Hello World with MOLA

Elina Kalnina, Audris Kalnins, Agris Sostaks, Janis Iraids, Edgars Celms

University of Latvia, IMCS, Raina bulvaris 29, LV-1459 Riga, Latvia
Elina.Kalnina@lumi.lv, Audris.Kalnins@lumi.lv, Agris.Sostaks@lumi.lv, Janis
Iraids@lumi.lv, Edgars.Celms@lumi.lv

Abstract. This paper describes the solution of Hello World transformations in MOLA transformation language. Transformations implementing the task are relatively straightforward and easily inferable from the task specification. The required additional steps related to model import and export are also described.

1 Introduction

In this paper we describe the solution to Hello World case [5] for TTC 2011\(^1\) contest, implemented in MOLA model transformation language. The core task and all optional extensions are implemented.

The “Hello World” task can be implemented in MOLA in a very straightforward way. We describe in the paper the basic principles of the solution. Before that, the situation with metamodels is described in some details, namely, how the metamodels are imported and extended and what implications this creates for source model import and target model export. The whole solution process using MOLA is briefly described as well.

2 MOLA Environment

MOLA [4] is a graphical transformation language developed at the University of Latvia. It is based on traditional concepts among transformation languages: pattern matching and rules defining how the matched pattern elements should be transformed. The formal description of MOLA and also MOLA tool can be downloaded in [1].

A MOLA program transforms an instance of a source metamodel (defined in the MOLA metamodelling language – MOLA MOF, close to EMOF) into an instance of a target metamodel.

Rule contains a declarative pattern that specifies instances of which classes must be selected and how they must be linked. Pattern in a rule is matched only once. The action part of a rule specifies which matched instances must be changed and what new instances must be created. The instances to be included in the search or to be created are specified using class elements in the MOLA rule. The traditional UML

\(^1\) http://planet-research20.org/ccc2011/
instance notation (instance_name:class_name) is used to identify a particular class element. Class elements may contain constraints and attribute assignments defined using simple OCL-like expressions. Additionally, the rule contains association links between class elements. Class elements matched in one rule may be referenced in another one using the reference element (prefixed with the "@" symbol).

In order to iterate through a set of the instances MOLA provides the foreach loop statement. The loophead is a special kind of a rule that is used to specify the set of instances to be iterated over. The pattern of the loophead is given using the same pattern mechanism used by an ordinary rule, but with an additional important construct. It is the loop variable—the class element that determines the execution of the loop. The foreach loop is executed for each distinct instance that corresponds to the loop variable and satisfies the constraints of the pattern. In fact, the loop variable plays the same role as an iterator in classical programming languages. The execution order in MOLA is specified in a way similar to UML activity diagrams.

MOLA has an Eclipse-based graphical development environment (MOLA tool [1]), incorporating all the required development support. A transformation in MOLA is compiled via the low-level transformation language L3 [2] into an executable Java code which can be run against a runtime repository containing the source model. For this case study Eclipse EMF is used as such a runtime repository, but some other repositories can be used as well (e.g. JGraLab [3]).

### 3 General Principles of Hello World Solution with MOLA

The transformation development in MOLA starts with the development of metamodels. The MOLA tool has a facility for importing existing metamodels, in particular, in EMF (Ecore) format. Though MOLA metamodelling language (MOLA MOF) is very close to EMOF, and consequently Ecore, there are some issues to be solved. The current version of MOLA requires all metamodel associations to be navigable both ways (this permits to perform an efficient pattern matching using simple matching algorithms). Since a typical Ecore metamodel has many associations navigable one way, the import facility has to extend the metamodel. Another issue is the variable coding of references to primitive data types.

Metamodel import facilities in MOLA are able to perform all these adjustments automatically. This way the provided metamodels were imported into MOLA tool.

In some tasks for transformation development in MOLA some additional metamodel elements are required, for example, in migration tasks to store relations between source and target models. These metamodel elements have to be added manually. In the migration tasks, these are associations between node classes in different graph encodings. Then the transformation itself (MOLA procedures) can be developed. The key features of transformations are described in the next section. The development ends with MOLA compilation.

Since the metamodels have been modified during import, the original source model does not conform directly to the metamodel in the repository, mainly due to added association navigability. Therefore a source model import facility is required. MOLA execution environment (MOLA runner) includes a generic model import facility,
which automatically adjusts the imported model to the modified metamodel. Now the migration transformation can be run on the model. Similarly, a generic export facility automatically strips all elements of the transformed model which does not correspond to the original target metamodel. Thus a transformation result is obtained which directly conforms to the target metamodel. The transformation user is not aware of these generic import and export facilities, he directly sees the selected source model transformed.

4 MOLA Transformations for Hello World

The Hello world case consists of several very simple tasks. The first group of tasks are “Greeting” transformations. In these transformations the MOLA pattern used is very similar to the corresponding instance diagram given in the task specification. Solution of the second subtask is presented in Fig. 1.

![Fig. 1. Solution of extended greeting creation task.](image)

Another group of tasks is various kind instance counting in a graph. To solve these tasks an integer variable “sk” (the counter) and a MOLA foreach loop is used. In most cases the loophead pattern directly specifies the set of instances to be counted. Each loop iteration increases the instance count by one. After the loop the result instance is created and the counter value is placed into it. An example of such type is given on the right side of Fig. 2. In this case looping edges are counted. As it has not been defined in the task specification whether there is only one graph in a model, we
assume that there can be many, therefore we process each graph separately. The graph processing is given on the left side of the Fig. 2. A similar graph processing is done for all tasks where the phrase “in a graph” is used.

The only counting task processed differently is the circle counting. In this case one loop variable is not sufficient. As in MOLA only one loop variable can be used, multiple loops or temporary metamodel elements are required to solve this task. For details see the appendix.

The next task is to reverse edges in a graph. We propose to solve this task by adding new reversed edge and removing the existing edge. This transformation is shown in Fig. 3.

Fig. 3. Transformation reversing edges in a graph.

For model migration tasks additional temporary metamodel elements typically are required in MOLA. In this case it is sufficient to have an association between nodes in different graph encodings. After that the transformation is quite straightforward. At first we migrate nodes and then edges. Fig. 4. shows the transformation.

The last group of tasks is deletion tasks. They are straightforward – the element to be deleted is found and then deleted.

We have also solved all optional tasks. The complete set of transformation procedures with short description is given in the appendix.

5 Conclusions

This case study has been quite appropriate to be implemented in MOLA. It confirms the assertion that simple tasks can be solved in a straightforward and easy readable way in MOLA. In most cases the basic part of the task is performed by one rule (or loophead). The used patterns have a natural form similar to what would appear in typical graph transformation languages. However, a natural solution of the circle counting task would require an even more powerful pattern feature in MOLA – a loophead with three loop variables.
Fig. 4. Model migration transformation. Migrates graph from encoding graph1 to encoding graph2.

Acknowledgments
This work has been partially supported by the European Social Fund within the project «Support for Doctoral Studies at University of Latvia».

References
Appendix A: Transformation sources.
In this appendix the complete set of transformation sources will be described.

The Hello World task consists of several small tasks. Therefore we have created one project with a package (compilation unit) for each sub-task. Content of the transformation project can be seen in Fig. 1. Each package contains a solution of one sub-task. "pX_Y" means that the sub-task was defined in section X.Y of the case description. In some sections several sub tasks have been defined. In this case the third digit is used to describe the sub-task order in section. Optional sub-tasks are marked using "_o". For one task two solutions are provided, the alternative solution is marked using "_a".

Fig. 1. Structure of Hello World transformation project.

Greeting transformations are given in Fig. 2, Fig. 3 and Fig. 4.

Fig. 2. Transformation creating constant Greeting instance.

Fig. 3. Transformation creating constant Greeting instance with references.
The next group of tasks in the task specification is instance counting tasks. A transformation counting nodes in a graph is given in Fig. 5. A transformation counting looping edges is given in a Fig. 6. Transformation counting isolated edges is given in Fig. 7. In MOLA the counting is implemented using an integer counter and a foreach loop where the counter is increased. For all these tasks it was required to count elements in a graph. However it was not defined whether the model contains only one graph or multiple. For transformations to work properly if there is more than one graph in a model we provide the graph to be processed as a parameter. Therefore to call these logical transformations we use another procedure where we iterate through all graphs in a model and call the logical transformation. Example of such transformation is given on the left side of Fig. 5. (The only thing that changes is the called procedure.) If there is always only one graph in a model this step could be omitted. The same could be said also about transformations in Fig. 11. -16.
The next task was to count circles consisting of three nodes. The solution of this task is different from the previous one because we want to find all different circles. In this case one loop variable is not sufficient and as a result, several loops are required.
In the task specification it was not clearly stated whether graphs or multi-graphs should be considered (i.e., is it possible to have multiple edges between two nodes.) As the provided metamodel supports multi-graphs and graphs are a subclass of multi-graphs, we decided to build our solution so that multi-graphs are supported. As we support multi-graphs, if there is a circle “n1;n2;n3” and two edges between “n1” and
“n2”, then there will be two circles “n1;n2;n3” (and 2*“n2;n3;n1” + 2*“n3;n1;n2”). Solution of this task is given in Fig. 8. As we want to distinguish different edges between the same nodes, edges are used as loop variables. There are three nested loops used in the solution. Each loop selects one edge for the circle. Actually finding of circles is defined in the loophead of the first loop, however using this loop we are only able to find all edges which are part of some circle, but we don’t know in how many circles this edge is used. Adding the second and third loop we count all circles consisting of different edges 3 times as required in the task specification.

If we know that there are no multi-graphs then the last loop can be omitted, because the existence of the third edge is already validated by patterns in the first and second loop. However in this case it is probably easier to understand if nodes are used as loop variables, but for that again three loops are needed.

Solution of this task is quite lengthy, however if we add temporary classes it is possible to create a shorter and more elegant solution. We extend the metamodel by adding a temporary class “Circle” and connecting it to the class “Edge”. The metamodel extension is shown on bottom of Fig. 9. If such extended metamodel is used then we can simply write a MOLA rule looking for circles and marking the found circles: connecting all edges of a circle to a new instance of the “Circle” class.
To ensure that each circle is found exactly once a NOT constraint (an equivalent to NAC in graph transformation languages) is used stating that this circle hasn’t been marked previously. As in this solution we don’t care about the order of edge finding, the loop counter is increased by 3, to ensure that each circle has been counted three times. This solution is given in Fig. 9.

Fig. 10. Solution of optional task: counting of dangling edges.

Next was an optional task to count dangling edges. The solution is given in Fig. 10. In this case two loops are used. The first one counts edges without a source. To ensure that edges without source and without target are counted only once the second loop counts only edges with a source and without target.

The next task we consider is edge reversing. We selected a solution where a new reverted edge is created and the old edge is deleted. The solution is displayed in Fig. 11. Actually a shorter solution in MOLA is possible; however it is not supported by the current version of MOLA tool.
Fig. 11. Transformation inversing edges.

The next group of tasks was model migration tasks. To implement such tasks it is necessary to add temporary traceability relations to metamodel. In this case it is sufficient to have an association between nodes in both metamodels. The migration transformation from the metamodel graph1 to metamodel graph2 is given in Fig. 12 and from metamodel graph1 to metamodel graph3 in Fig. 13. In both cases at first a new graph in the target model is created. After that all nodes are cloned and traceability links added. Finally all edges are transformed using the traceability information to find the appropriate source and target nodes in the migrated model.
Fig. 12. Model migration transformation. Migrates graph from encoding *graph1* to encoding *graph2*. 
Fig. 13. Solution of optional model migration task. Migrates graph from encoding graph1 to encoding graph3.

Fig. 14. Transformation that deletes node named “n1” (if such node exists) in a graph.
The last mandatory transformation is deletion of the node named “n1”. This transformation is very straightforward (see Fig. 14). We try to find such node using a MOLA pattern and if it is found we delete it. The deletion is represented by a black dashed line. In the extension it was required to delete all incident edges as well. The solution of extension is given in Fig. 15. In this case at first the node is found, then all outgoing edges are deleted, after that all incoming edges are deleted and finally the node itself is deleted.

Fig. 15. Transformation that deletes node named “n1” (if such node exists) and its incident edges in a graph
The last task in the case description is insertion of transitive edges. The easiest way to solve this task in MOLA is using a while loop. While it is possible to insert a transitive edge the MOLA rule is executed repeatedly, after that the end is reached.

Fig. 16. Solution of optional task: insertion of transitive edges.