

Specifying and Analyzing Program Refactorings With AGG

Javier Pérez¹, Olga Runge², and Gabriele Taentzer³

¹ Universidad de Valladolid, Spain, jperez@infor.uva.es

² Technische Universität Berlin, Germany, olga@cs.tu-berlin.de

³ Philipps-Universität Marburg, Germany
taentzer@mathematik.uni-marburg.de

Abstract. AGG is a general development environment for graph transformation systems which we use here to specify and analyze program refactorings. In this paper, we consider two commonly used refactorings, namely MoveMethod and EncapsulateField, to show our approach.

1 What is AGG?

The AGG tool environment [2] consists of a graph transformation engine, several analysis tools for graph transformations and a graphical user interface for convenient user interaction. AGG supports the algebraic approach to typed, attributed graph transformation. It provides a typing concept for nodes and arcs which supports node type inheritance. Its attribution concept is based on Java expressions. Transformation rules may be equipped with positive and negative application conditions. Rule applications may be controlled by graph constraints and explicit control constructs such as layered graph transformation. Analysis tools offer graph parsing, critical pair analysis and applicability checks for transformation rules, as well as checking of termination criteria for controlled rule applications.

2 Program Graph

The given program graph is quite large and it is not that easy to overview it. For a better orientation, the graph can be zoomed out by factor 0.5. All attributes can be automatically hidden. Zooming in again, the attributes show up again and we can decide which attributes are interesting to see. Furthermore, orientation in the program graph is supported by distinguishing different node and edge types not only by names but also by colors.

3 Specifying Refactorings

3.1 Refactoring Preparation

Before the following refactorings can be performed, the program graph has to be prepared by computing all transitive closures of class inheritance, update,

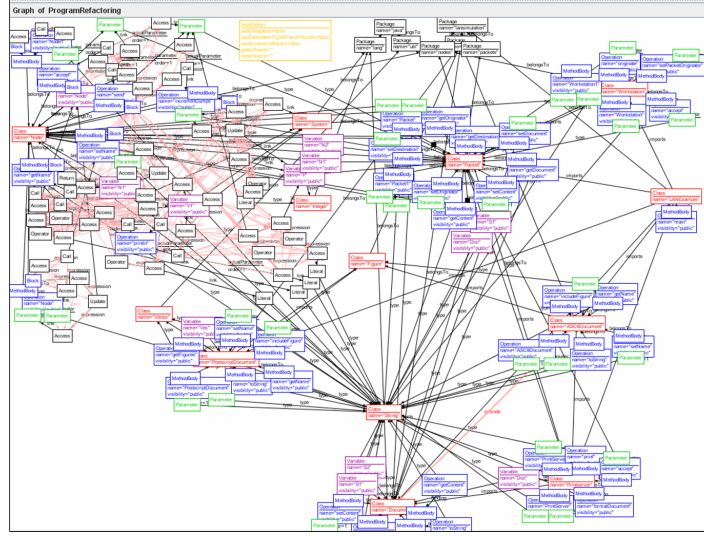


Fig. 1. The start graph

access and call hierarchies. For each kind of hierarchy we need two rules to be applied as long as possible to compute the transitive closure. This task has to be done only once in the beginning of a refactoring phase. Thereafter it has to be kept consistently with hierarchy modifications, of course. The inverse rules can be used to delete additional “tgen” edges after a refactoring process.

3.2 Refactoring “MoveMethod”

This refactoring is rather complex such that several rules are needed for specifying it, especially due to complex update actions. As a simple form of control flow, we use rule layers which means that rules of the lowest layer are applied first as long as possible, then the next layer is considered, and so on until the highest layer has been reached. All rules of layer 1 prepare the refactoring and check all preconditions, rule “moveMethod” of layer 2 perform the proper refactoring, rules of layer 3 do all updates, and layer 4 rules are used to clean up after refactoring. (See the left column of rules in Fig. 2 to get an overview.) If transitive closures shall be built up and deleted for each refactoring separately, two additional rule layers come along (see Subsection 3.1).

To perform this refactoring interactively, the trigger rule of the preparing layer, namely “enableMoveMethod” (see Figure 3), has input parameters to determine where the refactoring shall be applied. All rules following up use a node of type “InputOption” which stores parameter values and is connected to the method to be moved and to both classes that participate. Moreover, this rules checks for necessary pre-conditions of the refactoring, all listed as NACs in Figure 2. NAC “noCallToSuper” for example, checks that the method to be moved



Fig. 2. Overview on rules for Refactorings “MoveMethod” and “Encapsulated Field”

does not call its super method. All other NACs prohibit the existence of a method with the same name in the new class and all its super and subclasses.

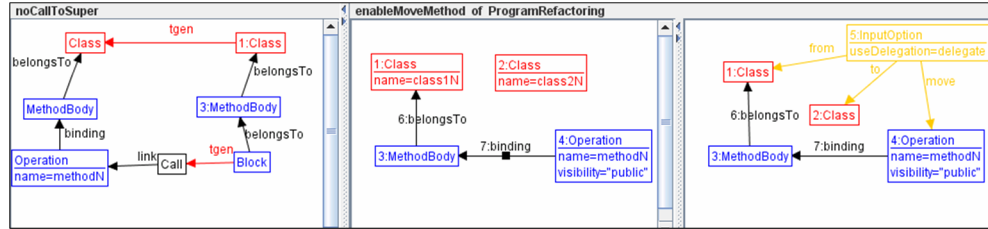


Fig. 3. Rule “EnableMoveMethod”

After checking all preconditions, rule “MoveMethod” (see Figure 4) is applied. If there is a variable in the source class which references the target class, this variable shall be used to update the call access to the moved method. After updating, the access is performed over this variable (see Figure 5).

3.3 Refactoring “EncapsulatedField”

Refactoring “EncapsulatedField” is specified similarly by first enabling this refactoring, then producing or adapting getter and setter methods independently, updating getters and setters, and last but not least finalizing the encapsulation refactoring.

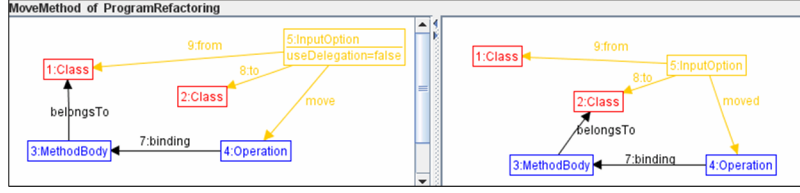


Fig. 4. Rule “MoveMethod”

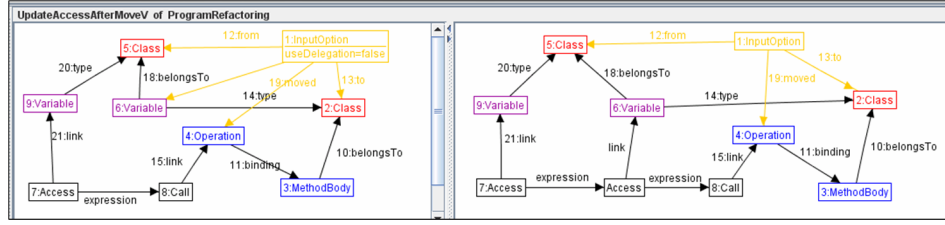


Fig. 5. Rule “UpdateAccessAfterMoveV”

In the following, we consider Rule “EncapsulateFieldNewGetter” which is depicted in figure 6. After having checked that there is not already a getter method in the hierarchy of the current class, a new getter method is created. Its visibility kind has to be larger or equal to the visibility of the encapsulated variable.

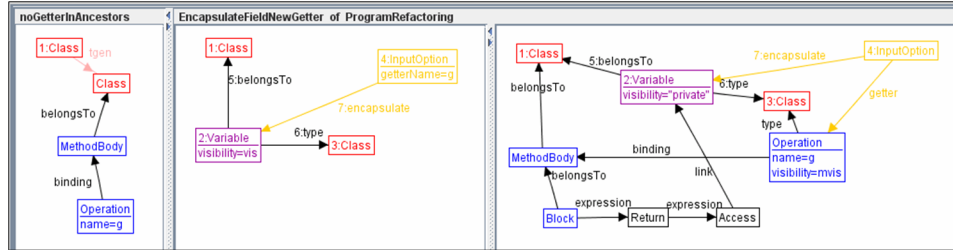


Fig. 6. Rule “EncapsulateFieldNewGetter”

4 Analyzing Refactorings

Having specified the refactorings, we can start to analyze them. Since both refactorings are quite complex and have to be specified by a number of rules which are applied in a controlled manner, we are interested in the applicability of a complete refactoring. As a prerequisite we check for dependencies of later rules from former ones. One example is a produce-use dependency of rule

“MoveMethod” from Rule “EnableMoveMethod”. Of course, the input options have to be captured before they are further used. (See the left graph of Figure 7 for this dependency.) Furthermore, rule should not have impeding predecessor rules in the layered execution. Rule “enableMoveMethod” for example does not cause a conflict with rules following up.

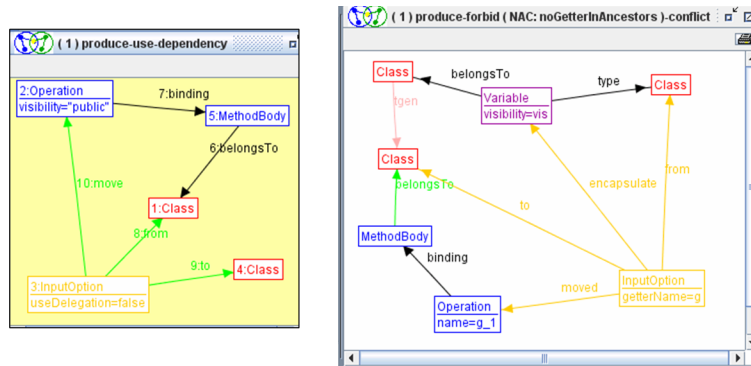


Fig. 7. A dependency and a conflict graph

Furthermore, we are interested in performing refactorings in parallel by different developers. (See [1] for more details.) In this case, we have to analyze conflicts between rules of one refactoring with rules of another. Checking for example refactorings “MoveMethod” and “EncapsulateField”, we find e.g. conflicts between rule “MoveMethod” and rule “EncapsualteFieldNewGetter”. The right graph in Figure 7 shows a produce-forbid conflict between these two rules. Rule “MoveMethod” produces a new “belongsTo” edge which is forbidden by Rule “EncapsualteFieldNewGetter”, since a method can be moved to a superclass with the same name as the getter method to be newly created. Please keep in mind that AGG computes all potential conflicts between rules. Considering a concrete transformation sequence, such a conflict does not have to show up.

References

1. T. Mens, G. Taentzer, and O. Runge. Analysing refactoring dependencies using graph transformation. *Software and System Modeling*, 6(3), Sept. 2007. 269-285.
2. G. Taentzer. AGG: A Graph Transformation Environment for Modeling and Validation of Software. In J. Pfaltz, M. Nagl, and B. Boehlen, editors, *Application of Graph Transformations with Industrial Relevance (AGTIVE'03)*, volume 3062 of *Lecture Notes in Computer Science*, pages 446 – 456. Springer, 2004.