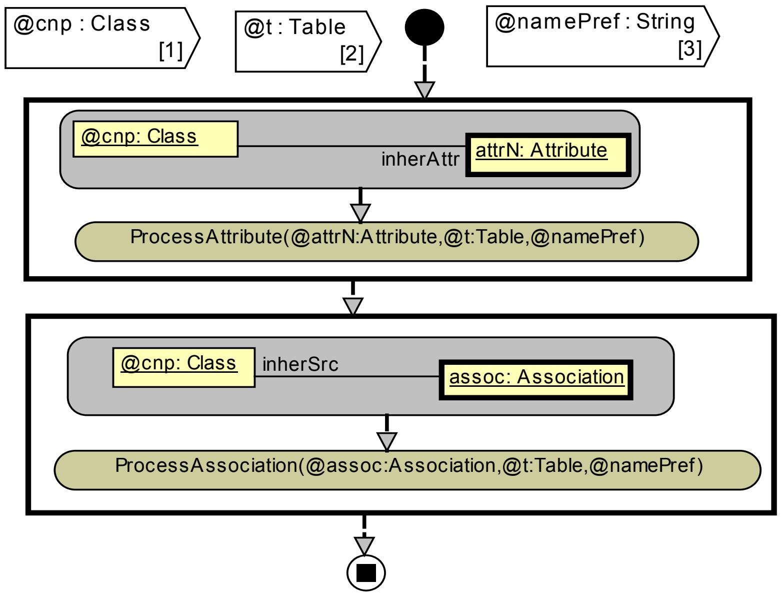


Fig. 8. ProcessNonPersistent subprogram.

4.4 Other subprograms of the transformation

We start with the `TransCloseInheritance` subprogram (Fig. 9), which was already mentioned in 4.2. Its role is extremely simple – for non-persistent classes perform ProcessInheritance, but for persistent – ProcessMerge (it was already explained in 4.2, why the specification implies such division). Both these subprograms process parent links recursively, therefore the "initial calls" to them have both parameters set to reference the current class (a class attribute is also an inherited attribute and so on). Alternatively, there could be one loop iterating over all classes, but with an if-then-else in the body.

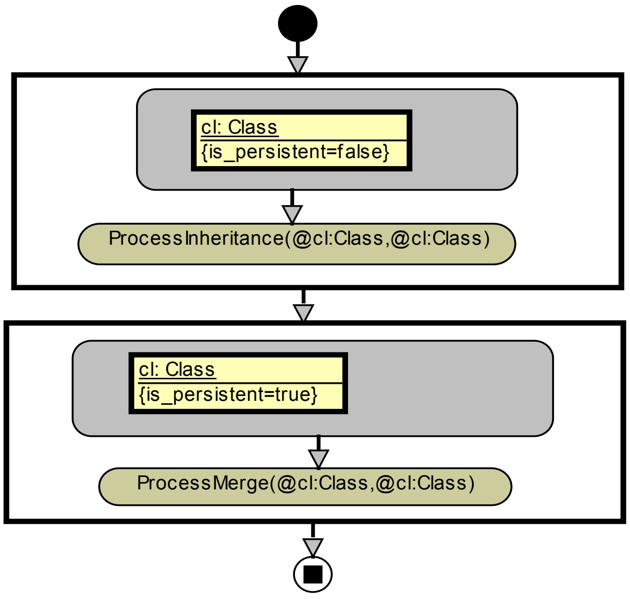

cl: Class
{is_persistent=false}
ProcessInheritance(@cl:Class,@cl:Class)
cl: Class
{is_persistent=true}
ProcessMerge(@cl:Class,@cl:Class)

Fig. 9. TransCloseInheritance subprogram.

Subprograms performing the real transitive closure – ProcessInheritance (Fig. 10) and ProcessMerge (Fig. 11) are very similar – the former iterates up via parent link, the latter – down. However, the difference in closure semantics implies some difference in programs. For inheritance, an attribute must not be inherited if there already is an (inherited) attribute with the same name. This fact is expressed by (the only one in the whole example) NOT constraint in the attr:Atribute pattern element – the instance of attrsup:Attribute doesn't match, if there is an instance of Attribute linked via inherAttr to the same Class and having a name equal to attrsup name.

Since the "up" multiplicity of parent is 0..1, there is no loop involving the recursive call, but just an if-then-else branch.

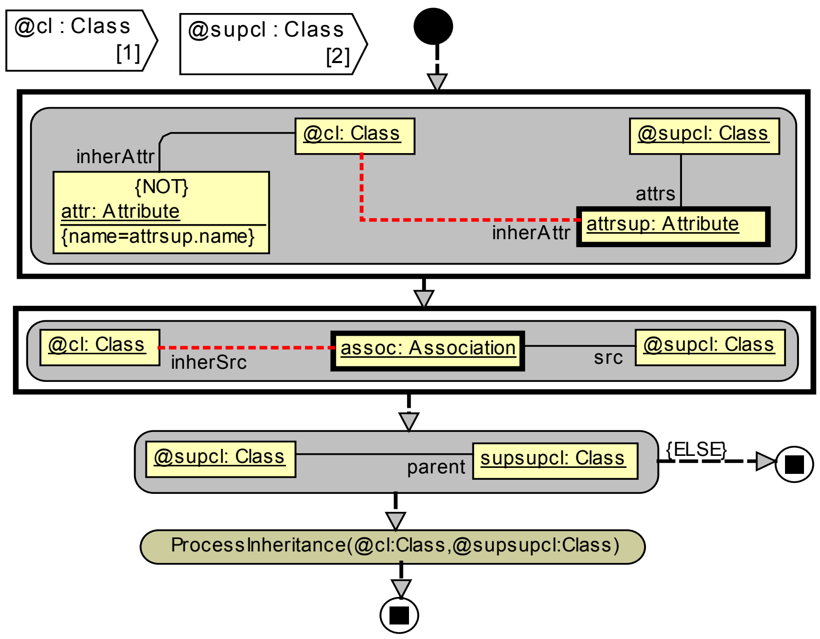


Fig. 10. ProcessInheritance subprogram.

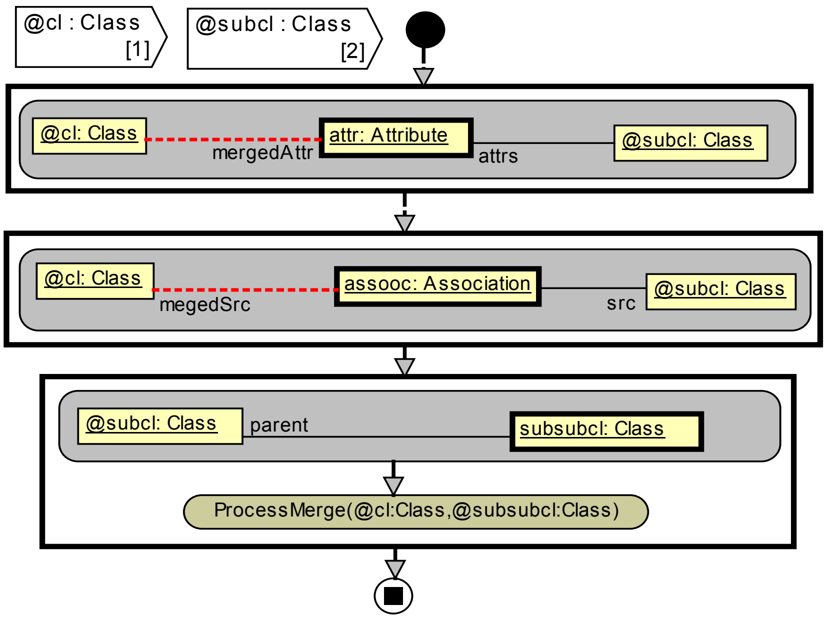


Fig. 11. ProcessMerge subprogram.

The ProcesMerge subprogram is simpler – there is no overriding in the merge definition. On the other hand, the "down" multiplicity of the parent link is *, therefore the recursive call is within a loop.

Finally, the CompleteForeignKeys subprogram does a simple job – it runs through all foreign keys and for each builds a set of columns (one for each column of the relevant primary key), using the name prefix, temporarily stored in FKey by the BuildForeignKey subprogram. Then the temporary attribute is cleared.

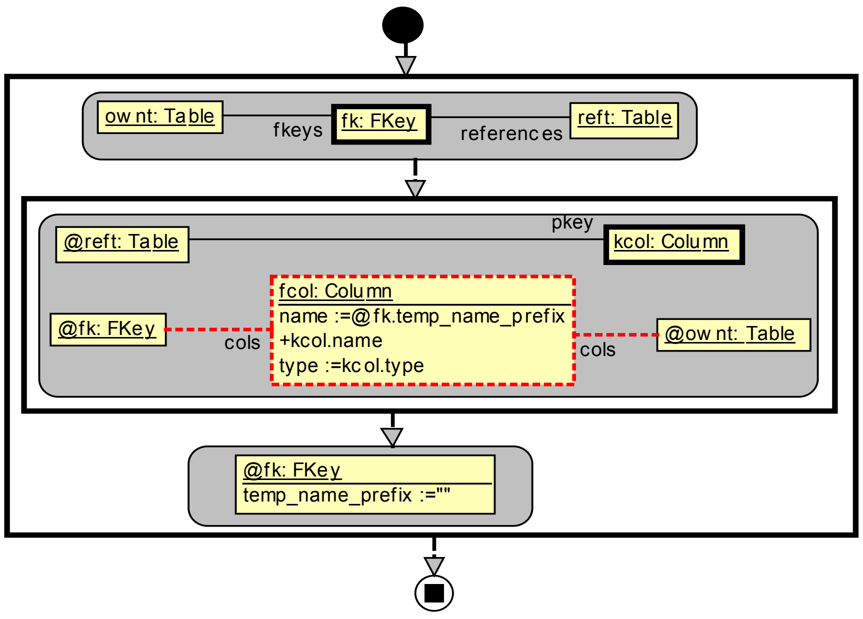


Fig. 12. CompleteForeignKeys subprogram.

This completes the mandatory example in MOLA.

4.5 Analysis of the example implementation

Certainly, the same way there is no one absolutely best implementation of OrderEntry subsystem for MySales, there is no absolutely best implementation of a transformation. Any analysis is subjective to a degree.

Authors themselves consider this implementation of the mandatory example a very nice application of MOLA. It seems to be very readable and clear (readability is subjective too!), no missing feature of the MOLA language has been found. It seems also that a certain optimum has been reached between the use of graphical patterns in loops and rules and purely programmatic constructs (sometimes one can replace another). It should be noted that only relatively recently the role of Recursive Call pattern in MOLA has been fully estimated. Though recursive calls have been permitted from the beginning, early examples of MOLA [11,14] all try to use only pure iteration for a very similar "drill-down" transformation, which makes the implementation more clumsy. It should be noted, that recursive calls could be used even more deeply – inherited (or merged) attributes or associations could be recursively found each time they are needed for the drill-down, but this was considered to be an overuse of recursion reducing the clarity. Namely therefore the temporary associations completely separating the processing of inheritance and drill-down were introduced. The recently introduced true if-then-else construct also makes the transformation behavior description clearer.

It is nearly impossible to compare textual transformation languages (textual QVT-Merge, ATL, MTL et al) to MOLA – simply each style has its proponents. The textual definitions are, certainly, much shorter but we consider them significantly less readable and consequently, more error prone. It should be noted that this example was intentionally completed without the use of MOLA TEE, using only manual "code inspections". Then it was subjected to proper testing via MOLA TEE, and only one error was found. Taking into account that published textual transformation examples contain bugs frequently enough it seems that more sizeable transformation definitions in MOLA pay off.

A more fair would be comparison to other graphical transformation languages (graphical QVT-Merge, FUJABA SDM, GReAT). Authors have not performed any direct comparisons due to unavailability of respective environments for these languages. Some indirect comparison could be made only to the graphical QVT-Merge, where the latest proposal document [1] contains a unidirectional transformation example, similar to this workshop example (but having some significant differences). An equivalent functionality seems to be definable more compactly in QVT-Merge than in MOLA. But since the only control structure in QVT-Merge governing rules actually is a recursive call (via the Where and When constructs), this notation seems to be much harder to read and understand. This fact was confirmed to a certain degree via experiments involving master students in CS.

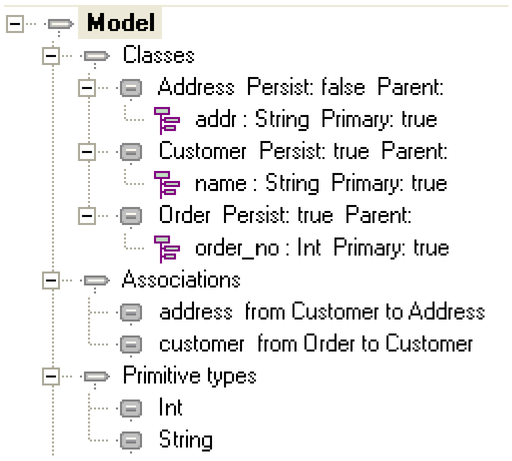
So it is up to users to decide which transformation definition facilities are better.

5 Use of MOLA TEE for the example

When a transformation is defined in MOLA (using the MOLA TDE) it can be compiled to check its syntax. However, a proper transformation validation can be done only using source model test examples within the MOLA TEE. Only the GMF-based version (see section 3) can be used for the example, since its metamodel is not part of the standard UML. As it was outlined in section 3, some visual facilities for building source models and viewing the transformed target models must be defined in GMF.

Initially the MOLA metamodel (combined) must be ported into the GMF metamodeling facility. In the case of the simple metamodel for the example (Fig.2) this could be done without any complexities (namely to facilitate the porting some role names were already added to the metamodel).

At first the simplest way of instance visualization – via customized model trees will be demonstrated. This approach is similar to the generated from a (meta) model tree and editor set in Eclipse EMF [18], but is significantly more flexible. For example, we can chose to represent a `Class` instance as a node, which shows the name, persistence and possible parent (the latter ones with keyword style separators to distinguish, which of the values are present). Then we can specify that child nodes of this node correspond to `Attribute` instances of the class (i.e., accessible via `attrs` link), each node showing the name, type and "primarity". Additional node type can be defined for associations, containing name plus source and target class names. Primitive types also must be shown as nodes. In addition, customized object dialogs can be defined for the main metaclasses (here `Class` and `Association`, with attributes as elements inside the `Class` dialog). GMF has also default object dialogs (like property editors in EMF), but they can be not so convenient for use. Fig. 13 shows the example tree in GMF (according to the abovementioned definitions), which corresponds to the input example – Fig. 3 from the workshop CFP. Parent is empty everywhere since there is no inheritance in this example (there is no way to remove the separator if the value is empty).

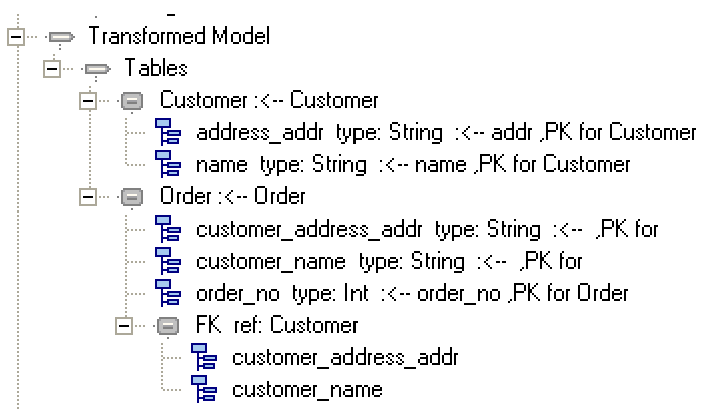


```
Model
  Classes
    Address  Persist: false  Parent:
      addr : String  Primary: true
    Customer  Persist: true  Parent:
      name : String  Primary: true
    Order  Persist: true  Parent:
      order_no : Int  Primary: true
  Associations
    address  from Customer to Address
    customer  from Order to Customer
  Primitive types
    Int
    String
```

Fig. 13. Input example in GMF.

Similarly, tree nodes for the target model must be defined. Here the sole top level node should be `Table`, showing the name. It has two types of children – columns and foreign keys. `Column` nodes display name, type and whether part of PK. For both table and column nodes it can be shown from which source model elements they were generated (via the mapping associations), visually separated by ":<-" string – this is an element of explicit traceability. For foreign key nodes the referenced table may be shown, with included columns as children nodes.

Now it remains to export the instance data (source model) from GMF repository to MOLA runtime repository, start the selected transformation and import back the transformed model to the GMF repository. All these actions have been added as standard services to GMF. Fig. 14 shows what was obtained from the source model in Fig. 13.



```
Transformed Model
    Tables
        Customer :<-- Customer
            address_addr  type: String  :<-- addr ,PK for Customer
            name  type: String  :<-- name ,PK for Customer
        Order :<-- Order
            customer_address_addr  type: String  :<--  ,PK for
            customer_name  type: String  :<--  ,PK for
            order_no  type: Int  :<-- order_no ,PK for Order
            FK  ref: Customer
                customer_address_addr
                customer_name
```

Fig. 14. Transformation results in GMF (obtained from data in Fig. 13).

It can be easily verified, that the results do comply with the Fig. 4 in the CFP [9] (columns which are not PK show the empty "`,PK for `" separator, columns which are not direct maps of source model attributes, show empty ":<-" string). Namely this way the sole transformation error was detected – the underscore symbol in names initially was placed wrongly.

Certainly, to validate the defined transformation to a certain degree, much more test examples would be needed, e.g., inheritance is not tested at all. Larger examples can be built via this visualization for sure, but we want to demonstrate briefly the other possibility in GMF – present models as custom diagrams. Both the source and target metamodels of the example satisfy "GMF diagramming" requirements, only a special metaclass (representing a "domain diagram") must be added to each. This requires also one "technical subprogram" to be added to the transformation end – the domain diagram instance must also be built automatically. All these "scaffolding activities" in no way affect the original models or transformation. Fig. 15 shows the source model represented as a slightly non-standard class diagram – according to the assumed metamodel. Additional metaattributes (`is_persistent`, `is_primary`) are displayed as tagged values. Definition of this diagram-style presentation is more complicated, it must be specified, e.g., that `Class` maps to an auxiliary metamodel element `ClassSymbol`, which in turn has a rectangular shape and contains three text compartments one of which (for attributes) is a list compartment. Thus a sort of model transformation (domain to presentation) actually is defined in GMF, more details can be found in [17]. The definition result is a "normal" graphical editor for this variation of class diagrams, with standard facilities to be found in diagramming tools. The example in Fig. 15 (built via this editor) is a slightly adapted advanced case study (Fig. 5 in CFP), which was not meant to be used for the strict transformation rules of the mandatory example (therefore the results will be slightly unexpected). The adaptation had to be done to satisfy the preconditions on class models. Nevertheless it is a good test for the transformation – many "use cases" can be observed on it.

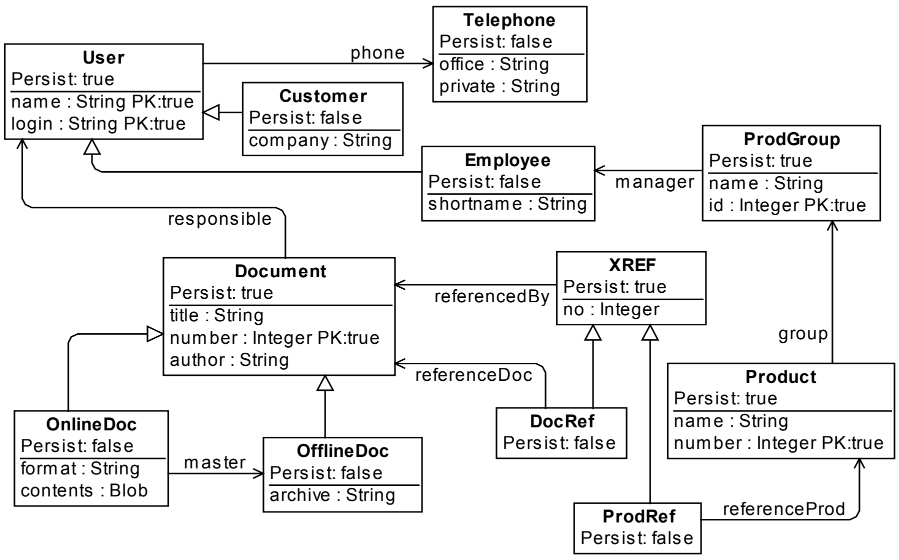


Fig. 15. Complicated input example as a GMF class diagram.

Transformation results frequently also can be displayed as a diagram, in this case an "RDBdiagram" (somewhat similar to Fig. 6 in CFP [9]) is defined. Tables are presented as rectangles showing columns in a list compartment, separate compartments present members of PK and the reference for each of the FKs. The columns included in an FK are shown as a list attached to the line representing this FK (unfortunately, FKs have no names in this transformation). When the transformation is run on the example and the transformed instances imported back into GMF, the diagram itself is displayed automatically via the GMF auto-layout facility.

Fig. 16 shows the result of transformation when applied to the model in Fig. 15. It can be noted that only persistent classes result into tables, but inheritance and drill-down generate a lot of new columns – according to the transformation specification. No transformation program errors were detected in this test, which can be considered as an exhaustive enough (though authors have not tried to apply any formal testing completeness criteria). The only conclusion is that in practice more sophisticated transformations from class models to RDB should be used.

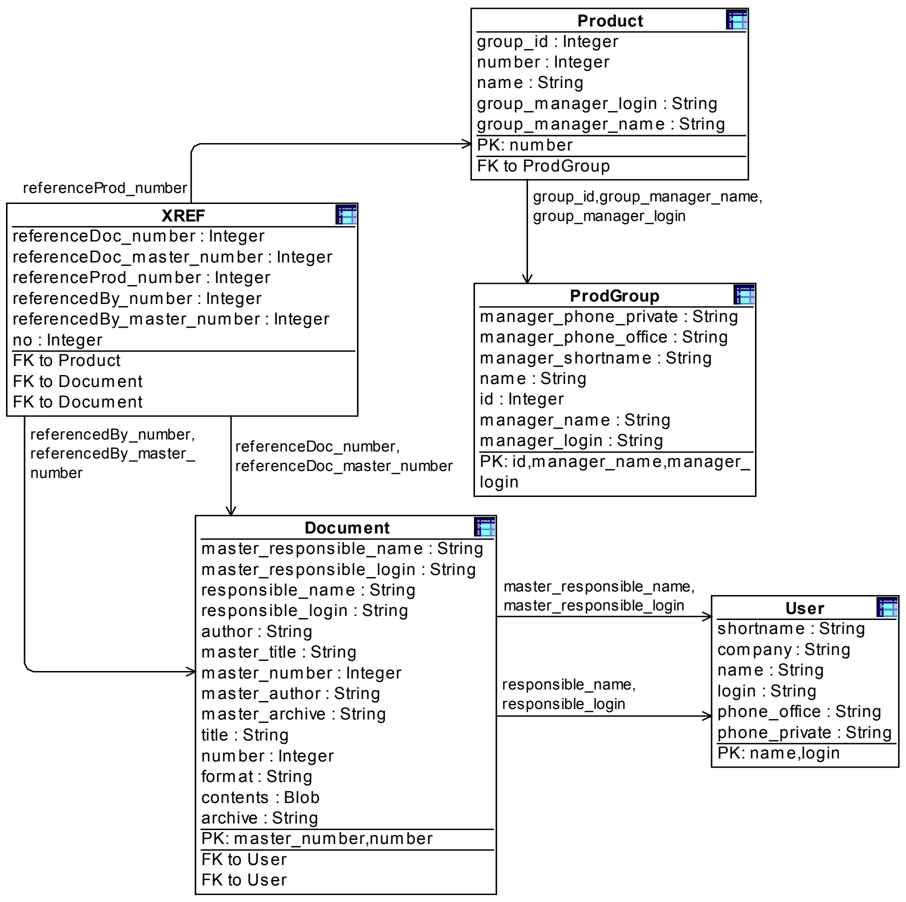


Fig. 16. The transformation result as an RDB diagram.

6 The optional example – nondeterministic FSM to deterministic

In this section we briefly describe one more example – the transformation of a nondeterministic automaton (FSM) to a deterministic one. Automata are assumed to be language recognizers (no output), a nondeterministic one can have many initial states and many final states, a string belongs to the language if there is a path from an initial to a final state marked by this string (empty or lambda moves are not included). Thus a simplest possible definition is assumed. For deterministic automaton the standard language recognizer definition is used. Automata are defined as sets consisting of state, event (=input alphabet element) and transition instances. The classical determinization algorithm is implemented – explore the state powerset (set of all subsets) space, by starting from the "initial set" and trying to expand the reachable set of statesets by applying transitions for all possible events and analyzing whether a new stateset has been reached by the given event (or it is a copy of existing one). When nothing more can be reached, the reached powerset elements are coded as new states of the deterministic FSM, and new transitions are defined accordingly, as well as the initial state and final states.

Fig. 17 shows the metamodel (source = target), with the `StateSet` class used during the algorithm run. Fig. 18 – 22 show the main MOLA program and subprograms implementing the abovementioned algorithm. Some of the subprograms use additional MOLA elements not used in the main example.

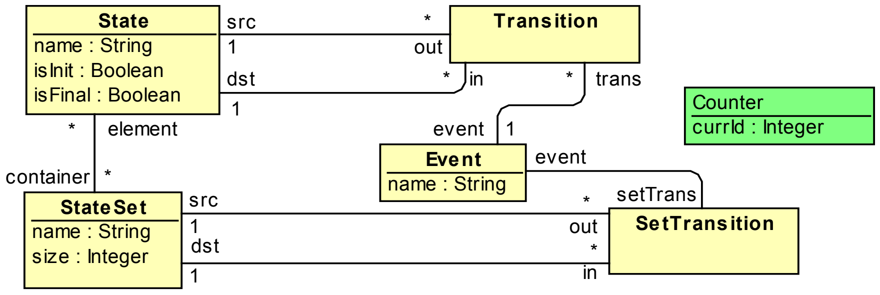


Fig. 17. Metamodel of automatons.

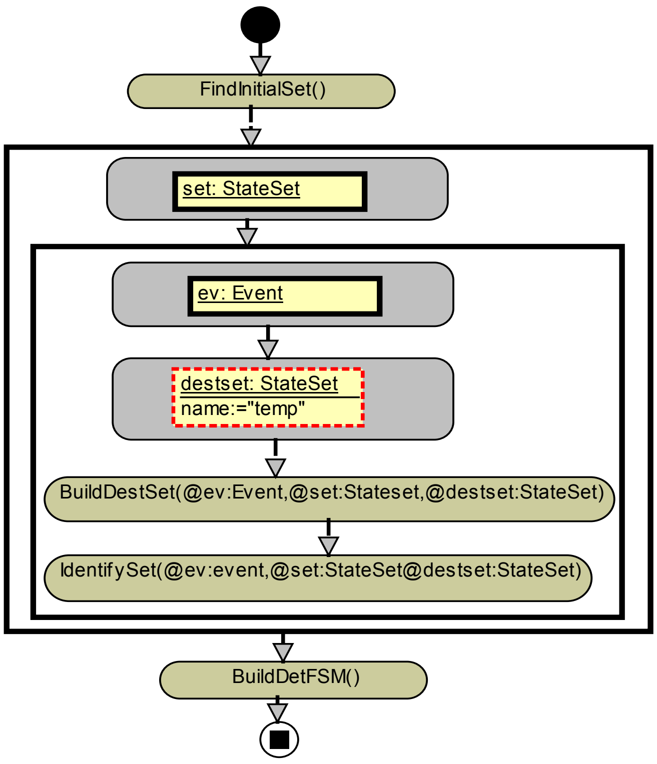

FindInitialSet()
set: StateSet
ev: Event
destset: StateSet
name:="temp"
BuildDestSet(@ev:Event,@set:Stateset,@destset:StateSet)
IdentifySet(@ev:event,@set:StateSet@destset:StateSet)
BuildDetFSM()

Fig. 18. Main MOLA program for the determinization.

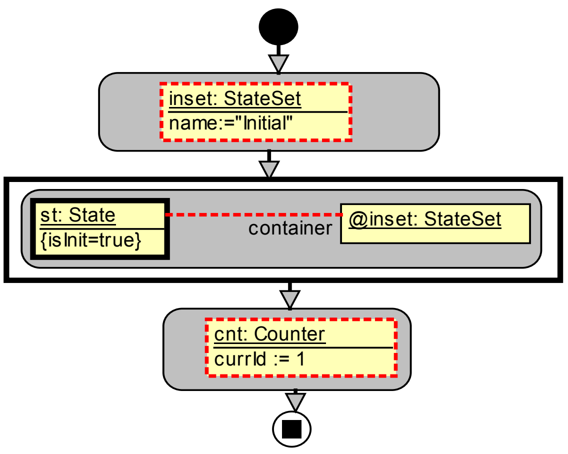


Fig. 19. Subprogram FindInitialSet.

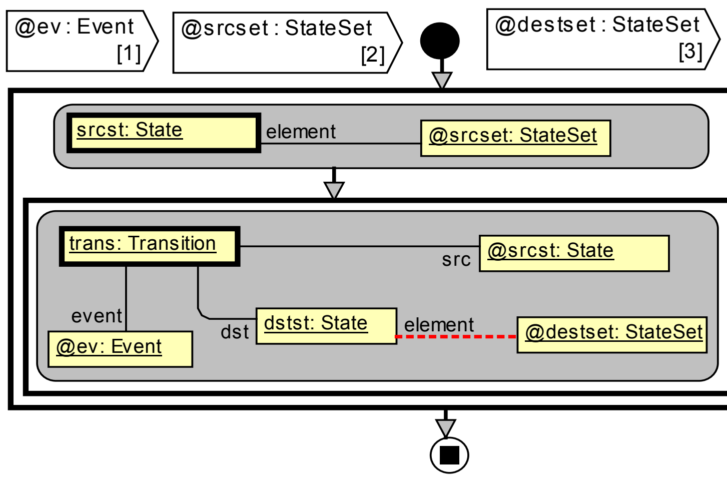


Fig. 20. Subprogram BuildDestSet.

The next subprogram `IdentifySet` uses more complicated OCL expressions in constraints – subexpressions of the form `element_name.role_name`, which denote an instance set (if the multiplicity is *) and elementary OCL operations on sets (here – the set equality). Two special control constructs – explicit *continue* (flow to the loop border) and *return* (flow to end symbol) are used in the first loop.

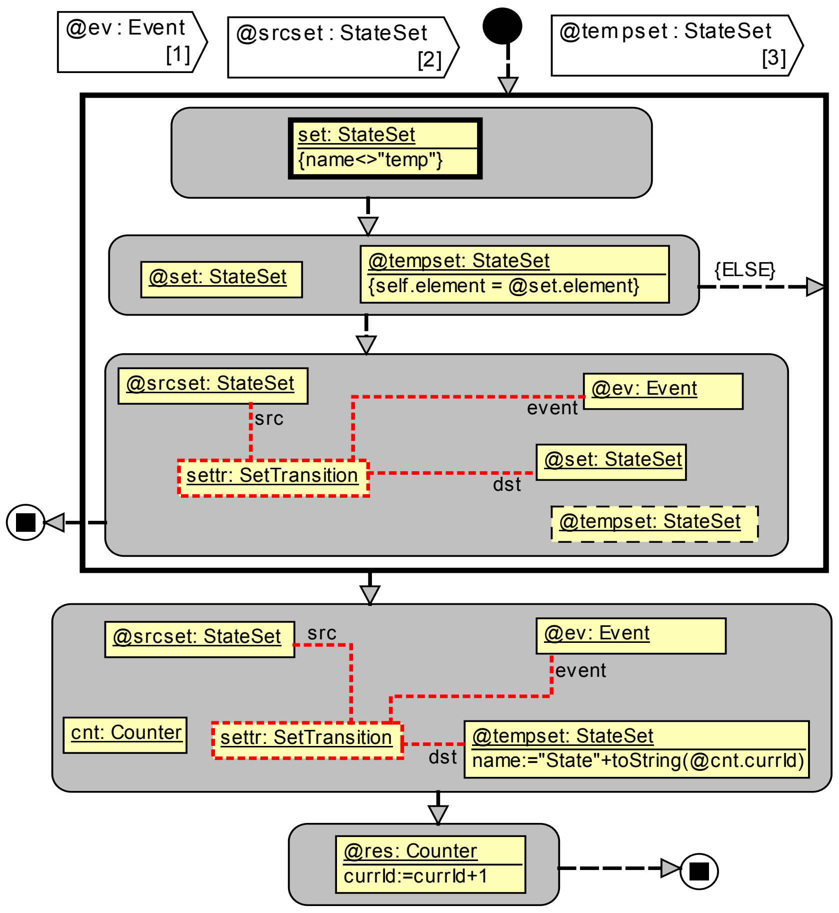


Fig. 21. Subprogram IdentifySet.

The subprogram `BuildDetFSM` also uses OCL set operations in constraints – `notEmpty` and the quantifier `exists`.

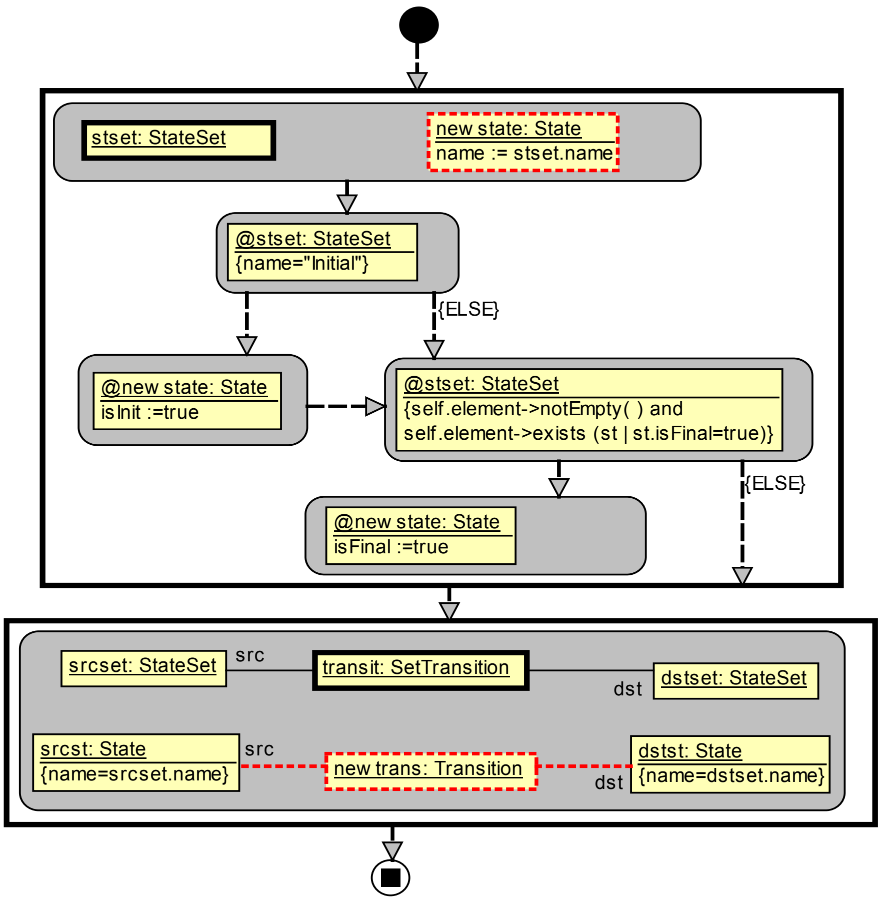


Fig. 22. Subprogram BuildDetFSM.

Authors consider this example also a right balance between the textual and graphical style of transformation specifications. Namely to make the example maximally readable, explicit sets defined via associations from an instance and OCL set operations are used in patterns. Certainly, the example could be specified "100% graphically", using nested loops, but this seems not to be the best choice. The extended use of OCL is not yet implemented in MOLA VM, therefore the example has not been validated in MOLA TEE.

6 Conclusions

The description of the implementation of the mandatory transformation example in MOLA (in section 4) provides, according to the authors' view, a good style of a graphical transformation definition. The increased size of the solution is compensated by a better readability, which in turn ensures that less effort for the transformation development is required and it is less error prone. The latter fact to a certain degree has been confirmed by a controlled experiment – developing the transformation and only then testing it. The MOLA execution environment, based on GMF, also occurred to be very fit for building test models and executing the transformation on them. Thus the practical transformation validation, using the facilities to build/view models in a graphical form, appeared to be completely satisfactory. The optional example, in turn, demonstrates that graphical pattern definition facilities should not be overused – they can naturally be combined with the use of OCL in MOLA element constraints.

Certainly, there are more problems in practical transformation development. First, the transformation composition is more the tool than language issue – in MOLA environment, for sure, it is possible to apply consecutively several transformations while the model data are in the runtime repository (certainly, if the metamodels are consistent to this). Bidirectional or incremental transformations certainly don't come for free in MOLA because it is an outspokenly procedural language. Reverse or incremental transformations must be developed specially with the goal in mind, but some experiments show that MOLA pattern features are powerful enough to implement the relevant source-target relations easily. It is especially easy if the mapping associations are used adequately for the direct transformation, e.g., it can be easily detected in the example that a new `Table` has been added to the target model which has no link to its `Class`. To sum up, the MOLA language seems to meet all the main transformation technology requirements, certainly, the existing MOLA tool will be extended to meet all the aspects of practical usability.

References

[1] QVT-Merge. URL: http://www.omg.org/docs/ad/05-03-02.pdf

[2] ATL. URL: http://www.sciences.univ-nantes.fr/lina/atl/

[3] MTF. URL: http://www.alphaworks.ibm.com/tech/mtf

[4] Tefkat. URL: http://www.dstc.edu.au/Research/Projects/Pegamento/tefkat/

[5] Tratt L. "The MT model transformation language". Technical report TR-05-02, Department of Computer Science, King's College London, May 2005.

[6] Fujaba User Documentation.
URL: http://wwwcs.uni-paderborn.de/cs/fujaba/documents/user/manuals/FujabaDoc.pdf

[7] 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

[8] The Attributed Graph Grammar System (AGG). URL: http://tfs.cs.tu-berlin.de/agg/

[9] Model Transformations in Practice Workshop. Call for papers (CFP).
URL: http://sosym.dcs.kcl.ac.uk/events/mtip/long_cfp.pdf

[10] Graphical Modeling Framework (GMF, Eclipse technology subproject).
URL: http://www.eclipse.org/gmf/

[11] A. Kalnins, J. Barzdins, E. Celms. "Model Transformation Language MOLA". Proceedings of MDAFA 2004 (Model-Driven Architecture: Foundations and Applications 2004), Linkoeping, Sweden, June 10-11, 2004. pp.14-28.

[12] Kalnins A., Barzdins J., Celms E. "Model Transformation Language MOLA: Extended Patterns". Selected papers from the 6th International Baltic Conference DB&IS'2004, IOS Press, FAIA vol. 118, 2005, pp. 169-184.

[13] A. Kalnins, J. Barzdins, E. Celms. “Basics of Model Transformation Language MOLA”. ECOOP 2004 (Workshop on Model Transformation and execution in the context of MDA), Oslo, Norway, June 14-18, 2004. URL: http://heim.ifi.uio.no/~janoa/wmdd2004/papers/
[14] A. Kalnins, J. Barzdins, E. Celms. “MOLA Language: Methodology Sketch”. Proceedings of EWMDA-2, Canterbury, England, 2004. pp.194-203.
[15] MOF 2.0 Core Final Adopted Specification.
URL: http://www.omg.org/docs/ptc/03-10-04.pdf
[16] A. Kalnins, E. Celms, A. Sostaks. “Tool support for MOLA”. (Preliminary version). GPCE'05. Paper accepted to the workshop on Graph and Model Transformation (GraMoT), Tallinn, Estonia, September 2005
[17] E. Celms, 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.
[18] UML 2.0 Eclipse EMF. URL: http://www.eclipse.org/uml2/
[19] Microsoft SQL Server 2000 Desktop Engine (MSDE 2000).
URL: http://www.microsoft.com/sql/msde/default.asp