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 for the Java Refactoring case study of the Transformation Tool Contest (TTC 2015) is implemented using eMoflon (`www.emoflon.org`), a meta-modeling and model transformation tool developed at the Real-Time Systems Lab of TU Darmstadt.

The solution, available as a virtual machine hosted on Share and at GitHub [4], includes a bidirectional synchronization between a Java model and an abstract program graph specified using Triple Graph Grammars (TGG) and 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 the solution developers, taking as input Java source code given in textual form and producing a refactored version of it as output. We participate in the extended version of the challenge, where the developer is forced to use a given meta-model, called the Program Graph (PG), for the abstract representation of the input Java program and 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 propagating the changes back to Java source code. However, as our tool eMoflon supports both EMF meta-modeling and a way to specify bidirectional transformations using *Triple Graph Grammars* (TGGs), we found that this qualifies us to deal with the extended challenge. To be more specific, TGGs [5] 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 capability is employed in the solution, namely, the support of the *Story Driven Modeling* (SDM) [1] language, which is a visual language for expressing programmed graph rewriting operations. An SDM method consists of a set of graph transformation rules with an additional control flow to describe their execution order dependencies in an imperative fashion. As the refactoring part of the challenge comprises an endogenous graph transformation without the need of bidirectionality, SDMs seem as a convenient choice for implementing the actual refactoring operations.

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. In the solution, 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

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

Submitted to:

which arise at each step and evaluate the solution according to the evaluation aspects given in the case study.

2 The Solution using eMoflon, TGGs and SDM

We start with describing the overall workflow of the solution.

1. **Java to JaMoPP.** The Java source code is parsed and converted into an intermediate EMF representation using the JaMoPP framework [2].
2. **Preprocessing JaMoPP for TGGs.** For the transformation with TGGs using eMoflon, we have to resolve some critical issues in the generated JaMoPP model: (i) parsing the namespaces to build up the package structure and (ii) turning the parameter nodes of a method node into a double-linked parameter list, as the non-deterministic fashion of TGG transformations could not guarantee the preservation of their order.
3. **Preprocessed JaMoPP to PG.** The PG is generated from the JaMoPP model of the program by using the forward transformation part of our specified TGG.
4. **Refactoring on the PG.** The refactorings are implemented using our dialect of *Story Driven Modeling* (SDM), which is a visual language for programmed graph rewriting supported by eMoflon.
5. **PG to JaMoPP.** Using the backward transformation resulting from our single TGG specification, we derive the refactored JaMoPP model from the modified PG based on a calculation of the refactoring delta.
6. **JaMoPP to Java.** The resulting JaMoPP model is easily converted back into Java code using the JaMoPP framework.

In the following subsections, we give a detailed description of the steps above.

2.1 Java to JaMoPP

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] We identified this task as the first step of our solution chain. JaMoPP supports the complete Java 5 language, i.e., all possible input Java sources within the scope of the case study (conforming to a restricted variant of Java 4) can be translated using JaMoPP. As the translation of Java code to an EMF model belongs to the central functionality of the tool, which has been already used successfully in various projects, this step is executed without any problems for all inputs.

2.2 Preprocessing JaMoPP for TGGs

While working with the JaMoPP metamodel for Java, we have found out that some parts of it do not comply with the PG metamodel and with some properties of the planned TGG translation. The following two preprocessing actions has to be taken to make the JaMoPP model instance more fit to the transformation into a PG instance.

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

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 as clean and concise as possible.

Putting the method parameters into a double-linked list. A transformation specified by a set of TGG rules is per definition non-deterministic, i.e., if the source side of a rule matches with some similar elements of the 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 parameter list into a double-linked list so that the parameter nodes can only be processed in the given order (with a TGG rule specified appropriately).

Consequently, at a later phase of the overall transformation, an analogous inverse processing has to be performed on the result of the TGG synchronisation before converting the JaMoPP model back into Java source code; this step consists in reverting the changes mentioned above and we omit its detailed presentation later on.

2.3 Preprocessed JaMoPP to PG

As already mentioned before, this step of our transformation chain is implemented using TGGs. TGGs describe a correspondence between instances of a *source* and a *target* metamodel. This correspondence between given elements of the two metamodels is 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, unprocessed before the rule application and matching to the corresponding part of the rule.

Our TGG specification consists of 20 rules. We have identified 7 main source model components which the transformation has to deal with:

- *Initialization.* This is a single rule which initializes the transformation by creating the root of the PG.
- *Packages.* Two rules are responsible for initializing the package structure root and for recursively creating the child nodes, respectively.
- *Classes.* There is one rule for classes and one for handling inheritance relations.
- *Methods.* There are six rules which deal with methods: Three rules are used for creating the three method parts on the target side and the other three rules are used to create the necessary references in the PG There are another two main components that handle further aspects of methods:

  Return types. Three methods handle the return types of the methods: the first translates user-defined return types, the second creates return type nodes for library types and the third one links those which already exist.

  Parameters. This is the biggest main component with 6 rules (3 for parameter names, 3 for parameter types): (i) creating new parameters, (ii) linking the first parameter of the list, (iii) linking the next parameter of a list (iterating through a double-linked parameter list), (iv) translating user-defined parameter types, (v) creating parameter type nodes for library types and (vi) linking existing parameter type nodes.

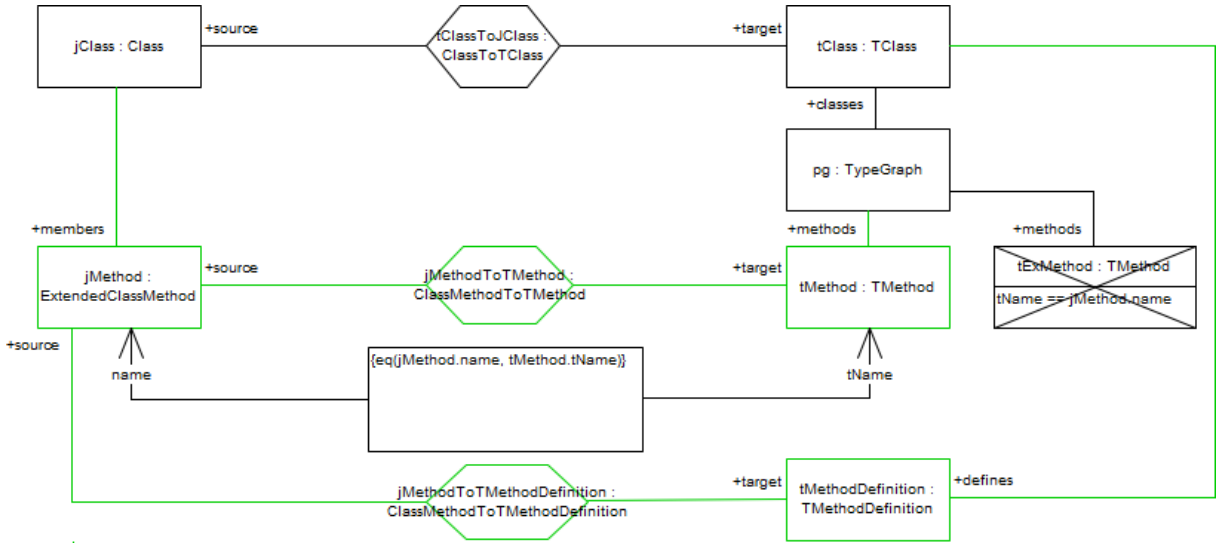

Figure 1: Example TGG rule `MethodNameCreate`

- *Fields.* There is one rule to create a new field and another one to process already existing ones.

Here, we do not have the possibility to show and explain all the rules; nevertheless, in Figure 1, we show a typical rule `MethodNameCreate`, introduce our visual TGG syntax and give an idea about the rule semantics.

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. The black boxes represent the context (elements which have to be present for the rule to be applied) and green boxes are the elements created by the rule application. A crossed-out box means a negative application condition: the object can 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.

2.4 Refactoring on 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 transformations compared to TGGs.

SDMs provide a way to implement methods of classes of a metamodel (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 metamodel. 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 will omit an elaborate presentation of all our SDM methods; as SDMs excel at being intuitively accessible for everyone, we think that after having explained the syntax and semantics on an excerpt of our SDMs, it is more convenient for the interested reader to explore the SDM diagrams on their own. Nevertheless, we show an example method, introduce or visual SDM syntax and give an intuition about how the method works.

Figure 2 shows an example SDM method `csc_Perform` which is the actual execution of `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 are the story nodes; each story node contains a graph transformation rule which is applied as the containing story node is activated. A rule application consists of finding a match for the depicted graph pattern in the model where the SDM method has been called, deleting the red-colored elements and creating the green-colored ones. Boxes with a thick edge mean bound object variables that are matched to a fix object in the model. There is a possibility for a story node to 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 as seen in the middle. Cascaded-style boxes represent `for each` loops, where the rule is applied to each possible match in the model with an optional loop body executed after each match (through an `Each Time` edge). After all the matches have been processed, the loop is left through a mandatory `End` edge.

The depicted rule, `csc_Perform`, roughly does the following (with omitting technical implementation details): 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.

2.5 PG to JaMoPP

As our TGG describes both a forward (cf. Section 2.3) and a backward transformation, this step of the transformation requires no extra development efforts. TGGs in eMoflon provide a synchronization 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.

In our solution, we do not perform any modifications on the original copy of the PG until we have not checked and performed all the prescribed refactorings on a copy of the PG, while doing a bookkeeping of those refactoring operations actually performed. The synchronization method is not called until the refactoring instructions have been performed and the synchronization instruction comes. At this point, the modifications are performed on the original PG as well using our bookkeeping, and this instance serves then as a basis for the delta calculation to synchronize the JaMoPP model with the PG.

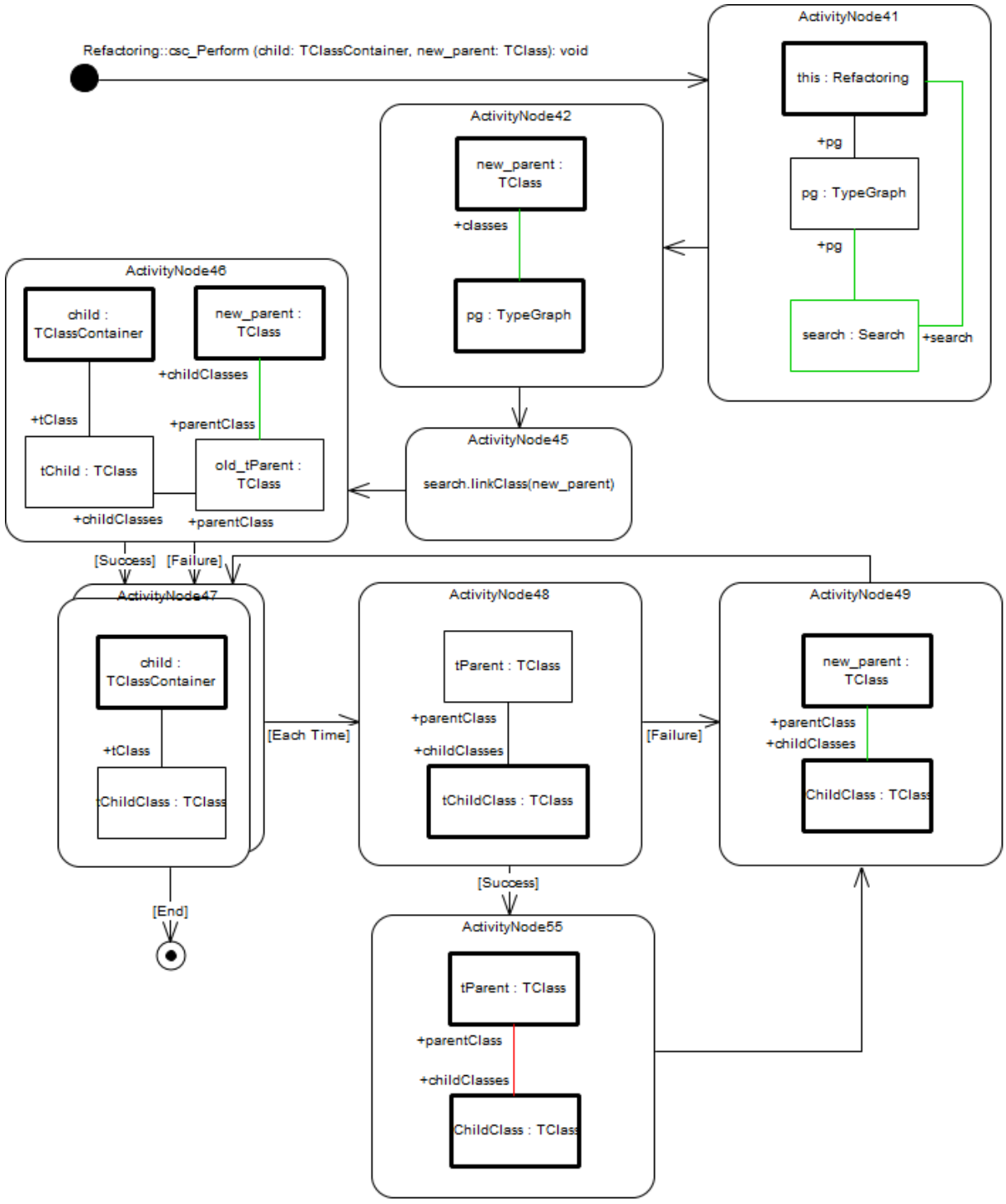

Figure 2: Example SDM method `csc_Perform`

2.6 JaMoPP to Java

Similar to the first step (cf. Section 2.1), the translation of the EMF model to Java code also belongs to the central functionality of the tool, this step does not pose any problems.

3 Evaluation

3.1 Correctness and Performance

Our solution handles successfully all *Create Superclass* refactorings. Whether the execution of a *Pull-Up Method* refactoring is possible is always correctly determined in the available test cases. In the generated code with *Pull-Up Method* refactorings are sometimes defect method calls, where the method name got lost during synchronization.

When it comes to performance, the declarative nature of TGG rules and the resulting expensive transformation algorithm has a negative influence on the execution time of our transformation. The execution of an SDM method (actually, the Java code which is automatically generated from the SDM specification) can also take longer compared to the same functionality implemented directly in Java by an experienced developer.

3.2 Soft Aspects

In the following, we comment on some soft aspects of our tooling used for the solution to provide a better insight on some strong and weak spots.

- **Comprehensibility.**

  TGGs. *Comprehending:* the concept of TGGs can be rather exotic at the first sight. It requires a special mindset, which costs a few hours for the newcomer. After getting used to the rule syntax, a given TGG becomes easily understandable. *Developing:* For a newcomer, the first steps with TGGs require some extra efforts with a lot of trial-and-error experimentation. Usually, for non-experts, most of the development time is spent with the TGG specification whenever TGGs are involved (as in the case of present solution).

  SDMs. *Comprehending:* SDMs are designed to be an intuitive method implementation technique. SDMs with a proper layout are therefore easily understandable. *Developing:* according to the intuitive nature of SDMs, learning how to develop with SDMs also requires moderate efforts.

- **Readability.** eMoflon consists of two plug-ins: one for Eclipse with an own perspective and one for Enterprise Architect for the visual specification of TGGs and SDMs. We think that both user interfaces provide good readability as they are well-known and well-designed.

- **Communication with the user.** Both plug-ins provide a validation option during design- as well as compile-time with informative error messages and warnings.

- **Robustness.** We think that in our case, robustness is more relevant to the TGG part. Here, unsupported elements (not in the rules) cause no problems as they are ignored by the TGG engine. Elements which are in the rules but in another context can cause the transformation to fail - in this case, an informative error message is provided. However, the detection of such cases requires a very large test set which may not be given and is also hard to design.

- **Extensibility.** TGGs have a modular nature, i.e., adding new Java language elements only requires specifying additional rules without changing the already existing ones. The SDM part is trivially extensible with a corresponding method if, e.g., a new refactoring should be supported.
- **Debugging.** For the TGG as well as the Java part, Java code is generated. In case of an error, this Java code can be effectively debugged with the standard Java debugging tools. Where necessary the generated Java code contains comments to identify the corresponding parts of the visual specification.

As a summary, utilizing TGGs for the synchronization part is responsible for the greatest advantages and disadvantages at once. TGGs provide a powerful declarative language, where the correctness of the transformation is proven by construction of the grammar itself. 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 what we have to pay for those formal and algorithmic properties is the slower execution time compared to task-optimized, imperative solutions. Another problem can arise if, e.g., a developer who is not a TGG expert might want to extend the solution to support more Java elements; in this case, an additional TGG rule which overlaps with an earlier one may cause translation problems (because of inherent nondeterminism) that are hard to debug.

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 cannot be as optimal as an equivalent hand-written implementation from an experienced Java developer.

We would also like to mention that during the development of our TGG rule set, it turned out that the interworking between JaMoPP and our TGGs is somewhat inharmonious. Although detailing those issues would go beyond the scope of present paper, it is our first identified task for the future to examine the possible alternatives of JaMoPP for this transformation scenario.

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 can 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.

As already mentioned, our future work includes, first of all, the examination of other tools with similar functionality to JaMoPP in order to potentially reduce the need for pre- and postprocessing and to be able 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] Thorsten Fischer, Jörg Niere, Lars Torunski & Albert Zündorf (2000): *Story Diagrams: A New Graph Grammar Language Based on the Unified Modelling Language and Java*. In Hartmut Ehrig, Gregor Engels, Hans-

Jörg Kreowski & Grzegorz Rozenberg, editors: *Proceedings of the Sixth International Workshop on Theory and Application of Graph Transformations (TAGT1998)*, 1764, Springer, Paderborn, Germany, pp. 157–167.

[2] Florian Heidenreich, Jendrik Johannes, Mirko Seifert & Christian Wende (2010): *Closing the Gap between Modelling and Java*. In Mark van den Brand, Dragan Gaevi & Jeff Gray, editors: *Software Language Engineering*, *Lecture Notes in Computer Science* 5969, Springer Berlin Heidelberg, pp. 374–383.

[3] Géza Kulcsár, Sven Peldszus & Malte 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] Sven Peldszus, Géza Kulcsár & Malte Lochau (2015): *Sources of a Solution to the Java Refactoring Casestudy using eMoflon*. Available at `https://github.com/SvenPeldszus/GravityTTC`.

[5] Andy Schürr (1994): *Specification of Graph Translators with Triple Graph Grammars*. In E. Mayr, G. Schmidt & G. Tinhofer, editors: *20th Int. Workshop on Graph-Theoretic Concepts in Computer Science*, *Lecture Notes in Computer Science (LNCS)* 903, Springer Verlag, Heidelberg, pp. 151–163.