The TTC 2011 Reengineering Challenge Using MOLA and Higher-Order Transformations

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

Institute of Computer Science and Mathematics, University of Latvia,
Raina bulvaris 29, LV-1459 Riga, Latvia
{agris.sostaks,elina.kalnina,audris.kalnins,
edgars.celms,janis.iraids}@lumii.lv
http://syslab.lumii.lv

Abstract. The Reengineering Challenge of the Transformation Tool Contest 2011 deals with automatic extraction of state machine from Java source code. The transformation task involves complex, non-local matching of model elements. This paper contains the solution of the task using model transformation language MOLA.

The MOLA solution uses higher-order transformation (HOT) to generate a part of the required MOLA program. The described HOT approach allows creating reusable, complex model transformation libraries for generic tasks without modifying an implementation of a model transformation language. Thus model transformation users which are not the developers of the language can achieve the desired functionality more easily.

Keywords: MOLA, model transformation, higher-order transformation

1 Introduction

A solution for the Reengineering Challenge case study[1] of the Transformation Tool Contest 20111 (TTC) has presented in this paper. Model transformation language MOLA[2] has been used to solve the task.

The task is to create a simple state machine model for a Java syntaxgraph model encoding a state machine with a set of coding conventions. The task consists of the core task and two extensions. All of them have been implemented in the solution described in this paper.

The core task is to build states and transitions. States are created from non-abstract Java classes that extends the class named State. The transitions are encoded by method calls to the specific method named Instance() of the next state returning the singleton instance of that state on which the activate() method is called. Transition’s trigger and action attributes are extracted in the extensions of the task. Values for the attributes are dependent mainly on the type of the container within which a transition activation call occurs. Thus, the

1 http://planet-research20.org/ttc2011/
solution should deal with the main challenge of the task - complex, non-local matching of model elements. The understandability, conciseness, correctness, completeness and also performance are the evaluation criteria in the case study.

The task largely has been designed so, that for Java elements meeting certain criteria a parent or child of the specific type must be found. Since the provided Java metamodel has a deep containment and inheritance hierarchy, there are lots of different navigation paths how the searched element may be reached. This leads to the necessity of specifying every single case in the transformation rule for MOLA language. The problem is that MOLA (and also most of other transformation languages) does not support the specific pattern feature - recursive, generic navigation using containment associations. The reference solution provided in the GReTL transformation language[3] heavily relies on this feature and it makes the solution readable and concise. No doubt, it is a great feature and it could be implemented also in MOLA and other languages which lack it. However, we want to present another solution - using higher-order transformations to generate MOLA procedures dealing with complex, non-local matching of model elements.

The paper is structured as follows. Section 2 provides a short overview of MOLA language. Section 3 describes higher-order transformation used by the solution. Section 4 contains description of the solution. The paper ends with discussion in Section 5.

2 MOLA Language

MOLA is a graphical model transformation language which combines the declarative means for pattern specification and imperative control structures determining the order of transformation execution. The formal description of MOLA and also MOLA tool can be downloaded here - http://mola.mii.lu.lv.

The main element of MOLA transformation is a rule (See the gray rounded rectangle in Figure 1). 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 instances to be included in the search are specified using class elements. The traditional UML instance notation (instance name : class name) is used to identify a particular class element. Class elements may contain constraints defined using simple OCL-like expressions. Additionally, a rule may contain association links between class elements. Association links specify links of the exact type required to exist between corresponding instances in a model. Class elements and association links may have {NOT} label which means negative application condition (NAC). No other means for pattern specification exist in MOLA.

In order to iterate through a set of instances MOLA provides a foreach loop statement (See the black rectangle in Figure 1). The loophead is a special kind of a rule that is used to specify the set of instances to be iterated over. The pattern is specified for the loophead in the same way as for ordinary rule. However, the loop variable is a special class element (see the cf : ClassifierReference element
in Figure 1). For each loop is executed for each distinct instance that corresponds to the loop variable and satisfies all constraints of the pattern. In fact, the loop variable plays the same role as an iterator in the classical programming languages.

![Diagram](image)

**Fig. 1.** The isClassSubclassOf procedure

Figure 1 provides an example of a model transformation procedure written in MOLA language. The `isClassSubclassOf` procedure answers whether a class is a subclass of another class. It is a simple case for non-local matching of model elements. Since the example is a part of the solution, here and in other examples the Java metamodel given by case authors is used. There are three parameters for the procedure (white boxes in the upper part of the procedure). The first two parameters are classes to be examined. The third parameter is an in-out parameter where the result (true or false) is stored.

Since MOLA language (unfortunately) does not have recursive patterns, the task should be solved using a recursive procedure. The procedure contains a foreach loop iterating over all superclasses of the given class. If a superclass is the given one, then the result is positive. Otherwise the procedure is called recur-
sively passing the superclass as the first parameter. If no class in the inheritance hierarchy correspond to the given class, then the result is negative.

A much more complex example for non-local matching is finding a class method which contains the given expression. This task arises because the value of trigger attribute of a transition depends primarily on the class method and secondarily on the statement (e.g. is it a switch case or catch block) within which activation call occurs. The solution of the task also requires a recursive approach, because the containment hierarchy of Java expressions and statements is deep and recursive. E.g. the `references::MethodCall` class is a subclass of the `expressions::Expression` class. There are 18 classes between them in the inheritance hierarchy and many of them have a containment link to an owner class which is a subclass of a class in the same hierarchy (e.g. `XXXExpression` and `XXXExpressionChild` classes). It means that every containment case should be specified in MOLA procedure as a separate rule producing large number of rules to be created.

Figure 2 shows an excerpt of a procedure finding an owning class method for an arbitrary Java element. The procedure has two parameters - the first one is the
Java element corresponding to `commons::Commentable` class (every element in the metamodel extends this class), the second is an in-out parameter containing the result - the owning class method. If the passed argument already is a class method, it is the answer and execution of the procedure ends. Otherwise the type of the argument should be determined (actually the kind of the argument, not exact type). It is done using a chain of text statements (yellow rounded rectangles) linked by `{ELSE}` flows. We are interested only in those types which have containment associations. When the type of the argument is known, we can check whether the argument has a containment link. If it has, then the procedure is called recursively passing the container as an argument. Otherwise we try to determine whether the argument corresponds to another type. The recursive calls stop, when the owning class method is found or the root of the containment hierarchy has been reached.

In total 22 distinct types and 44 distinct composition links should be checked which results into 22 text statements and 44 rules in the MOLA procedure. However no element has been specified by hand - the `FindOwnerOfTypeClassMethod` has been entirely generated. The details are provided in the next sections. Of course, if some semantic constraints would be taken into account (e.g. the `activate()` method is a void method and can not be part of an `and` or `or` expression), the number of rules needed for MOLA would be much less. However, since these constraints haven’t been stated in the task description, we have made a full solution.

3 Higher-Order Transformations using MOLA

MOLA Tool has been built using the transformation-based graphical tool building framework METAcclipse[4]. The functionality of the graphical tool is specified using model transformations in METAcclipse. MOLA Tool has been built using MOLA itself. As a consequence a MOLA transformation definition is stored as a model and we can operate with it as with an ordinary model. Thus, we can use MOLA to generate MOLA. Of course, the abstract syntax of MOLA language has been used. See MOLA metamodel in the reference manual2.

Figure 3 shows the higher-order transformation written in MOLA generating the procedure `FindOwnerOfClassTypeMethod` described in the previous section. Actually, the procedure is slightly modified. See [http://mola.mii.lu.lv/img/generated.png](http://mola.mii.lu.lv/img/generated.png) for an image of the procedure. Two additional parameters are added to help to find the information required for implementing extensions of the case study. The main purpose of the `FindOwnerOfClassTypeMethod` procedure was to find the owning class method of a Java model element. Additionally a normal switch case and a catch block are found containing the element if they exist.

Two upper MOLA rules generate the header and the text statement of the procedure checking if the argument is a class method. The next two rules generate two text statements checking if the argument is a normal switch case or

---

2 [http://mola.mii.lu.lv/mola2fin_refmanual.pdf](http://mola.mii.lu.lv/mola2fin_refmanual.pdf)
a catch block. The next statement is a foreach loop. The loop iterates over all
metamodel classes from the references, expressions and statements packages having and outgoing composition. For each class a text statement checking
if the argument is an instance of this class is generated. The nested foreach loop
goes through all composite associations of the class and generates a MOLA rule
and a call statement for each. All other MOLA rules and text statements have
been used to create appropriate flows between generated MOLA statements.

Fig. 3. Higher-Order Transformation generating the FindOwnerOfTypeClassMethod
procedure

Although it is possible to generate MOLA using MOLA itself, we think a
better option would be Template MOLA[5]. Template MOLA uses the concrete
syntax of MOLA to specify MOLA elements being generated in much more
readable way. However, the tool for Template MOLA has not been yet built properly.

The approach is reusable - the same HOT can be used to generate similar MOLA procedures finding owner of another type or even using another metamodel. Of course, the HOT definition should be changed accordingly, but the number of changes would be rather small. Names of the searched owner class and common superclass should be supplied.

Though only abstract syntax has been provided for generated MOLA procedure, MOLA Tool generates the concrete syntax automatically and uses GraphViz\(^3\) dot for auto layout. Thus, the generated procedures can be viewed and edited further using MOLA Tool as ordinary MOLA procedures.

4 MOLA Transformation for the Reengineering Challenge

Since the helper procedures for the solution of reengineering challenge have been introduced in the previous sections, we can proceed to the complete solution of the task.

The MOLA solution has been shown in Figure 4. The first rule finds the class named State. If the class exists, then the state machine is created. The creation is denoted using red color for class elements and association links in a rule. If the State class does not exist, then the transformation ends.

Next the foreach loop is used to iterate through all non-abstract classes. Call to the isClassSubclassOf procedure is used to determine whether a class is a subclass of the State class. If the class is a subclass then a state instance is created and put into the state machine. Additionally a traceability link is created.

The next foreach loop is used to create transitions. Every method call conforming to the task description (State_Instance().activate()) is handled by the loop. Call to the FindOwnerOfTypeClassMethod method returns the owning class method (@own), owning normal switch case (@ownNSC) and owning catch block (@ownCB). When the owning method has been obtained, corresponding class and state can be found. In the same rule a transition is created. That was the solution for the core task.

The solution of the first extension starts with the next rule. Accordingly to the description of the task the trigger name for transitions whose activation occurs outside run method is equal to the methods name. If this condition is satisfied, then the trigger name is set. Otherwise the second case should be examined. If the pattern of a MOLA rule fails, then the next statement reached by flow labeled {ELSE} is executed. In our case, it is a rule checking, whether an owning normal switch case exists. If it exists, then the enumeration constant is found and set as a trigger name. Otherwise the third case should be examined. If an owning catch block exist, then the corresponding exception class is located.

\(^3\) http://http://www.graphviz.org/
and the trigger name set to the name of the class. Otherwise the trigger name
is set to “- -”.

The two last rules of the loop solve the second extension. A call to the send
method is searched in the statement list container owning the activation call. If
such method call is found then the transitions action name is set to the name of
equation constant passed as an argument to the call. Otherwise the actions
name is set to “- -”.

Fig. 4. MOLA solution
4.1 Architecture of the Solution

Since the solution involves the generation of model transformations, an overview of how it has been achieved technically is given in this section.

MOLA transformations can be compiled to several technical spaces (model repositories) - Eclipse Modeling Framework (EMF), JGraLab [6] and MII_REP developed by IMCS, University of Latvia. Example models and metamodels in the reengineering challenge conform to the EMF technical space.

Development of model transformations begins with importing source and target Ecore metamodels into MOLA Tool. 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 - missing opposite references are added automatically. Efficient transformation development in MOLA requires additional metamodel elements for storing the traceability links between the source and target model elements. These elements have to be added manually. In the given case, one association between the Class and State classes is sufficient. When the metamodel is ready, the development of MOLA procedures can go on.

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. 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.

Development of higher-order transformation requires the model transformation metamodel. Since MOLA Tool has been built using MOLA itself, the MOLA metamodel for model transformations is available. Development of higher-order transformation does not differ from development of ordinary MOLA transformation - the same MOLA Tool is used. However the technical space where transformations are executed is different from EMF. Although METAclipse framework is based on Eclipse technologies, the models are stored and transformations run on the metamodel-based repository MII_REP. Thus MOLA transformations should be compiled to C++ rather than Java and different execution environment should be used.

The higher-order transformation is executed directly on the repository used by MOLA Tool. Another pre-defined model transformation creates the concrete syntax elements from the MOLA model in abstract syntax. Now the MOLA procedure can be opened in the MOLA Tool. MOLA elements can be positioned in a MOLA diagram manually or using auto layouter as it has been done in the solution.
5 Conclusions

In this paper the MOLA solution to the Reengineering Challenge has been described. Since MOLA language lacks means how to deal with recursive, generic patterns in a concise and elegant way, an approach involving model transformation generation (higher-order transformations) has been proposed (HOT approach).

The HOT approach allows to deal with complex situations when a model transformation language lacks the desired constructs. In such cases the solution using means available in the language requires plenty of manual coding (as it was in our case). Another solution would be introducing the new means into the language. However it requires changes in the implementation of the language - the compiler or interpreter as well as editor should be changed.

The main advantages of HOT approach are:

- There is no need to change the implementation of language to introduce the desired functionality. We have added a procedure finding a class method owning the given arbitrary Java element.
- HOTs are reusable - the same transformation can be used in another model transformation project.
- HOTs are flexible - if some changes are needed to the functionality of the generated model transformation it can be easily added by changing the HOT definition or even adding the new functionality manually to the generated code. In our solution the generated procedure searches an owning normal switch case and catch block additionally to the class method.

The main disadvantage of the HOT approach is that it requires a deep knowledge of the language being generated. In our case, MOLA metamodel (abstract syntax) should be familiar to the HOT developer. A HOT specification using the concrete syntax of the transformation language being generated would be a great improvement.

Another issue is the maturity of HOT tools. In our case running higher-order transformation requires rather deep understanding of the architecture of MOLA Tool. However, we believe that it is matter of time until the HOT tools will reach sufficient maturity.

The MOLA solution of reengineering challenge consists of three MOLA procedures. The main procedure consists mainly of rules describing patterns corresponding to the requirements described in the case description. Both subprocedures hide the recursive patterns needed to traverse the class and containment hierarchy of a Java model. The FindOwnerOfTypeClassMethod procedure has been entirely generated using a higher-order model transformation. It is hard to compare the conciseness and understandability of the solution to the reference solution provided by case author, because of the different nature of languages and approaches - graphical versus textual language. However, no doubt the recursive pattern elements are a great advantage for the GReTL.

The MOLA solution implements the core task as well as extensions. The MOLA solution has been tested on the models provided by the case author.
The execution of transformation on the simple model took in total 942 ms on the Intel(R) Core2 CPU 2.67 GHz, 4 GB RAM desktop computer having 32-bit Windows Vista operating system. It includes 269 ms loading model, 201 ms copying to the intermediate representation (having associations navigable both ways), 329 ms actually running the transformation, 1 ms extracting target model from intermediate representation and finally 133 ms saving the target model. The execution of the transformation on the medium model took 1118 ms in total. Loading model took 301 ms and running transformation took 508 ms. Other times were comparable to the execution times on the small model. The execution of transformation on the big model failed due to the Java heap space problems of Eclipse environment.

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

**References**

6. Universit Koblenz-Landau, Institute for Software Technology, Graph Laboratory http://userpages.uni-koblenz.de/~ist/JGraLab