Detecting Structural Refactoring Conflicts Using Critical Pair Analysis

Tom Mens [1]

Software Engineering Lab
Université de Mons-Hainaut
B-7000 Mons, Belgium

Gabriele Taentzer and Olga Runge [2]

Technische Universität Berlin
D-10587 Berlin, Germany

Abstract

Refactorings are program transformations that improve the software structure while preserving the external behaviour. In spite of this very useful property, refactorings can still give rise to structural conflicts when parallel evolutions to the same software are made by different developers. This paper explores this problem of structural evolution conflicts in a formal way by using graph transformation and critical pair analysis. Based on experiments carried out in the graph transformation tool AGG, we show how this formalism can be exploited to detect and resolve refactoring conflicts.

Key words: refactoring, restructuring, graph transformation, critical pair analysis, evolution conflicts, parallel changes

1 Introduction

Refactoring is a commonly accepted technique to improve the structure of object-oriented software [2]. Nevertheless, there are still a number of problems if we want to apply this technique in a collaborative setting, where different software developers can make changes to the software in parallel.

To illustrate these problems, consider the scenario of a large software development team, where two developers independently decide to refactor the same software. It is possible that these parallel refactorings are incompatible,

[1] Email:tom.mens@umh.ac.be
[2] Email:gabi@cs.tu-berlin.de

This is a preliminary version. The final version will be published in Electronic Notes in Theoretical Computer Science
URL: www.elsevier.nl/locate/entcs

in the sense that they cannot be combined together. As an example, assume that a *Move Variable* refactoring and an *Encapsulate Variable* refactoring are applied in parallel to the same variable in the same class. Both refactorings are clearly in conflict since they cannot be serialised as they both affect the same variable in different incompatible ways.

It is also possible that two parallel refactorings can only be combined in a particular order. As an example, assume that a *Rename Variable* refactoring and an *Encapsulate Variable* refactoring are applied in parallel to the same variable in the same class. One can decide to rename the variable first, and then encapsulate it, but not the other way round. The reason is that the encapsulation introduces an auxiliary setter and getter method whose names rely on the variable name.

To address the problems illustrated above, we propose to take a formal approach based on *graph transformation* and *critical pair analysis* [1,4,5]. We will perform a feasibility study using the AGG tool. As such, the contribution of our paper will be twofold:

- to show the feasibility of the technique of critical pair analysis for a new practical application;
- to support refactoring tool developers with a formal means to analyse the consistency of refactoring suites, and to allow them to identify unanticipated dependencies between pairs of refactorings.

2 The AGG tool

We decided to use the tool AGG (see `http://tfs.cs.tu-berlin.de/agg`) for our experiments. It is the only graph transformation tool we are aware of that supports critical pair analysis, a crucial ingredient of our approach towards the detection of refactoring conflicts.

2.1 Specifying graph transformations

To reason about object-oriented software evolution, we specify object-oriented programs as graphs, that have to respect the constraints specified by a *type graph*. This type graph acts as an object-oriented metamodel. The metamodel we expressed in AGG is shown in Figure 1. It expresses the basic object-oriented concepts (such as classes, methods and variables), their attributes (such as name and visibility), and their relationships (such as inheritance, containment and typing) with associated multiplicities. We deliberately decided to use this simple metamodel instead of a full-fledged one, because our goal was to perform a feasibility study.

Representative refactorings are expressed as graph transformations using this metamodel. A *graph transformation* $t : G \xrightarrow[p(m)]{} H$ is defined as a pair consisting of a *graph transformation rule* $p : L \to R$ and a *match* $m : L \to G$.

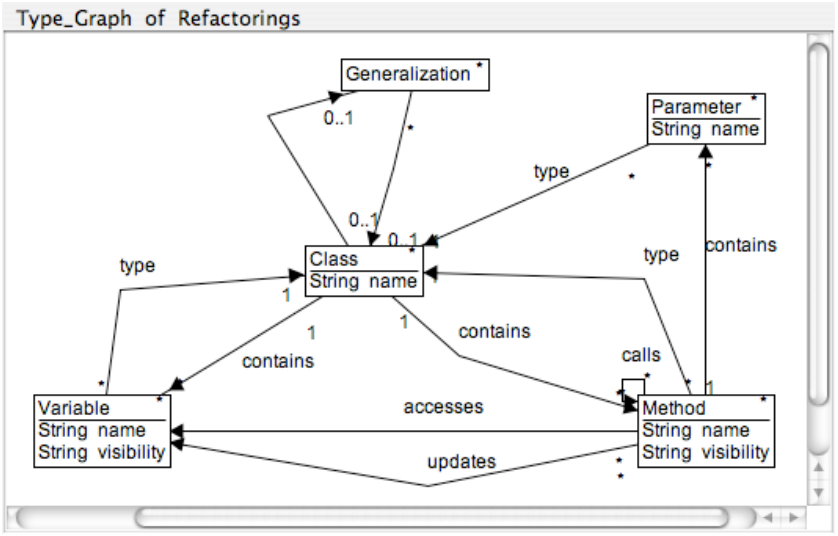


Fig. 1. Type graph representing the object-oriented metamodel.

The rule p specifies how its left-hand side (LHS) L has to be transformed into its right-hand side (RHS) R. The match m specifies an occurrence of this LHS in the graph that needs to be transformed. Note that there may be more than one possible match. As shown in [5], one can easily extend this definition to come to a notion of *typed graph transformations* that respect the type constraints imposed by the type graph.

As a concrete example, the transformation *Encapsulate Variable* in Figure 2 can be applied to a class containing a variable of a particular type. After the transformation, a setter method and getter method are added to the class, but the rest of the structure is preserved. This is visualised by assigning numbers 1 to 5 to nodes and edges in the LHS and RHS. Nodes and edges that have the same number in the LHS and RHS are preserved by the transformation. All nodes and edges in the RHS that do not have a number assigned (such as the setter and getter method) are newly introduced.

Note that the graphs we use are attributed, i.e., the nodes in the graph may contain attributes whose values may be modified by the transformation. This is for example the case in Figure 2 with the attribute `visibility` of variable node `1`, whose value is modified from `public` to `private`.

Another useful feature of AGG is the possibility to specify *negative application conditions* (NACs) [3] that capture the preconditions of a transformation. These NACs can be considered as a kind of forbidden subgraphs. For example, the NAC *No Setter* for transformation rule *Encapsulate Variable* in Figure 2 expresses that the class containing the variable to be refactored must not contain a setter method for this variable, since this method will be added by the transformation. To express this, we need to specify an attribute condition which relates the name of the method in the NAC to the corresponding one in the RHS. The rule *Encapsulate Variable* contains a second NAC that forbids the existence of a getter method for the variable to be encapsulated.

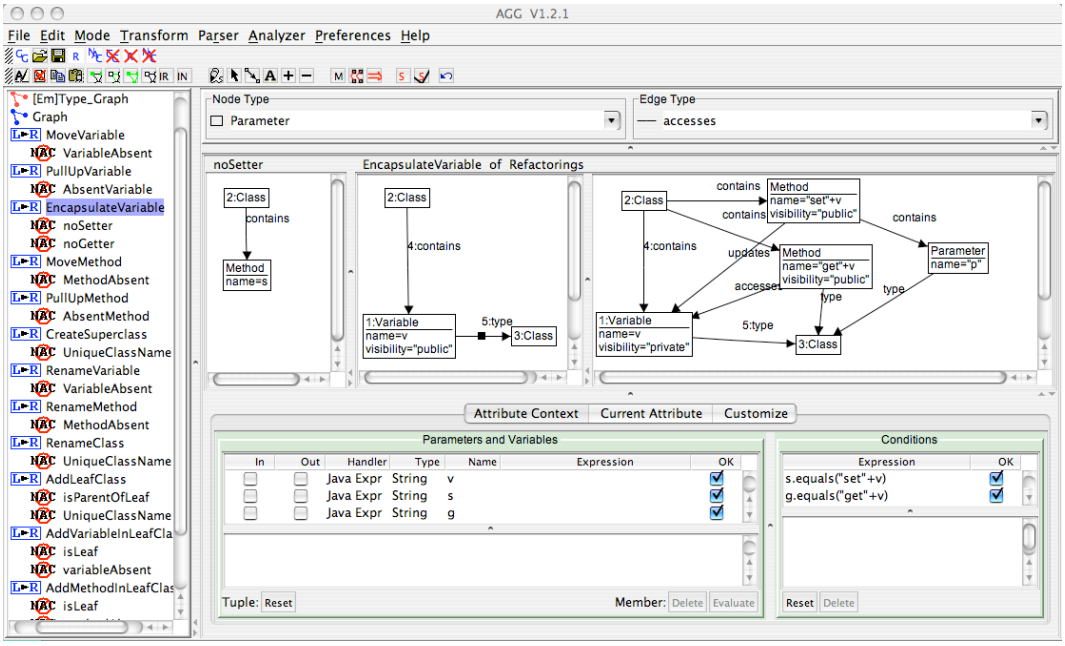


Fig. 2. The AGG tool in action. In the left pane, all refactorings specified as graph transformations are listed, together with their NACs. On the right of it, the specification of the *Encapsulate Variable* refactoring is given as a graph transformation rule with a NAC *No Setter*, a left-hand side, and a right-hand side. The Attribute Context for the Method attribute `name` in the bottom panes specifies the additional relation that its value `s` in the NAC must be equal to `"set"+v` in the RHS.

2.2 Critical pair analysis

Critical pair analysis was first introduced in term rewriting, and has been generalized to graph rewriting later. A critical pair formalises the idea of a minimal example of a potentially conflicting situation. Given two transformations $t_1 : G \xrightarrow[p_1(m_1)]{} H_1$ and $t_2 : G \xrightarrow[p_2(m_2)]{} H_2$, t_1 has an *asymmetric conflict* with t_2 if it can be performed before, but not after t_2. If the two transformations disable each other in any order, they have a *symmetric conflict*.

The reasons why rule applications can be conflicting are threefold:

(i) One rule application deletes a graph object which is in the match of another rule application.

(ii) One rule application generates graph objects that give rise to a graph structure that is prohibited by a NAC of another rule application.

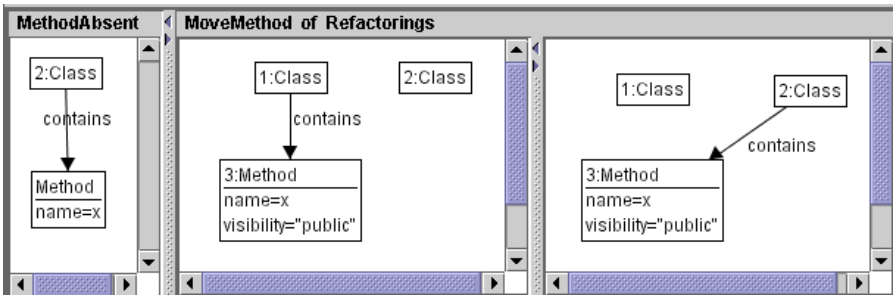
(iii) One rule application changes attributes being in the match of another rule application.

To find all conflicting rule applications, minimal critical graphs are computed to which rules can be applied in a conflicting way. Basically we consider all overlapping graphs of the left-hand sides of two rules with the obvious matches and analyze these rule applications. All conflicting rule applications

thus found are called *critical pairs*. If one of the rules contains NACs, the overlapping graphs of one LHS with a part of the NAC have to be considered in addition.

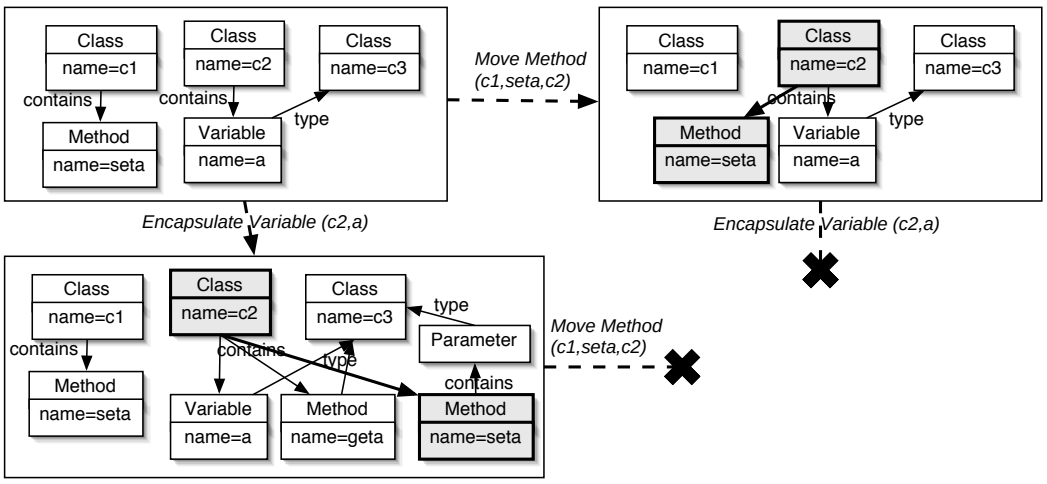
AGG supports the critical pair analysis for typed attributed graph transformations. Given a set of graph transformation rules, it computes a table which shows the number of critical pairs for each pair of rules. The number of detected critical pairs for transformation rules can be reduced drastically if there is a type graph with multiplicity constraints (as in Figure 1). Upper bounds of the multiplicity constraints are then used to reduce the set of critical pairs by throwing out the meaningless ones.

Fig. 3. Graph transformation rule for *Move Method*.



As a concrete example, let us compute the critical pairs between the graph transformation rules *Encapsulate Variable* (of Figure 2) and *Move Method* (shown in Figure 3). There is a symmetric conflict between both rules, and the number of computed critical pairs in both cases is 2. Figure 4 illustrates this graphically.

Fig. 4. Example of a symmetric conflict between graph transformations *Move Method* and *Encapsulate Variable*



If we move a method to a class in which we want to encapsulate a variable afterwards, there is a first critical pair that represents the conflict that the name of the method that is moved coincides with the name of the setter method that needs to be introduced by *Encapsulate Variable*. The second critical pair, which is very similar, represents a name conflict with the getter method.

The other way around, if we first apply the *Encapsulate Variable* transformation, we get a similar situation. *Move Method* cannot be applied when the method needs to be moved to the class of the encapsulated variable, and the method name coincides with either the name of the setter method or the name of the getter method.

3 Specification of refactorings

To be able to detect conflicts between refactorings applied in parallel by different software developers, we specified some representative refactorings identified by Fowler [2] as typed attributed graph transformations. The preconditions of the refactorings were directly expressed as negative application conditions on the graph transformations.

- *Encapsulate Variable* takes a public variable in a class and replaces it by a private variable with two public accessor methods. One for getting the value of the variable, and one for setting its value. The graph transformation rule for this particular refactoring is shown in Figure 2;
- *Move Method* moves a public method from a class to another class, not necessarily belonging to the same inheritance hierarchy. The graph transformation rule is shown in Figure 3.
- *Move Variable* moves a public variable from a class to another class, not necessarily belonging to the same inheritance hierarchy. The graph transformation rule is very similar to the one for *Move Method*.
- *Pull Up Variable* moves a public or protected variable from a class to a superclass that resides one level up the inheritance hierarchy. The graph transformation rule is shown in Figure 5.
- *Pull Up Method* moves a public or protected method from a class to a superclass that resides one level up the inheritance hierarchy. The graph transformation rule is similar to the one for *Pull Up Variable*.
- *Create Superclass* creates an intermediate abstract superclass for a given class. The graph transformation rule is shown in Figure 6.
- *Rename Method* changes the name of a method in a class to a new one which is unique within this class. The graph transformation rule is shown in Figure 7.
- *Rename Variable* changes the name of a variable in a class to a new one which is unique within this class. The graph transformation rule is similar

to the one for *Rename Method*.

- *Rename Class* changes the name of a class to a new unique name. The graph transformation rule is similar to the one for *Rename Method*.

Fig. 5. Graph transformation rule for *Pull Up Variable*.

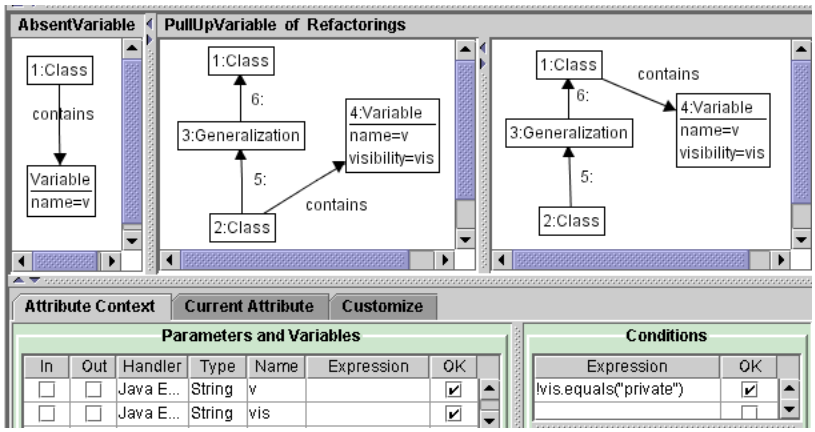
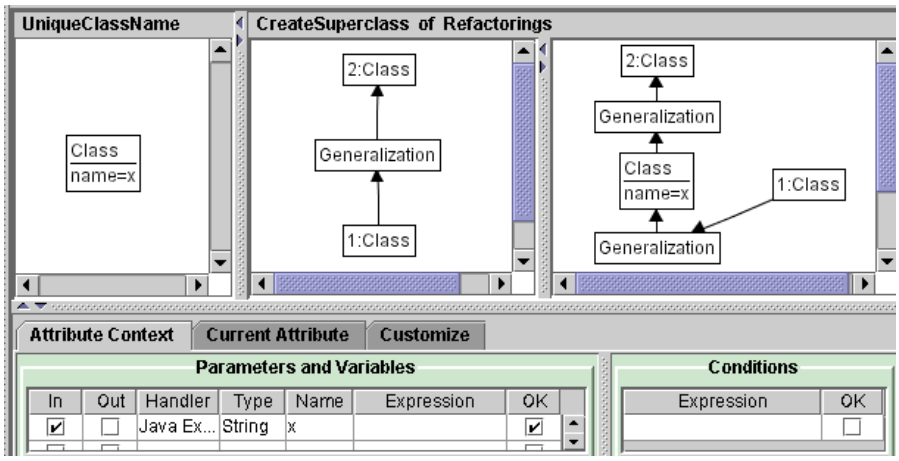
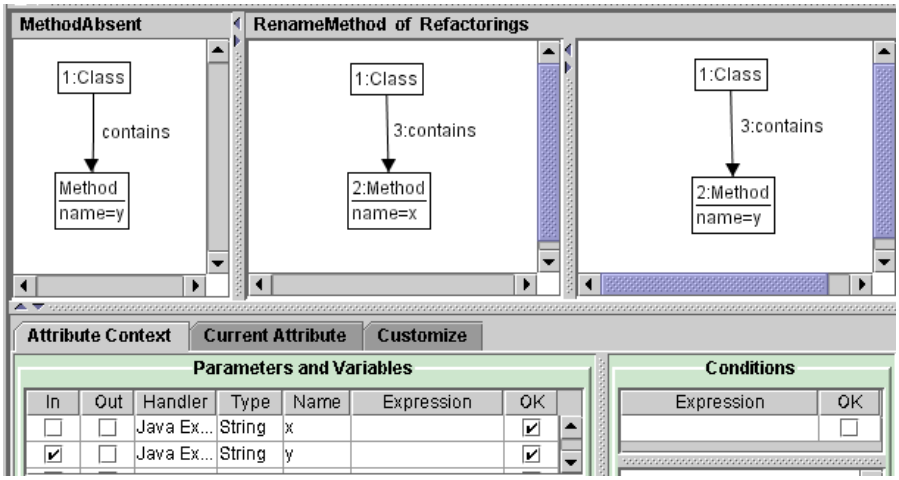


Fig. 6. Graph transformation rule for *Create Superclass*.



One should note that we deliberately did not implement all details of each refactoring in our graph transformations, since it was not our intent to build a full-fledged refactoring tool, but rather to perform a feasibility study that would show that the most important conflicts between parallel refactorings can be detected by critical pair analysis. For example, we decided to restrict *Create Superclass*, *Pull Up Variable* and *Pull Up Method* to a single subclass rather than a set of subclasses. We also did not express all necessary preconditions for each refactoring, as this would only make the analysis more difficult and computation intensive. Although, in theory, these simplifications may lead to false negatives during conflict detection, in practice, it turned out that all of

Fig. 7. Graph transformation rule for *Rename Method*.



the conflicts we expected to occur were actually detected, as we will show in the next section.

4 Analysis of refactoring conflicts

We applied the critical pair analysis algorithm of AGG to our selection of 9 representative refactorings. We observed that, for many pairs of refactorings, duplicate critical pairs were reported for the same conflict. We even found some bugs in the initial critical pair analysis algorithm. Therefore, we improved the algorithm so that it reports only those critical pairs that actually correspond to distinct conflicts. The results of this improved algorithm are shown in Figure 8. All critical pairs can be considered in detail on the AGG Web page.

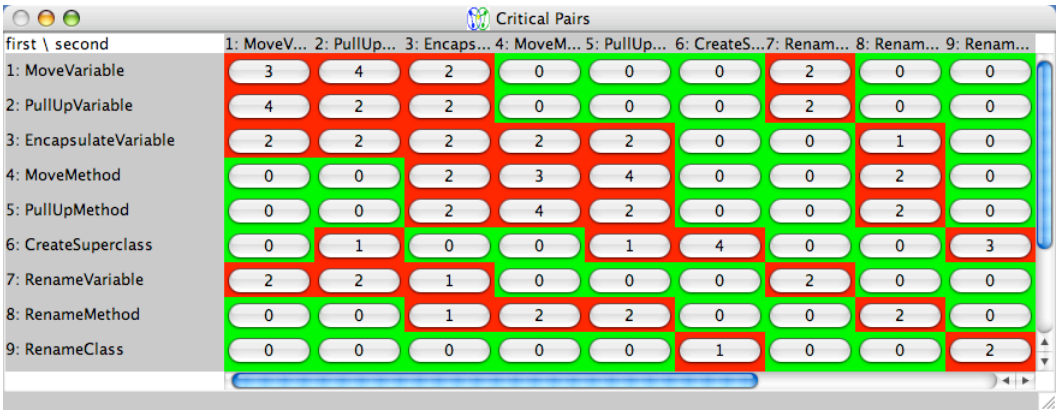


Fig. 8. Critical pair analysis of the refactoring transformations.

The obtained results correspond to what we expected. For example, we

expected a certain similarity between the conflicts generated by *Move Method* and *Pull Up Method* (resp. *Move Variable* and *Pull Up Variable*) since they both move a method (resp. variable) to another location. We also expected similar conflicts for *Move Variable* and *Move Method*, as well as for *Pull Up Variable* and *Pull Up Method*. Finally, we expected many similarities between *Rename Class*, *Rename Variable* and *Rename Method*.

What follows is a detailed discussion of the analysis we performed on the computed critical pairs. A first observation is that parallel applications of the same rule are always in potential conflict. In other words, the diagonal of the critical pair table always contains critical pairs. The reason for this is given below:

(i) Applying *Move Variable* twice to the same variable means that it should be moved to two different classes which is obviously a conflict. Also, two different variables with the same name cannot be moved to the same class due to the negative application condition. Applying *Move Method* twice generates similar conflicts as applying *Move Variable* twice.

(ii) *Pull Up Variable* is in conflict with itself because it cannot be used to pull up two different variables with the same name to the same class due to the negative application condition. Applying *Pull Up Method* twice generates similar conflicts as applying *Pull Up Variable* twice.

(iii) Applying *Encapsulate Variable* twice generates a conflict because one cannot introduce the same accessor methods twice.

(iv) *Create Superclass* is in conflict with itself, since the generalization relation between the class for which a new superclass must be created and its current superclass is deleted. Stated differently, the introduction of two new superclasses would give rise to a multiple inheritance hierarchy, which is prohibited by the multiplicities imposed in the type graph of Figure 1. Another conflict arises if two superclasses with the same name are inserted.

(v) Applying *Rename Class* twice generates a conflict, if the name of one and the same class is changed twice in a different way. Another conflict occurs, if two different classes are renamed with the same name. Applying *Rename Variable* or *Rename Method* twice generates similar conflicts as applying *Rename Class* twice.

A *symmetric conflict* arises in the following situations:

(i) *Move Variable* and *Pull Up Variable* are in conflict if the same variable is pulled up and moved. Furthermore, pulling up one variable and moving another with the same name into the same class causes a conflict due to the negative application conditions of both rules. *Move Method* versus *Pull Up Method* gives rise to a similar symmetric conflict.

(ii) *Move Variable* versus *Encapsulate Variable* causes a symmetric conflict. After moving a variable, it cannot be encapsulated (within the original

class) anymore. Conversely, encapsulating a variable it is no longer public and cannot moved anymore. *Pull Up Variable* versus *Encapsulate Variable* gives rise to a similar symmetric conflict.

(iii) *Move Method* versus *Encapsulate Variable* generates a symmetric conflict as explained in section 2.2. *Pull Up Method* versus *Encapsulate Variable* gives rise to a similar symmetric conflict, and so does *Rename Method* versus *Encapsulate Variable*.

(iv) *Create Superclass* is in conflict with *Rename Class*, if both rules create a new class with the same name.

(v) *Rename Variable* and *Move Variable* resp. *Pull Up Variable* are in symmetric conflict, since the variable to be moved or pulled up is renamed. Otherwise, the variable to be renamed is moved (pulled up) to another class. The symmetric conflicts between *Rename Method* and *Move Method* resp. *Pull Up Method* are similar.

We encountered *asymmetric conflicts* in the following situations:

(i) *Create Superclass* causes an asymmetric conflict on *Pull Up Variable*, since it modifies the generalization relation needed for pulling up the variable. It causes a similar asymmetric conflict on *Pull Up Method*.

(ii) *Rename Variable* causes an asymmetric conflict on *Encapsulate Variable*, since it renames the variable to be encapsulated.

It is important to stress here that the number of conflicts that are detected by the algorithm relies on the chosen metamodel as well as on the specification of the refactorings. Since we made some simplifications to both in our feasibility study, the number of detected critical pairs is likely to increase if we would apply it to a more realistic refactoring suite.

5 Conflict resolution

Critical pairs describe potential conflicts between different rule applications. Often it is possible to show that this critical situation is confluent. Intuitively, this means that the application of one conflicting rule may prohibit the application of the other one, but further transformations may be applied to resolve the conflicting situation. Formally, a critical pair $(G \rightarrow H_1, G \rightarrow H_2)$ is confluent if there are transformations $(H_1 \rightarrow X, H_2 \rightarrow X)$ that lead to the same result graph X.

In the following, we discuss to which extent the potential conflicts found by critical pair analysis are confluent and can thus be resolved. We performed the conflict resolution analysis manually. It is left to future work to automate this analysis in AGG.

We start with explaining all conflicts due to parallel applications of the same rule:

(i) Moving a variable first to some class and then to another class leads to a

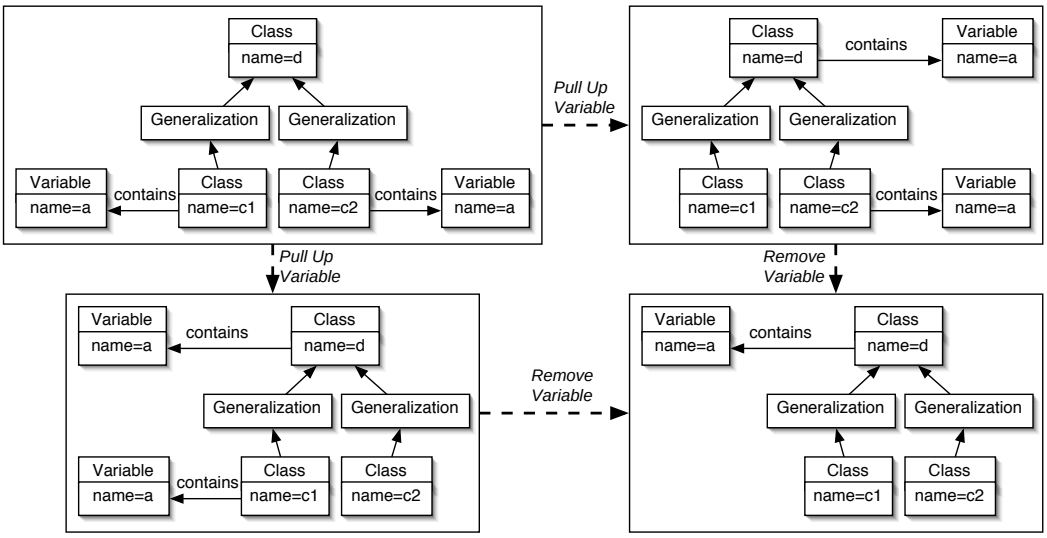


Fig. 9. Resolving parallel evolution conflicts by analysing confluence of critical pairs.

conflict that cannot be solved automatically. One of these moves has to be given the priority by the developer. Trying to move different variables with the same name to the same class also results in a critical pair. It can be solved by renaming one of the variables, i.e., applying rule *Rename Variable* to it, and moving the other variable afterwards. Applying *Move Method* twice generates similar conflicts as applying *Move Variable* twice. Thus, conflict solving is similar.

(ii) If two different variables with the same name (but residing in different subclasses) need to be pulled up into the same class, this conflict can be solved by deleting one of the two variables and pulling the other one up. This solution is visualised in Figure 9. Applying *Pull Up Method* twice generates similar conflicts as applying *Pull Up Variable* twice. Thus, conflict solving is similar.

(iii) Applying *Encapsulate Variable* twice for the same variable needs to be resolved by ignoring one of both rule applications.

(iv) Applying *Create Superclass* twice leads to conflicts that can be resolved by ignoring one of both rule applications.

(v) Renaming a class twice leads to a conflict that cannot be solved automatically. One of these renamings has to be given the priority by the developer. If two different classes with different names should be renamed using the same name, this also results in a critical pair. It can be solved by manually choosing only one of the two classes to be renamed. Applying *Rename Variable* or *Rename Method* twice generates similar conflicts as applying *Rename Class* twice. Thus, conflict solving is similar.

Now, let us see how the symmetric conflicts can be resolved:

(i) Pulling up and moving the same variable is confluent, if the variable is

moved to a class that has a superclass. In this case, the variable can still be pulled up after moving. The other way round, the variable can always be moved after pulling it up.
If the variable is moved to a class without superclass, the critical pair is not confluent, because the pull up refactoring cannot be performed (due to absence of the superclass).
A third situation, pulling up and moving two different variables with the same name into the same class causes a confluent conflict situation. It can be solved by renaming first one of the variables and performing the refactoring afterwards.
Move Method versus *Pull Up Method* generates similar conflicts as *Move Variable* and *Pull Up Variable*. Thus, conflict solving is similar.

(ii) *Move Variable* versus *Encapsulate Variable*: Moving first a variable, it can be encapsulated within its new class, thus this situation is confluent. Encapsulating the variable first we reach the same state of changes if afterwards not only the variable is moved, but also the newly created getter and setter methods. These refactorings are only possible, if such accessor methods do not already exist in the new class. Otherwise, additional renamings have to be performed to make the situation confluent.

(iii) *Pull Up Variable* versus *Encapsulate Variable*: If we pull up the variable first, it can be encapsulated within the superclass. If we encapsulate it first, not only the variable but also its accessor methods have to be pulled up (using *Pull Up Method*). Again, as in the previous case, additional renamings may have to be performed to make the situation confluent.

(iv) *Move Method* versus *Encapsulate Variable*: If encapsulating a variable results in the creation of a method with the same name as the method to be moved to the same class, this conflict can be solved by first renaming the method to be moved and then moving it and encapsulating the variable.
Pull Up Method versus *Encapsulate Variable* generate a similar conflict as *Move Method* versus *Encapsulate Variable*. Thus, conflict solving is similar.

(v) Applying *Create Superclass* and *Rename Class* leads to conflicts that cannot be solved automatically. One of these refactorings has to be given the priority.

(vi) *Rename Variable* versus *Move Variable*: Moving a variable first, it has to be renamed within its new class. Renaming it first, the renamed variable is moved.
Pull Up Variable causes a similar conflict on *Rename Variable*. The conflicts between *Rename Method* and *Move Method* resp. *Pull Up Method* are also similar. Thus, conflict solving is similar in all those cases.

Finally, we discuss resolution of the asymmetric conflicts:

(i) Applying *Create Superclass* first *Pull Up Variable* has to be applied twice to get the same effect as pulling first up and then creating a superclass for the subclass. A similar conflict is caused on *Pull Up Method*. Thus, conflict solving is similar.

(ii) *Rename Variable* versus *Encapsulate Variable*: Renaming a variable first, the encapsulation has to be done on the renamed variable. The same effect is obtained by encapsulating first and renaming then not only the variable, but also its accessor methods.

(iii) *Rename Method* versus *Encapsulate Variable*: Encapsulating a variable first a new method is created. If a method is renamed to the name of this new method, this causes a conflict that needs to be resolved by ignoring one of the refactorings, or by performing an additional renaming.

6 Related work

In [5], the formalism of critical pairs was explained and related to the formal property of confluence of typed attributed graph transformations. In [4], critical pair analysis is used to detect conflicting requirements in independently developed use case models. In [1], critical pair analysis has been used to increase the efficiency of parsing visual languages by delaying conflicting rules as far as possible.

The problem that has been addressed in this paper is a well-known problem in the context of version management, and is referred to as software merging [7]. Two other approaches that rely on graph transformation to tackle the problem of software merging were proposed by Westfechtel [13] and Mens [6]. Like our approach, they attempt to detect structural merge conflicts. The novel contribution of the current paper, however, is the use of critical pair analysis to address this problem.

Refactoring is also a very active research domain [9]. Formal approaches have mainly been used to prove that refactorings preserve the behaviour of the program. Graph transformations have also been used to express refactorings [8,12]. To our knowledge, no formal attempt has been made to detect conflicts between refactorings applied in parallel.

7 Discussion

In this paper, we explored the problem of detecting and resolving structural conflicts that arise due to parallel evolution. We expressed refactorings as typed attributed graph transformations with negative application conditions, we used critical pair analysis to detect evolution conflicts, and confluence analysis to resolve the conflicts. From a practical point of view, the feasibility study we performed already provided very useful results. It allowed us to gain insight in the similarities of, and interactions between, different refactorings.

We believe that our approach has a lot of potential, and requires further exploration. For example, our approach may be very beneficial for refactoring tool developers. [11,10] proposed to combine the detection of "code smells" with a refactoring engine that resolves these smells. For each detected smell, there are typically many different refactorings that can be applied to resolve them [2], and some of these refactorings may be in conflict. Hence, a critical pair analysis of the possible choices may help the programmer to decide which refactoring to apply.

Another interesting application would be to incorporate conflict resolution strategies (based on confluence analysis) into refactoring tools. Suppose that a user wants to apply two refactorings sequentially, but the second one is not applicable due to a critical pair conflict. Rather than simply refusing to apply the second refactoring, the tool could suggest to perform an automatic resolution of the conflict that enables to apply the second refactoring.

During our experiments with AGG we encountered a number of limitations, which required us to improve the critical pair analysis algorithm. In the new version of AGG that we developed, the preparation of the critical pairs is already quite user-friendly, but there is still a potential for improvement to better understand the critical situations.

Another problem we have to deal with is the presence of *false positives* and *false negatives*. In order to reduce the possibility of *false negatives*, one needs to provide a more complex metamodel and more realistic refactorings. *False positives* arose because our transformation rules did not take the transitive closure of the specialization hierarchy into account. A straightforward solution would be to add specific transformation rules that compute the transitive closure before actually applying the refactoring rules. An alternative solution would be to use path expressions, but this would be very difficult to implement in AGG due to inherent limitations in the underlying formal approach.

References

[1] Bottoni, P., G. Taentzer and A. Schürr, *Efficient parsing of visual languages based on critical pair analysis and contextual layered graph transformation*, in: *Proc. IEEE Symp. Visual Languages*, 2000.

[2] Fowler, M., "Refactoring: Improving the Design of Existing Programs," Addison-Wesley, 1999.

[3] Habel, A., R. Heckel and G. Taentzer, *Graph Grammars with Negative Application Conditions*, Special issue of Fundamenta Informaticae **26** (1996).

[4] Hausmann, J. H., R. Heckel and G. Taentzer, *Detection of conflicting functional requirements in a use case-driven approach*, in: *Proc. Int'l Conf. Software Engineering* (2002).

[5] Heckel, R., J. M. Küster and G. Taentzer, *Confluence of typed attributed graph transformation systems*, 2002.

[6] Mens, T., *Conditional graph rewriting as a domain-independent formalism for software evolution*, in: *Proc. Int'l Conf. Agtive 1999: Applications of Graph Transformations with Industrial Relevance*, Lecture Notes in Computer Science **1779** (2000), pp. 127–143.

[7] Mens, T., *A state-of-the-art survey on software merging*, Transactions on Software Engineering **28** (2002), pp. 449–462.

[8] Mens, T., S. Demeyer and D. Janssens, *Formalising behaviour preserving program transformations*, in: *Graph Transformation*, Lecture Notes in Computer Science **2505** (2002), pp. 286–301, proc. 1st Int'l Conf. Graph Transformation 2002, Barcelona, Spain.

[9] Mens, T. and T. Tourwé, *A survey of software refactoring*, Transactions on Software Engineering **30** (2004), pp. 126–139.

[10] Tourwé, T. and T. Mens, *Identifying refactoring opportunities using logic meta programming*, in: *Proc. 7th European Conf. Software Maintenance and Re-engineering (CSMR 2003)* (2003), pp. 91–100.

[11] van Emden, E. and L. Moonen, *Java quality assurance by detecting code smells*, in: *Proc. 9th Working Conf. Reverse Engineering* (2002), pp. 97–107.

[12] Van Gorp, P., H. Stenten, T. Mens and S. Demeyer, *Towards automating source-consistent UML refactorings*, in: P. Stevens, J. Whittle and G. Booch, editors, *UML 2003 - The Unified Modeling Language*, Lecture Notes in Computer Science **2863** (2003), pp. 144–158.

[13] Westfechtel, B., *Structure-oriented merging of revisions of software documents*, in: *Proc. Int'l Workshop on Software Configuration Management* (1991), pp. 68–79.