Solving the TTC 2011 Reengineering Case with MOLA and Higher-Order Transformations

Agris Sostaks Elina Kalnina Audris Kalnins Edgars Celms
Janis Iraids

Institute of Mathematics and Computer Science, University of Latvia,
Raina bulvaris 29, LV-1459, Riga, Latvia

{agris.sostaks, elina.kalnina, audris.kalnins, edgars.celms, janis.iraids} @lumii.lv

The Reengineering Case 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 transformations (HOT-s) 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 who are not the developers of the language can achieve the desired functionality more easily.

1 Introduction

A solution for the Reengineering Challenge case study [3] of the Transformation Tool Contest 2011[1] (TTC) has been presented in this paper. Model transformation language MOLA [5] has been used to solve the task. The task is to create a simple state machine model for a Java syntax graph model encoding a state machine with a set of coding conventions. The task consists of a core task and two extensions.

The core task is to build states and transitions. States are created from non-abstract Java classes that extend 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 solution should deal with the main challenge of the task - *complex, non-local matching* of model elements.

The task has been largely designed so, that for Java elements meeting certain criteria a parent or a 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. A higher-order transformation has been used 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 the MOLA language and tool. Section 3 describes the higher-order transformation used by the solution. Section 4 contains a description of the solution. The paper ends with discussion in Section 5. Detailed description of the transformation definition (including *source code* - MOLA diagrams) can be found in Appendix.

[1] `http://planet-research20.org/ttc2011/`

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. The instances to be included in the search are specified using *class elements*. 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 the corresponding instances in a model.

In order to iterate through a set of instances MOLA provides a *foreach loop* statement (See the bold 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). A foreach loop is executed for each distinct instance that corresponds to the loop variable and satisfies all constraints of the pattern.

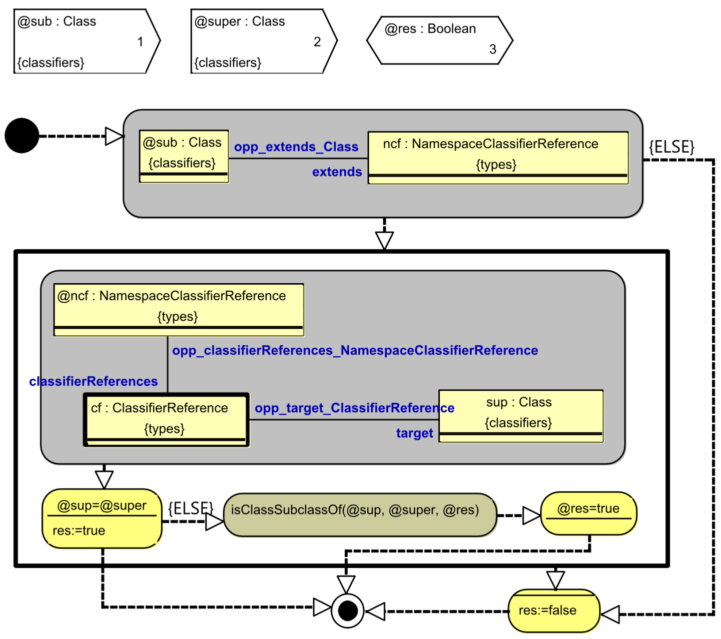

Figure 1: The isClassSubclassOf procedure

MOLA transformations can be compiled to several technical spaces (model repositories) - Eclipse Modeling Framework (EMF), JGraLab [1] 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 in both directions (this permits to perform an efficient pattern matching using simple matching algorithms). Since a typical Ecore metamodel has many associations navigable in one direction, the import facility has to extend the metamodel - *missing* opposite references are added automatically. The traceability links between the source and target model elements can be added manually. In the given case, one association between the `Class` and `State` classes is sufficient. 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.

Since the MOLA language does not have recursive patterns (as most graphical languages), tasks like to answer whether a Java class is a subclass of another class should be solved using a recursive procedure. A much more complex example for non-local matching is finding a Java class method which contains the given expression. This task arises because the value of a 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 the 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. It means that every containment case should be specified in MOLA procedure as a separate rule thus producing large number of rules to be created.

In total 22 distinct types and 44 distinct composition links should be checked which results in 22 text statements and 44 rules in the MOLA procedure. However no element has been specified *by hand* - the MOLA procedure has been entirely generated. Of course, if some semantic constraints would be taken into account (e.g. the `activate()` method is a void method and cannot 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 METAclipse [6]. The functionality of the graphical tool is specified using model transformations in METAclipse. 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. The abstract syntax of MOLA language has been used. See the MOLA metamodel in the reference manual [2].

Figures 3 and 4 show the higher-order transformation written in MOLA generating the procedure `FindOwnerOfTypeClassMethod` which implements the more complex task described in the previous section. See `http://mola.mii.lu.lv/img/generated.png` for an image of the procedure. Two additional parameters are added to help finding the information required for implementing extensions of the case study. A normal switch case and a catch block containing the element are found.

Although it is possible to generate MOLA using MOLA itself, we think a better option would be to use Template MOLA [4] - a graphical template language for transformation synthesis. Template MOLA uses the concrete syntax of MOLA to specify MOLA elements being generated in a more readable way. However, the tool for Template MOLA has not been built properly yet.

Though generated MOLA procedure is in the abstract syntax, MOLA Tool generates the concrete syntax automatically and uses GraphViz[2] *dot* for auto layout. Thus, the generated procedures can be viewed and edited further using MOLA Tool as ordinary MOLA procedures. 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

[2]`http://www.graphviz.org/`

EMF. Although the METAclipse framework is based on Eclipse technologies, models are stored and transformations run on the metamodel-based repository MII_REP. Therefore the higher-order transformation is executed directly on the repository used by MOLA Tool.

4 MOLA Transformation for the Reengineering Challenge

The MOLA solution has been shown in Figure 5 (see Appendix). 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 terminates. Next the foreach loop is used to iterate through all non-abstract classes (see A in Figure 5). 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. At first the core task has been solved (see B in Figure 5). 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, the corresponding class and state can be found. In the same rule a transition is created.

The solution of the first extension starts with the next rule (see C in the Figure 5). According to the description of the task the trigger name for transitions whose activation occurs outside the `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 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 (see D in Figure 5). A call to the send method is searched in the statement list container owning the activation call. If such a method call is found then the action names of the transitions are set to the name of enumeration constant passed as an argument to the call. Otherwise the name of the action is set to "- -".

5 Conclusions

In this paper the MOLA solution to the Reengineering Challenge has been described. MOLA solution implements the core task as well as both extensions. 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 these constructs.

The main advantages of the HOT approach are:

- There is no need to change the implementation of a language to introduce the desired functionality. We have added a procedure which finds 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 for an owner (normal switch case or 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. MOLA metamodel (abstract syntax) should be familiar to a developer of a HOT. This leads to the overspecification. It is the main reason why the solution was awarded with a lower score for the conciseness. We believe that the graphical versus textual languages issue has also played a significant, but not the main, role (graphical languages got the lowest scores for the conciseness). 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 a deep understanding of the architecture of MOLA Tool.

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 1280 ms on a virtual machine within the SHARE environment[7]. It includes 280 ms loading model, 170 ms copying to the intermediate representation (having associations navigable both ways), 720 ms actually running the transformation, less than 1 ms extracting the target model from intermediate representation and finally 110 ms saving the target model. The execution of the transformation on the medium model took 1600 ms in total. The execution of transformation on the big model failed due to memory consumption problems of MOLA runtime environment. The problem is being fixed by improving the algorithm and fixing bugs in the runtime 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

[1] *Graph Laboratory*. Available at `http://userpages.uni-koblenz.de/~ist/JGraLab`.

[2] *The MOLA Language*. Available at `http://mola.mii.lu.lv/mola2fin_refmanual.pdf`.

[3] Tassilo Horn (2011): *Program Understanding: A Reengineering Case for the Transformation Tool Contest*. In: *Pieter Van Gorp, Steffen Mazanek & Louis Rose, editors: TTC 2011: Fifth Transformation Tool Contest, Zürich, Switzerland, June 29-30 2011*.

[4] Elina Kalnina, Audris Kalnins, Edgars Celms & Agris Sostaks (2010): *Graphical Template Language for Transformation Synthesis*. In Mark van den Brand, Dragan Gaševic & Jeff Gray, editors: *Software Language Engineering*, *LNCS* 5969, Springer Berlin / Heidelberg, pp. 244–253, doi:10.1007/978-3-642-12107-4_18.

[5] Audris Kalnins, Janis Barzdins & Edgars Celms (2005): *Model Transformation Language MOLA*. In Uwe Aßmann, Mehmet Aksit & Arend Rensink, editors: *Model Driven Architecture, European MDA Workshops: Foundations and Applications, MDAFA 2003 and 2004, Twente, The Netherlands, June, 2003 and Linköping, Sweden, June, 2004, Revised Selected Papers*, *LNCS* 3599, Springer, pp. 62–76, doi:10.1007/11538097_5.

[6] Audris Kalnins, Oskars Vilitis, Edgars Celms, Elina Kalnina, Agris Sostaks & Janis Barzdins (2007): *Building Tools by Model Transformations in Eclipse*. In: *Proceedings of DSM'07 Workshop of OOPSLA 2007*, Jyväskylä University Printing House, Montreal, Canada, pp. 194–207.

[7] Agris Sostaks, Elina Kalnina, Audris Kalnins, Edgars Celms & Janis Iraids (2011): *SHARE demo related to the paper Solving the TTC 2011 Reengineering Case with MOLA*. Available at `http://is.ieis.tue.nl/staff/pvgorp/share/?page=ConfigureNewSession&vdi=XP-TUe_TTC11_MOLA.vdi`.

6 Appendix

Figure 1 (on page 2) 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. 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.

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 *true*. Otherwise the procedure is called recursively passing the superclass as the first parameter. If no class in the inheritance hierarchy corresponds to the given class, then the result is *false*.

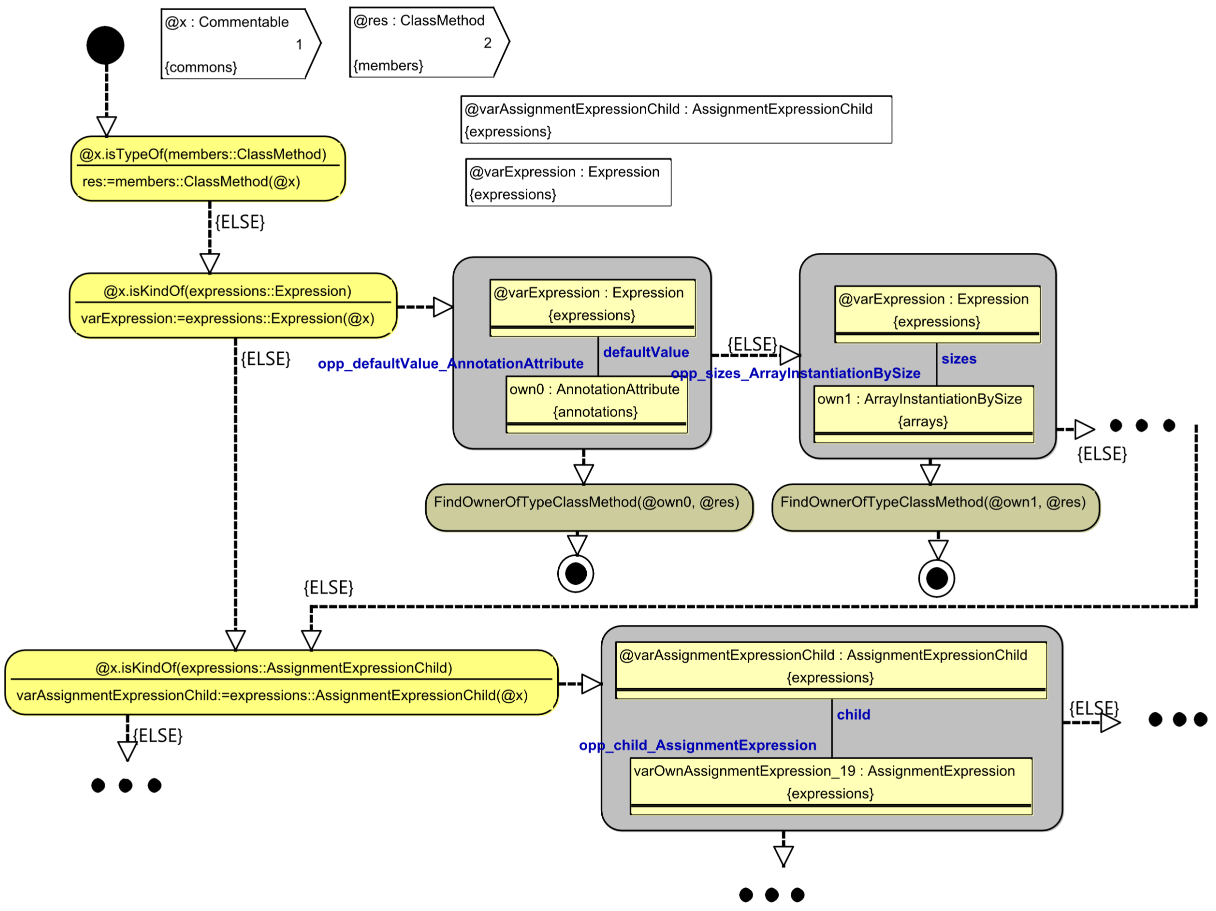
@x : Commentable
1
{commons}
@res : ClassMethod
2
{members}
@varAssignmentExpressionChild : AssignmentExpressionChild
{expressions}
@varExpression : Expression
{expressions}
@x.isTypeOf(members::ClassMethod)
res:=members::ClassMethod(@x)
{ELSE}
@x.isKindOf(expressions::Expression)
varExpression:=expressions::Expression(@x)
{ELSE}
@varExpression : Expression
{expressions}
defaultValue
opp_defaultValue_AnnotationAttribute
own0 : AnnotationAttribute
{annotations}
{ELSE}
@varExpression : Expression
{expressions}
sizes
opp_sizes_ArrayInstantiationBySize
own1 : ArrayInstantiationBySize
{arrays}
{ELSE}
FindOwnerOfTypeClassMethod(@own0, @res)
FindOwnerOfTypeClassMethod(@own1, @res)
{ELSE}
@x.isKindOf(expressions::AssignmentExpressionChild)
varAssignmentExpressionChild:=expressions::AssignmentExpressionChild(@x)
{ELSE}
@varAssignmentExpressionChild : AssignmentExpressionChild
{expressions}
child
opp_child_AssignmentExpression
varOwnAssignmentExpression_19 : AssignmentExpression
{expressions}
{ELSE}

Figure 2: Excerpt of the FindOwnerOfTypeClassMethod procedure

Figure 2 shows an excerpt of a procedure finding an owning class method for an arbitrary Java element. The procedure has two parameters (white boxes in the upper part of the Figure 2) - 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 is already a

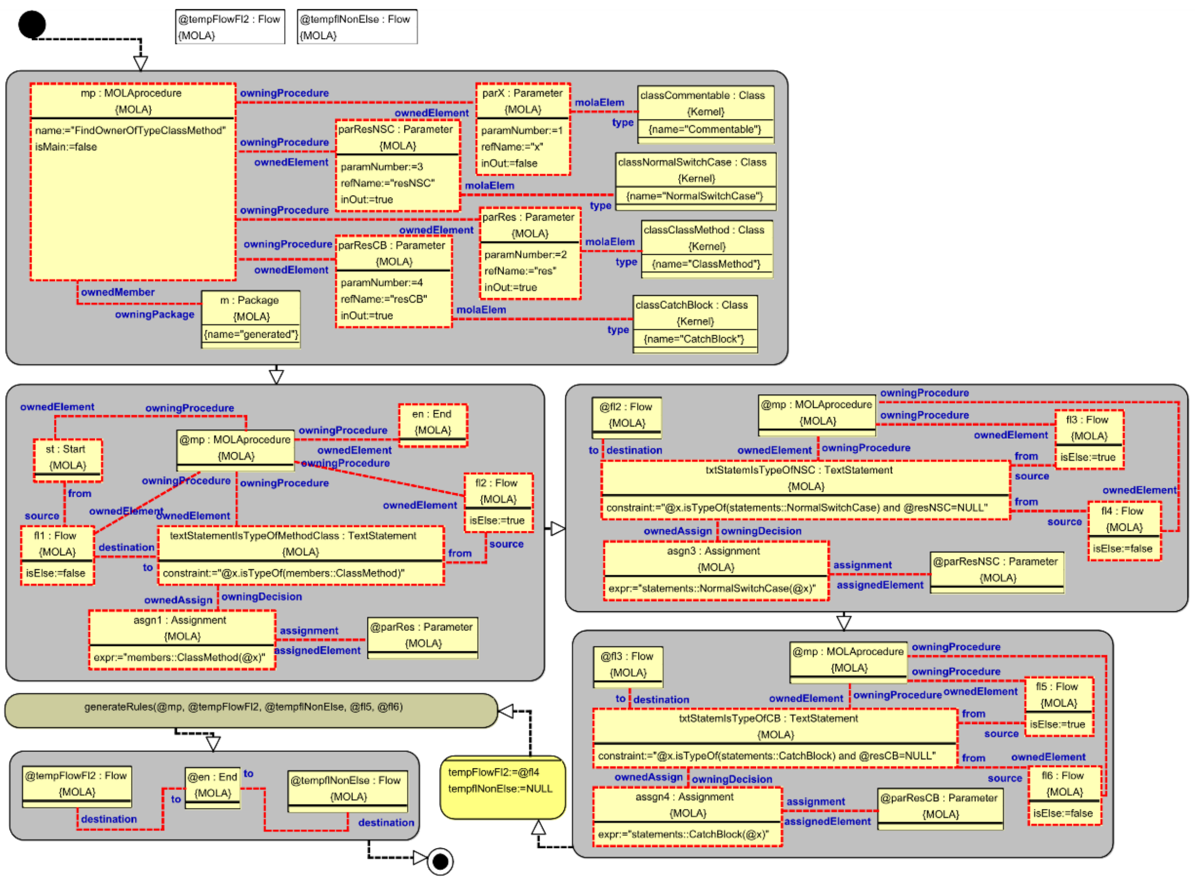
mp : MOLAprocedure {MOLA}
name:="FindOwnerOfTypeClassMethod"
isMain:=false
m : Package {MOLA} {name="generated"}
generateRules(@mp, @tempFlowFl2, @tempflNonElse, @fl5, @fl6)
tempFlowFl2:=@fl4
tempflNonElse:=NULL
textStatementIsTypeOfMethodClass : TextStatement {MOLA}
constraint:="@x.isTypeOf(members::ClassMethod)"
txtStatemIsTypeOfNSC : TextStatement {MOLA}
constraint:="@x.isTypeOf(statements::NormalSwitchCase) and @resNSC=NULL"
txtStatemIsTypeOfCB : TextStatement {MOLA}
constraint:="@x.isTypeOf(statements::CatchBlock) and @resCB=NULL"

Figure 3: Higher-Order Transformation generating the fixed part of FindOwnerOfTypeClassMethod procedure

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 the 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 so, 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.

Figure 3 shows the first part of higher-order transformation. 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 a catch block.

The next statement is a foreach loop (see Figure 4). The loop iterates over all metamodel classes from the `references`, `expressions` and `statements` packages having an 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

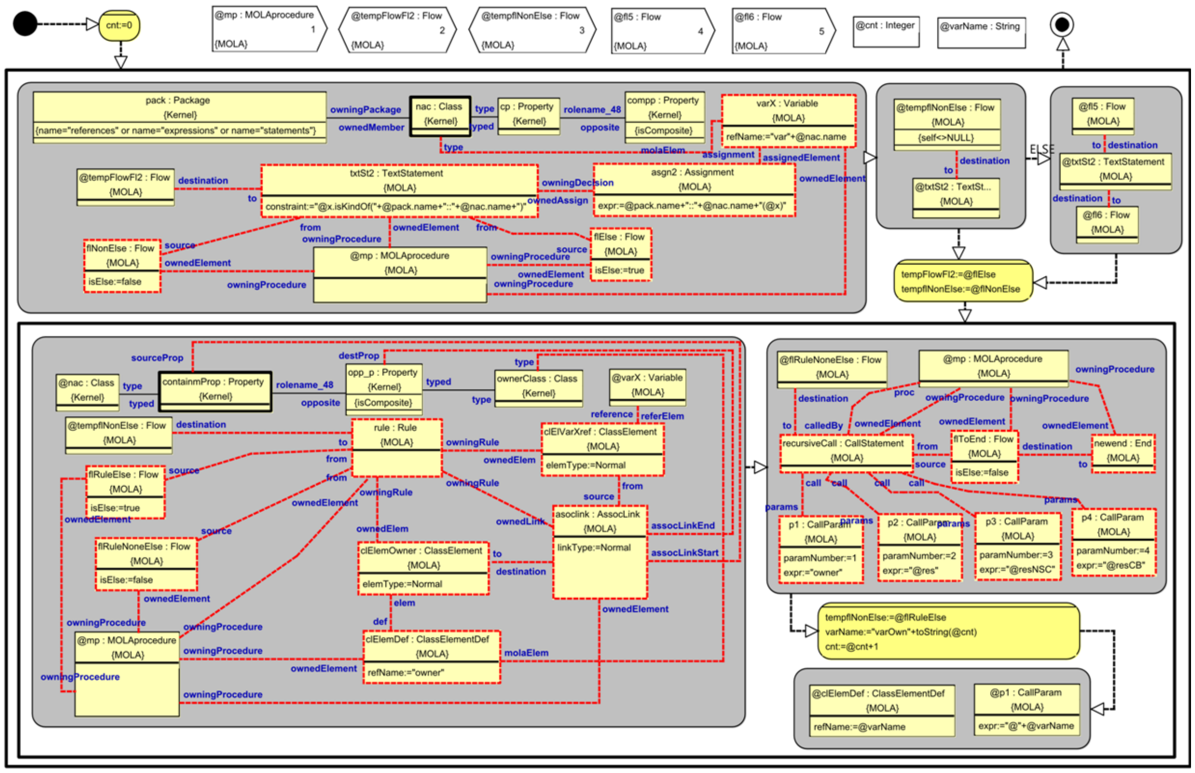

Figure 4: Higher-Order Transformation generating the variable part of FindOwnerOfTypeClassMethod procedure

between generated MOLA statements.

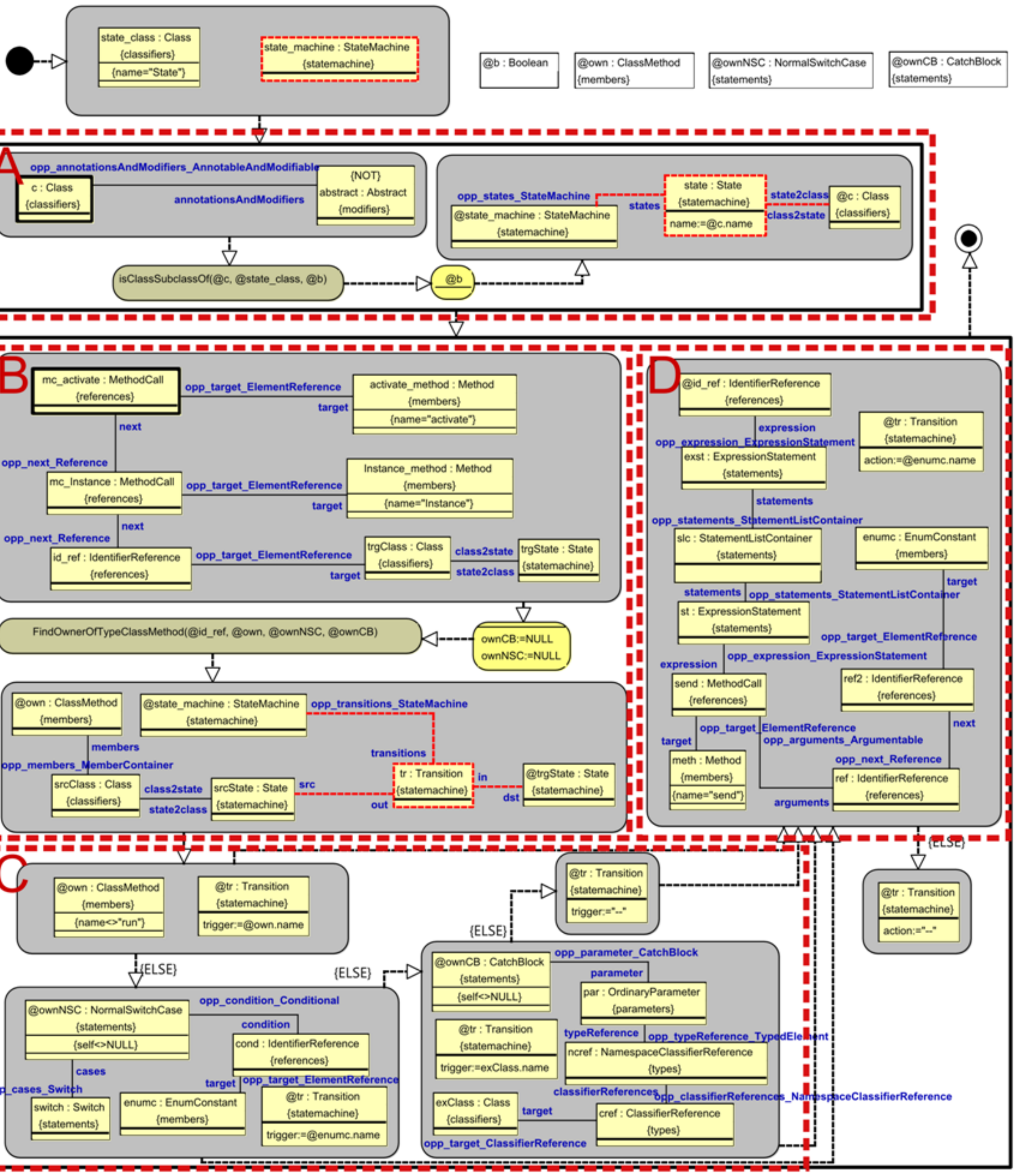

Figure 5: The main part of the MOLA solution