A Solution to the Java Refactoring Case Study using eMoflon

Sven Peldszus
Technische Universität Darmstadt
Real-Time Systems Lab
Merckstr. 25
64283 Darmstadt, Germany

Géza Kulcsár
Technische Universität Darmstadt
Real-Time Systems Lab
Merckstr. 25
64283 Darmstadt, Germany

Malte Lochau
Technische Universität Darmstadt
Real-Time Systems Lab
Merckstr. 25
64283 Darmstadt, Germany

{sven.peldszus@stud|geza.kulcsar@es|malte.lochau@es}.tu-darmstadt.de

Our solution to the Java Refactoring case study of the Transformation Tool Contest (TTC 2015) is implemented using eMoflon, a meta-modeling and model transformation tool developed at the Real-Time Systems Lab at TU Darmstadt. The solution, available as a virtual machine hosted on SHARE [5] and at GitHub [6], includes a bidirectional synchronization between a Java model and an abstract program graph specified using Triple Graph Grammars (TGG) as well as a graph-based implementation for two refactoring operations using Story Driven Modeling (SDM).

1 Introduction

The Java Refactoring case study [3] of the Transformation Tool Contest 2015[1] revolves around a challenging object-oriented refactoring scenario. Two classical refactoring operations, `Create Superclass` and `Pull Up Method`, have to be implemented by solution developers, taking Java source code as input and producing a refactored version of it as output. We use a meta-model specified in the case study, called the Program Graph (PG). The PG is an abstract representation of the input Java program and is used to define and perform the given refactoring operations on this model of the program. One of the main difficulties comes from the bidirectional nature of synchronizing source code and program graph. Our tool eMoflon [4] supports both EMF meta-modeling and bidirectional transformations using *Triple Graph Grammars* (TGGs). TGGs [8] are a rule-based, declarative language, which can be used for specifying transformations, where both directions (forward and backward transformation) can be derived from the same specification.

Another eMoflon feature, *Story Driven Modeling* (SDM), [1] is used in our solution to implement refactorings. SDM is a visual language for describing programmed graph rewritings; an SDM method consists of a set of graph transformation rules with an additional control flow specification to describe their execution order dependencies in an imperative fashion.

In this paper, we investigate to what extent TGGs are able to cope with advanced bidirectional text-to-model scenarios with change propagation by solving the Java Refactoring case study of the TTC 2015. We use the given PG format as the abstract representation for Java programs. In the following, we provide a stepwise, detailed description of the solution including the technical difficulties that arose and evaluate the solution.

[1] http://www.transformation-tool-contest.eu/

2 The Solution using eMoflon, TGGs and SDM

In the following, we give a detailed description of the steps of our solution.

Java to JaMoPP. The Java source code is parsed and converted into an intermediate EMF representation using the JaMoPP framework [2]. To quote the website of JaMoPP: "JaMoPP is a set of Eclipse plug-ins that can be used to parse Java source code into EMF-based models and vice versa."[2]

JaMoPP to PG. While working with the JaMoPP meta-model for Java, we have found out that some parts of it do not comply with the PG meta-model and with some properties of the planned TGG translation. Two preprocessing actions are necessary to make a JaMoPP model instance TGG-conform.

Creating the package structure. JaMoPP encodes the package hierarchy of the program into dot separated string or as array of strings. As it would require extra efforts and the usage of external hand-written code to handle these constructs when specifying our TGG, we decided to implement this transformation as a preprocessing step in order to keep our TGG clean and concise.

Retaining the parameter order of methods. A transformation specified by a set of TGG rules is per definition nondeterministic, i.e., if the source side of a rule has multiple matches in a source model, we cannot be sure in which order they will be processed. To preserve the original order of a parameter list, which is represented by independent child nodes of a method node, we have to turn the set of parameter nodes into a list representation so that the parameter nodes can only be processed in the given order.

TGGs describe a correspondence between instances of a *source* and a *target* meta-model, specified by means of a mediating *correspondence graph* (hence the name Triple Graph Grammars). A TGG specification consists of declarative rules. A transformation using TGGs consists in building up a target model incrementally on the basis of a source model (or vice versa) using the correspondence links between the elements of the models. Applying a TGG rule essentially means that a given structure in the target model is built up which corresponds to a part of the source model which is matched by the source side of the rule definition.

Our TGG specification consists of 20 rules. We have identified 5 main components which have to be considered: initialization of the PG, packages, classes, methods, and fields. In Figure 1, we show a sample rule `MethodNameCreate` to introduce our visual TGG syntax and to give an idea about the rule semantics. For further details of the TGG implementation, please refer to [5, 6].

By convention, the source node part is on the left and the target node side is on the right, with the correspondence graph (hexagonal boxes) in between. Boxes and edges marked with ++ (highlighted in green) are the elements created by the rule application. All other boxes and edges represent the context (elements which have to be present for the rule to be applied). A crossed-out box denotes a negative application condition: the object must not be part of the context. The box with an expression and two outgoing edges (in the middle) is a constraint, which ensures that the name attributes of the referred elements have the same value (here, the built-in `eq` function is used; however, there are various other built-in functions and the developer can also create custom ones). The meaning of this rule is the following: whenever there is a class in the source with a corresponding class in the target, if a method of the source class is not yet translated (thus, processed at application time, hence its green color), and the target PG does not have a method with the same name, then a new method and a corresponding method definition are created in the target.

Refactoring of the PG. The refactoring rules `Pull Up Method` and `Create Superclass` have been implemented using Story Driven Modeling (SDM) [1]. As these operations do not have to be bidirectional, it was a convenient choice to use SDMs which comprise a more flexible way of specifying

[2]http://www.jamopp.org/

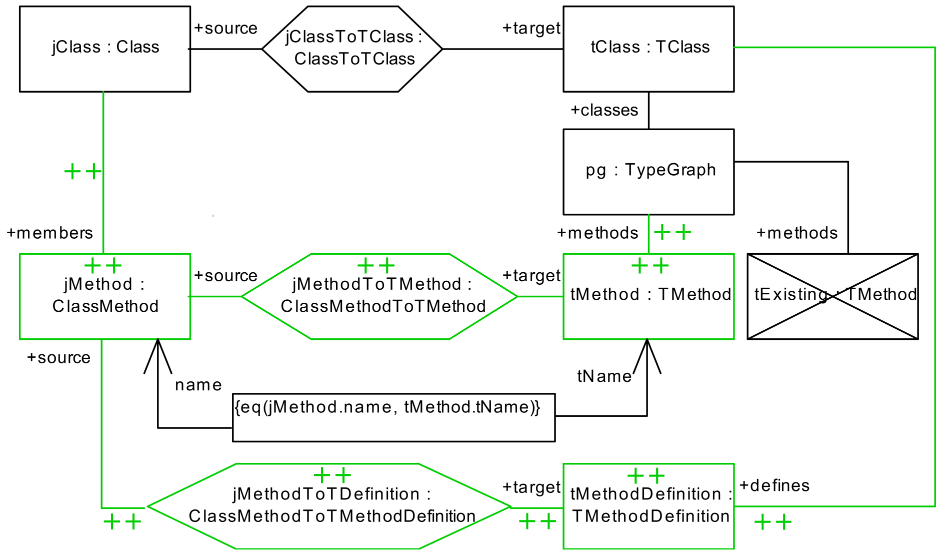

Figure 1: Example TGG rule `MethodNameCreate`

transformations compared to TGGs.

SDMs provide a way to implement methods of classes of a meta-model (similar to object-oriented programming) in a visual manner based on graph transformation, combining declarative graph transformation rules with an imperative control flow. The basic building blocks of an SDM specification are the *story nodes*. Each story node contains a single graph transformation operation, which is applied according to the standard graph transformation principles (i.e., nondeterministically on a matching part of the model) when the story node is activated. The story nodes are activated as determined by the control flow, with the additional possibilities of adding `if-else` conditions and `for each` loops.

There are two methods implemented for both refactoring operations in the corresponding classes of the PG meta-model. The `isApplicable` methods simply check the feasibility of the rule application to prevent the modification of the PG if a refactoring is not even executable. Thereupon, the `Perform` methods perform the actual refactorings if possible.

In this paper, we omit an elaborated presentation of all our SDM methods; instead we show an example method, introduce our visual SDM syntax, give an intuition about how the method works and refer the reader to [5, 6] for further details.

Figure 2 shows an example SDM method `csc_Perform` which implements the actual application of the `Create Superclass` after the preconditions have been checked. The execution starts with the start node (black circle on top left) and follows the arrows. The larger rounded boxes denote story nodes; each story node contains a graph transformation rule which is applied as the containing story node is activated. A rule application consists in finding a match for the depicted graph pattern in the model where the SDM method has been called, deleting the elements marked with `--` (highlighted in red) and creating the ones marked with ++ (highlighted in green). Boxes with a thick edge correspond to bound object variables that are matched to a fixed object in the model. A story node may have two outgoing edges: the execution continues through `Success` if the application was successful and through `Failure` if not. Story nodes can alternatively contain external method calls. Cascaded-style boxes represent `for each` loops, where the rule is applied to each possible match in the model with a loop body executed after each match (`Each Time` edge). After all the matches have been processed, the loop is exited (`End` edge).

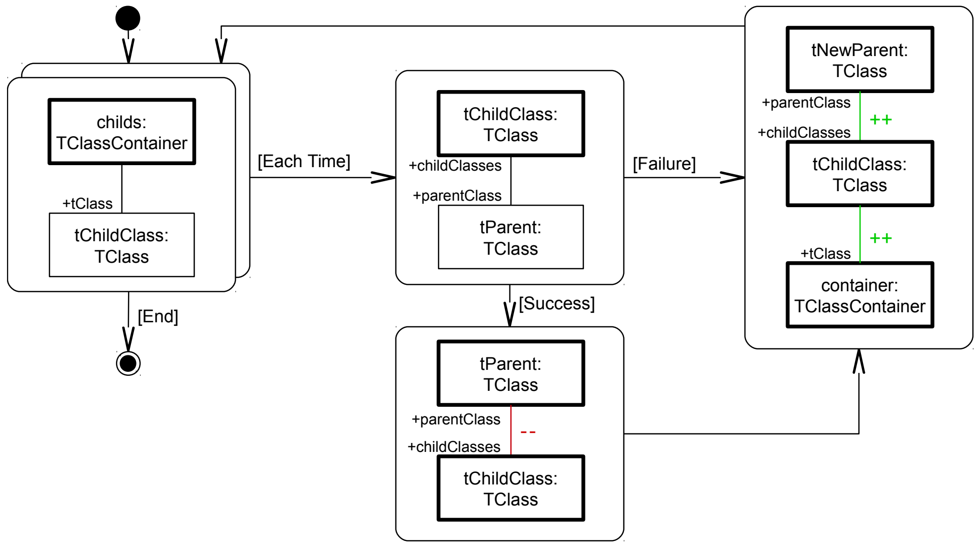

Figure 2: Example SDM method `csc_Perform`

The depicted rule, `csc_Perform`, does the following: after putting the new parent class into the PG by creating the corresponding edge, the old parent of the child classes is identified. Afterwards, in a loop, the parent reference of each child class is newly created to point to the parent created by the refactoring and the old reference is deleted.

PG to JaMoPP. As our TGG describes both a forward and a backward transformation, this step of the transformation requires no extra development efforts. TGGs in eMoflon provide a synchronisation algorithm based on model deltas: whenever one side of a TGG (in our case, the PG instance) is changed, the modification delta is calculated and the TGG mechanism is able to update the other side of the model in correspondence with the change delta. Multiple refactoring operations are performed as a single batch after all the preconditions have been checked by using a bookkeeping mechanism.

JaMoPP to Java. Similar to the first step, the translation of the EMF model to Java code belongs to the central functionality of JaMoPP.

3 Evaluation

Correctness and performance. The case study contains 20 test cases, in which one or more refactorings have to be performed. The feasibility of the given refactoring operations is correctly determined in all test cases. Most of the execution time (60 %) is spent with the Java-to-PG transformation, where JaMoPP consumes almost 30 % of the overall time; although we expected the TGG execution to be the most expensive step, it only takes about 14 % of the whole process (together in both directions). The average execution time for one test case is 0.3367 sec.

Soft aspects. Utilizing TGGs for the synchronization part is responsible for the greatest advantages and disadvantages at once. TGGs provide a powerful declarative language, where the resulting transformations between the source and the target models are consistent regarding the correspondence specified by the TGG. Moreover, by using TGGs, the synchronization part of the challenge requires no extra efforts as a model synchronization algorithm for TGG specifications is already part of eMoflon. The price to pay for those formal and algorithmic properties is the slower execution time compared to task-optimized, imperative solutions. Extending a TGG might also become problematic as new rules might overlap with old ones, thus, possibly altering the behavior of the core specification.

By using SDMs for specifying refactorings, we have an approach based on graph transformation to handle the PG-based refactoring scenario of the challenge. In addition, the visual specification style facilitates the understanding of the refactoring conditions and operations. Naturally, the resulting generated Java code might fall short in terms of performance if compared to an equivalent hand-written implementation from an experienced Java developer.

4 Conclusion and Future Work

In this paper, we presented our solution for the object-oriented Java refactoring case study of the Transformation Tool Contest 2015. Our solution is implemented using the eMoflon meta-modeling and graph transformation tool, developed at the Real-Time Systems Lab of the TU Darmstadt.

We conclude that both of the transformation languages supported by eMoflon, namely TGGs and SDMs can be utilized for different subtasks of the required transformation chain. TGGs in eMoflon also provide a synchronization algorithm which makes eMoflon a highly adequate tool to deal with bidirectional model synchronization problems similar to the one described in the challenge. With SDMs, we have the possibility to specify the actual refactoring operations in a visual and graph-based manner. (For more information about the difference between TGG and SDM as well as their interplay in the present refactoring scenario, we refer the interested reader to [7].)

Our future work includes the examination of the tool MoDisco[3] (having similar functionality to JaMoPP) in order to potentially reduce the need for pre- and postprocessing and to define a more structured and sophisticated TGG. Moreover, we would like to conduct experiments on real-life Java inputs to evaluate the practical relevance of our approach.

References

[1] T. Fischer, J. Niere, L. Torunski & A. Zündorf (2000): *Story Diagrams: A New Graph Grammar Language Based on the Unified Modelling Language and Java*. In: *TAGT*, *LNCS* 1764, Springer, pp. 157–167.

[2] F. Heidenreich, J. Johannes, M. Seifert & C. Wende (2010): *Closing the Gap between Modelling and Java*. In: *Software Language Engineering*, *LNCS* 5969, Springer, pp. 374–383.

[3] G. Kulcsár, S. Peldszus & M. Lochau: *Case Study: Object-oriented Refactoring of Java Programs using Graph Transformation*. In: *Transformation Tool Contest 2015*. Available at `https://github.com/Echtzeitsysteme/java-refactoring-ttc/`.

[4] E. Leblebici, A. Anjorin & A. Schürr (2014): *Developing eMoflon with eMoflon*. In: *Theory and Practice of Model Transformations*, *LNCS* 8568, Springer, pp. 138–145.

[5] S. Peldszus, G. Kulcsár & M. Lochau (2015): *SHARE Image*. Available at `http://is.ieis.tue.nl/staff/pvgorp/share/?page=ConfigureNewSession&vdi=XP-TUe_TTC15-Refactoring.vdi`.

[6] S. Peldszus, G. Kulcsár & M. Lochau (2015): *Source Code at GitHub*. Available at `https://github.com/SvenPeldszus/GravityTTC`.

[7] S. Peldszus, G. Kulcsár, M. Lochau & S. Schulze (2015): *Incremental Co-Evolution of Java Programs Based on Bidirectional Graph Transformation*. In: *PPPJ'15*, ACM, NY, USA, pp. 138–151.

[8] A. Schürr (1994): *Specification of Graph Translators with Triple Graph Grammars*. In: *20th Int. Workshop on Graph-Theoretic Concepts in Computer Science*, *LNCS* 903, Springer, pp. 151–163.

[3]https://eclipse.org/MoDisco/