SoSyM manuscript No.
(will be inserted by the editor)

Analysing Refactoring Dependencies Using Graph Transformation

Tom Mens[1], **Gabriele Taentzer**[2], **Olga Runge**[2]

[1] Software Engineering Lab
Université de Mons-Hainaut
B-7000 Mons, Belgium
e-mail: tom.mens@umh.ac.be

[2] Technische Universität Berlin
D-10587 Berlin, Germany
e-mail: {gabi,olga}@cs.tu-berlin.de

Received: date / Revised version: date

Abstract Refactoring is a widely accepted technique to improve the structure of object-oriented software. Nevertheless, existing tool support remains restricted to automatically applying refactoring transformations. Deciding what to refactor and which refactoring to apply still remains a difficult manual process, due to the many dependencies and interrelationships between relevant refactorings. In this paper, we represent refactorings as graph transformations, and we propose the technique of critical pair analysis to detect the implicit dependencies between refactorings. The results of this analysis can help the developer to make an informed decision of which refactoring is most suitable in a given context and why. We report on several experiments we carried out in the AGG graph transformation tool to support our claims.

Key words refactoring, graph transformation, critical pair analysis, dependency analysis, AGG

1 Introduction

Refactoring is a commonly accepted technique to improve the structure of object-oriented software [1,2]. For all common object-oriented languages and programming environments integrated support for applying refactorings is readily available. Even at the level of design models, support for model refactoring is starting to emerge [3–7].

Nevertheless, there are still a number of problems if we want to apply refactorings as automatically as possible. To illustrate these problems, consider the following scenario.

Assume that we have a tool that allows us to detect opportunities for refactoring [8]. Such a tool will detect badly structured code based on *code smells* [1,9], metrics [10, 11] or other techniques. It will use this information to propose a set of refactorings that can be used to improve the software structure. The developer then has to choose interactively which refactorings he would like to apply, and use a refactoring tool to apply these refactorings.

A problem with the above scenario is that the set of refactorings that will be proposed to the developer may be quite large, so that it is difficult to determine which refactorings in this set will be most beneficial. The problem becomes even worse since there may be implicit dependencies between the proposed refactorings. Applying any one of the suggested refactorings may prohibit the application of other refactorings that have been selected by the developer.

Therefore, the goal of this paper is to explore automated techniques to determine what are the implicit dependencies between a list of refactorings. In this way, we can help the developer to decide in which order the refactorings need to be applied (due to sequential dependencies between them), and which refactorings are more appropriate. In this article, a refactoring is considered to be more appropriate if it gives rise to fewer potential conflicts.[1]

The above analysis will allow the developer to get precise answers to the following concrete questions when selecting a concrete refactoring in a list of proposed refactorings:

- What are the alternatives of a selected refactoring (i.e., other mutually exclusive refactorings that address the same design smell)?
- Which other refactorings need to be applied first in order to make the selected refactoring applicable?
- Which other refactorings are still applicable after applying the selected refactoring?

Being able to answer these questions will allow the developer to perform "what if" scenarios and will allow him to get a better insight into the effect of applying a refactoring.

In order to achieve the above goal, we first of all need a precise formal specification of refactorings. We rely on graph transformation theory for this purpose. Next we need to be able to analyse mutual exclusion and sequential dependencies between refactorings. To this aim, we make use of critical pair analysis [12–14] of graph transformations. For our experiments, we used AGG[2], a general purpose graph transformation tool that supports critical pair analysis on typed attributed graph transformations.

[1] Alternatively, appropriateness could be expressed as a function of software quality. Different refactorings improve different quality aspects, and can be considered to be more appropriate if they address the particular quality aspect the software developer intends to improve.

[2] The most recent version of AGG can be downloaded from `http://tfs.cs.tu-berlin.de/agg`.

2 Motivating Example

As a running example throughout this article, we use a simplified version of a Local Area Network simulation (LAN) that has been adopted at various universities to teach object-oriented design and refactoring techniques [15]. The class hierarchy for this LAN is shown in Figure 1. We used the standard notation of UML class diagrams, enhanced with information to express message sends (e.g. `calls this.send(p)`), variable accesses (e.g. `accesses Packet.sender`) and variable updates (e.g. `updates Packet.sender`). Observe the need for dynamic method lookup in `calls this.send(p)` for the subclasses `Workstation`, `PrintServer` and `FileServer`. The message `send` is found in their common superclass `Node` via the method lookup mechanism. Also observe the need for dynamic (i.e., late) binding in the method `send` of class `Node` which calls `this.accept(p)`. `send` is a so-called *template method*[3] that relies on the method `accept` whose implementation is not specified by `Node`, but deferred to its subclasses. During program execution, the implementation of one of the `accept` methods in the subclasses will be used depending on the dynamic context in which the message was sent.

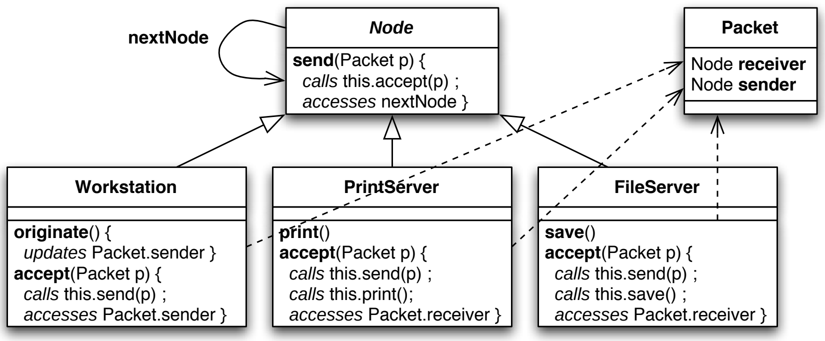

nextNode
Node
send(Packet p) { calls this.accept(p) ; accesses nextNode }
Packet
Node receiver
Node sender
Workstation
originate() { updates Packet.sender }
accept(Packet p) { calls this.send(p) ; accesses Packet.sender }
PrintServer
print()
accept(Packet p) { calls this.send(p) ; calls this.print(); accesses Packet.receiver }
FileServer
save()
accept(Packet p) { calls this.send(p) ; calls this.save() ; accesses Packet.receiver }

Fig. 1 Motivating example: simplified class diagram of a LAN simulation. `Node` is an abstract superclass that cannot be instantiated.

A software developer may want to improve the structure of the design in Figure 1 by applying a variety of different refactorings. Below we present and motivate some of these:

T_1 *Rename Method* `print` in class `PrintServer` to new name `process`. This refactoring should be performed in combination with the following one.

T_2 *Rename Method* `save` in class `FileServer` to new name `process`. This new name is deliberately the same as in T_1, since it prepares for the application of refactoring T_4 explained below.

[3] *Template Method* is a well-known object-oriented design pattern. For more details on this matter, we refer to [16].

T_3 *Create Superclass* `Server` for `PrintServer` and `FileServer`. The purpose of this refactoring is to show that the classes `PrintServer` and `FileServer` are similar in nature. They can both `accept` a packet sent by another node in the network and `process` it in a specific way.

T_4 *Pull Up Method* `accept` from classes `PrintServer` and `FileServer` to the superclass `Server` that was created by T_3. This refactoring is only possible thanks to the renamings performed by T_1 and T_2, which had as a deliberate side effect that the implementation of `accept` in both subclasses `PrintServer` and `FileServer` became identical, a prerequisite for being able to pull up the method.

T_5 *Move Method* `accept` from class `PrintServer` to class `Packet`. This refactoring is motivated by the fact that `accept` directly accesses the variable `receiver` in class `Packet`. Moving the method `accept` to `Packet` facilitates the implementation of active packets, which are packets that are responsible themselves for deciding to which destination they should be sent and what they should perform at this destination. Typical examples are broadcast packets that send information to a given set of nodes in the LAN, and collecting packets that collect information from a given set of nodes in the LAN.

T_6 *Move Method* `accept` from class `FileServer` to class `Packet`. The motivation for this refactoring is the same as for the previous one.

T_7 *Encapsulate Variable* `receiver` in class `Packet`. This refactoring is useful for increasing modularity, by avoiding direct accesses of the local state of a packet. Thanks to such encapsulation, it becomes possible to change the internal representation of the packet independent of its external clients.

T_8 *Add Parameter* `p` of type `Packet` to method `print` in class `PrintServer`. In order to `print` the information stored in a packet, it is necessary to pass this packet as a parameter.

T_9 *Add Parameter* `p` of type `Packet` to method `save` in class `FileServer`. The motivation for this refactoring is the same as for T_8.

Even though the LAN simulation is a very simple example, the list of refactorings proposed above is already quite large. In addition, there may be many implicit or explicit interactions between these refactorings:

- Some of the proposed solutions (e.g., T_4 and T_5) are mutually exclusive, because they are incompatible with one another. Obviously, one cannot pull up a method to a superclass, and at the same time move this method to another unrelated class. This scenario is depicted in Figure 2.
- Some of the refactorings are sequentially dependent, in the sense that they rely on other refactorings that have to be applied before. This is for example the case for T_4, which relies on all previous refactorings T_1, T_2 and T_3.
- It is also possible to have pairs of refactorings where each refactoring in the pair can be applied in isolation, but when combined together they can only applied in a certain order. This is for example the case with T_1 and T_8 (and similarly, with T_2 and T_9). They can both be applied separately to Figure 1, but we can only apply them together in a specific order. Indeed, if we first add a parameter to the method `print` and afterwards decide to rename the

method `print`, there is no problem. If we try it in the opposite order, we will not be able to add a parameter to the method `print` as this method has been renamed.

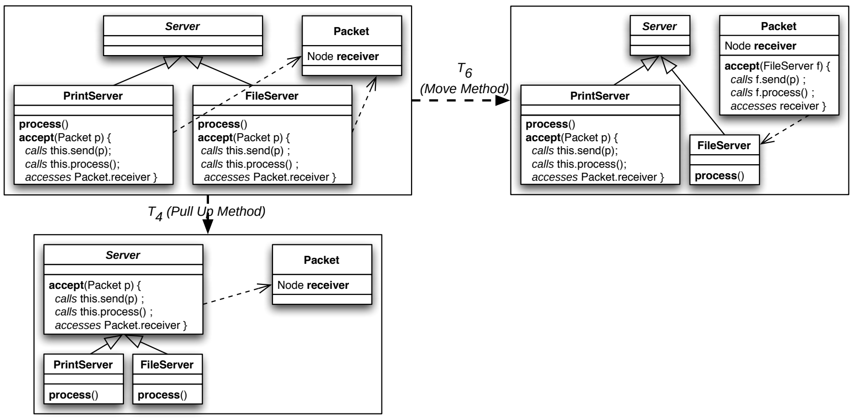


Fig. 2 Example of a mutual exclusion relationship between refactorings *Pull Up Method* and *Move Method*. `Server` is an abstract superclass of `PrintServer` and `FileServer` that cannot be instantiated. Its sole purpose is to capture the commonalities between its subclasses. After applying the *Pull Up Method* refactoring, the method `accept` becomes a template method in `Server` since it relies on another method `process` whose implementation is dynamically deferred to the subclasses.

It should be clear from this motivating example that automated support is needed to detect, for a given list of refactorings, which of these refactorings are mutually exclusive (and why), and which refactorings are sequentially dependent from each other. For example, it would be nice if we could automatically compute the Table 1 summarising all dependencies concerning the situation illustrated above (and more).

This table provides a lot of useful information to the software developer. For example, one can see at a glance that T_4 has a lot of sequential dependencies; T_4, T_5 and T_6 are in conflict to one another; and T_7 is the only refactoring in the list that does not interfere with any of the other refactorings. The information in the table also allows us to suggest an optimal way to apply the refactorings. For example, it is possible to apply the refactorings in the following order without giving rise to conflicts: $T_8, T_1, T_9, T_2, T_3, T_4, T_7$. Notice that T_5 and T_6 do not appear in this sequence as they are in mutual exclusion with T_4.

3 Suggested Solution

In order to automatically compute the information displayed in Table 1, we will specify a representative selection of refactorings by graph transformation rules.

	T_1	T_2	T_3	T_4	T_5	T_6	T_7	T_8	T_9
T_1	$\times$			$\leftarrow$				$\gg$	
T_2		$\times$		$\leftarrow$					$\gg$
T_3			$\times$	$\leftarrow$					
T_4	$\rightarrow$	$\rightarrow$	$\rightarrow$	$\times$	$\times$	$\times$			
T_5				$\times$	$\times$	$\times$			
T_6				$\times$	$\times$	$\times$			
T_7							$\times$		
T_8	$\ll$							$\times$	
T_9		$\ll$							$\times$

Table 1 Refactoring dependency table. $\times$ denotes mutual exclusion between two refactorings, $\leftarrow$ denotes a sequential dependency, and $\gg$ denotes an asymmetric conflict.

Concrete contexts (e.g., a program or a design model) will be represented by abstract syntax graphs. Basing refactoring specification on graph transformation in this way, we can use the techniques of *critical pair analysis* [12–14] and *sequential dependency analysis*. The first technique can be used to identify mutual exclusions and asymmetric conflicts as identified in Table 1, whereas the second technique can be used to detect sequential dependencies between refactorings.

An advantage of these analysis techniques is that they can be performed at an abstract level first, relying only on the abstract refactoring specifications, without taking into account the concrete context in which they will be applied. Once a concrete program is provided that needs to be refactored, the abstract analysis can be applied straightforwardly in this concrete refactoring context. The benefit of this approach is that the abstract analysis, which is the most time-consuming operation, needs to be performed only once, when the refactoring specifications are provided.

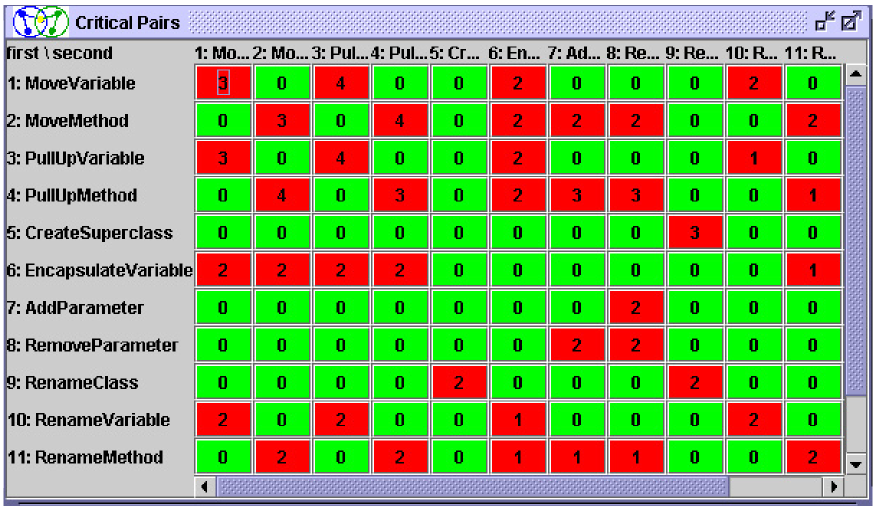
3.1 Conflict analysis of refactoring specifications

Critical pair analysis was first introduced for term rewriting, and later generalised to graph rewriting [17, 18]. The idea of critical pair analysis is quite simple. We explain it here in the context of refactoring. Given a predefined set of generic refactoring specifications (such as *Pull Up Method*, *Encapsulate Variable*, *Move Method*, *Create Superclass*, and so on), all pairs of such specifications are analysed for potential conflicts. A *critical pair* is detected when it is possible to find a minimal critical context to which both refactorings in the pair can be applied in a conflicting (i.e., mutually exclusive) way.

A critical pair analysis algorithm has been implemented in the AGG tool.[4] Applying it to a selection of 11 representative refactorings, we get the results displayed in Figure 3. For each conflicting pair of refactorings, a strictly positive number is shown in this figure, corresponding to the actual number of critical situations that can be computed between a given pair of refactorings. This number can

[4] AGG is the only available graph transformation tool that supports critical pair analysis.

be higher than one if the two considered refactorings conflict in different ways. By clicking on a number in the conflict table, all corresponding detailed conflict situations will be displayed to the user. We will explain this critical pair analysis in more detail in Section 6.



first \ second	1: Mo...	2: Mo...	3: Pul...	4: Pul...	5: Cr...	6: En...	7: Ad...	8: Re...	9: Re...	10: R...	11: R...
1: MoveVariable	3	0	4	0	0	2	0	0	0	2	0
2: MoveMethod	0	3	0	4	0	2	2	2	0	0	2
3: PullUpVariable	3	0	4	0	0	2	0	0	0	1	0
4: PullUpMethod	0	4	0	3	0	2	3	3	0	0	1
5: CreateSuperclass	0	0	0	0	0	0	0	0	3	0	0
6: EncapsulateVariable	2	2	2	2	0	0	0	0	0	0	1
7: AddParameter	0	0	0	0	0	0	0	2	0	0	0
8: RemoveParameter	0	0	0	0	0	0	2	2	0	0	0
9: RenameClass	0	0	0	0	2	0	0	0	2	0	0
10: RenameVariable	2	0	2	0	0	1	0	0	0	2	0
11: RenameMethod	0	2	0	2	0	1	1	1	0	0	2

Fig. 3 Critical pair analysis of the refactoring specifications.

While the critical pair analysis shown in Figure 3 is very useful to show conflicts in principal, it is too conservative in practice, in the sense that it computes all *potential* conflicts. In the context of a concrete scenario (such as the one in Figure 1, for example), only a small fraction of these conflicts will actually occur. Therefore, the obvious and straightforward solution is to consider the critical pair analysis as some kind of preprocessing stage which needs to be carried out only once. Whenever we provide a concrete context, we only need to filter the set of computed critical pairs to find out which of the *potential conflicts* are *actual conflicts* that match in this concrete context. After this filtering phase, we expect to get as results all the mutual exclusions and asymmetric conflicts that are shown in Table 1.

3.2 Conflict analysis of refactoring applications

In a second phase, we can consider concrete conflicts and sequential dependencies as they occur when analysing the refactoring possibilities for a concrete context (i.e., the syntax graph of a concrete program or design model).

Applicability analysis: First of all, we can determine which refactorings are applicable in a concrete context. For example, in the context of Figure 1, refactoring T_4 is not immediately applicable (because the method bodies of `accept` are

not identical, and because the superclass `Server` is missing). But after applying refactorings T_1, T_2 and T_3, T_4 does become applicable. On the other hand, after T_4, refactorings T_5 and T_6 are not applicable anymore.
We see that there are different reasons for refactorings not being applicable. Either they sequentially depend on other ones that have to be applied first, or they mutually exclude each other. Moreover, certain refactorings are never applicable in a given context. For example, refactoring *Pull Up Variable* is not applicable in the context of Figure 1, as none of the subclasses in the example contain variables that can be pulled up. Because we specified refactoring applications by graph transformations, AGG can perform this applicability analysis automatically, and it reports for each context graph the set of refactoring rules that are applicable. After this analysis, the user can even browse through all possible matches of each applicable rule in the concrete context graph.

Parallel conflict analysis: As we discussed above, critical pairs of refactoring specifications describe potential conflicts. Given a concrete context, we can further analyse which of these potential conflicts actually occur in this context. To this extent, we first check which refactorings are applicable and then look up the potential conflicts of applicable refactorings to find out which of these real conflicts occur in the given context. For example, the results of the critical pair analysis in Figure 3 show how refactorings *Move Method* and *Pull Up Method* give rise to different conflicts. One of these potential conflicts is an actual conflict in the context of Figure 1, since we found in Table 1 that refactoring T_4 (*Pull Up Method* `accept`) is mutually exclusive with refactorings T_5 and T_6 (*Move Method* `accept`). Parallel dependency analysis is also supported by AGG. See Section 7 for more details.

Sequential dependency analysis: Based on the applicability analysis of refactorings we can also analyse which refactorings are sequentially dependent of each other. For example, considering our motivating example we find that refactoring T_4 is not applicable in the beginning but it becomes applicable after applying refactorings T_1, T_2 and T_3. After having applied a refactoring in a concrete context, the applicability check can be triggered manually to find out whether new refactorings become applicable, or existing refactorings stop being applicable.
In the version of AGG that we used for our experiments, sequential dependency analysis was not yet supported. As of version 1.3 of AGG, however, sequential dependency analysis is supported, which means that also this part of the refactoring analysis can be automated.

4 Graph representation of object-oriented models

To be able to use the technique of critical pair analysis, we need to specify object-oriented refactorings as graph transformations. Before we can do this, however, we need to agree on how object-oriented models (or programs) can be specified as graphs. More specifically, we will use directed, attributed graphs. Additionally, these graphs must be typed to be able to determine whether or not a graph is well-formed. To this extent, we also need to specify a type graph that corresponds to the meta model to which all concrete graphs need to conform.

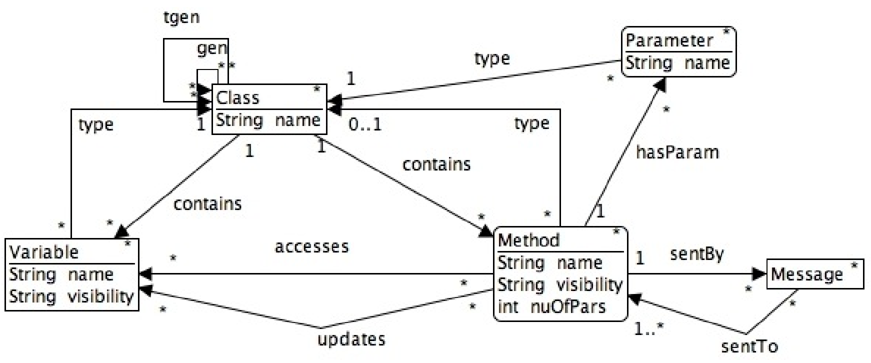


Fig. 4 Type graph representing the object-oriented meta model.

The notion of a *type graph* has been formally introduced in [19]. The (attributed) type graph for our object-oriented meta model is shown in Figure 4. 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 generalization, containment, typing, variable accesses, variable updates and message sends) with associated multiplicities. Note that *gen*-edges represent the usual generalisation relationship, whereas *tgen*-edges represent their transitive counterpart. Dynamic binding of message sends can be modelled with this type graph by a node of type `Message` with one sender (a node of type `Method` linked to the `Message` via an edge of type `sentBy`) and multiple potential receivers (all nodes of type `Method` linked to the `Message` via an edge of type `sentTo`).

To keep the paper (and especially the pictures) readable, we have deliberately restricted ourselves to a simplified meta model. For example, we did not model interfaces, abstract classes, abstract methods and the like. Even associations (such as `nextNode` in Figure 1) are not represented explicitly, but modelled as variables instead.

Figure 5 represents the LAN example of Figure 1 as a graph conforming to the type graph of Figure 4. Observe how the mechanism of late binding is represented statically: method `send` contained in class `Node` sends a message `accept` with three potential receivers, the methods `accept` defined in subclasses `Workstation`, `PrintServer` and `FileServer`, respectively.

Note that not all possible well-formedness constraints can be expressed in the type graph. In AGG, this problem can be resolved by adding additional global graph constraints. For example, we expressed the following constraints in this way:

– no two classes should have the same name
– no two methods contained in the same class should have the same name

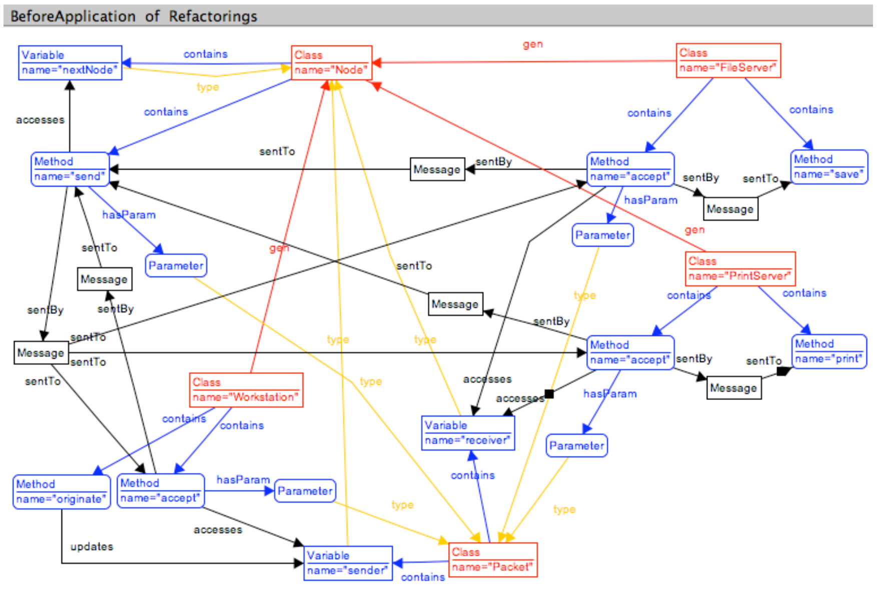


Fig. 5 Concrete graph representing the LAN example. In this figure, we omitted all *tgen*-edges because they coincide with *gen*-edges. In general, *tgen*-edges can be derived from the *gen*-edges by repeatedly applying two straightforward graph transformations specifically implemented for this purpose.

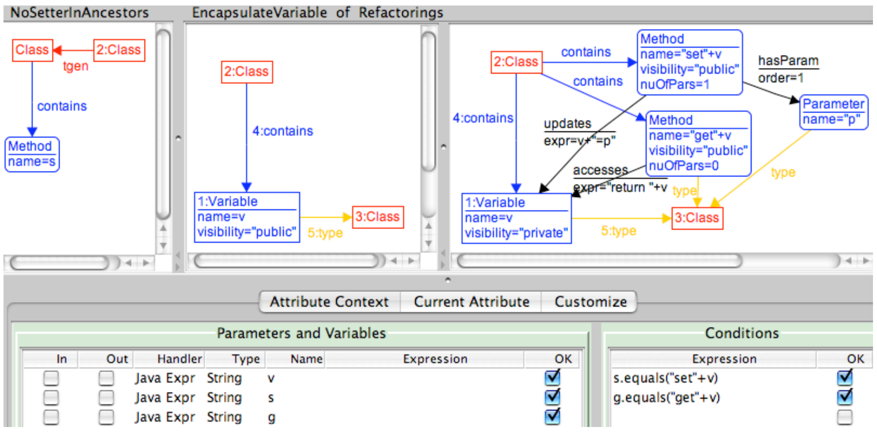
- no two variables contained in the same class should have the same name
- If there are multiple methods with the same name in the same class hierarchy, any message sent to one of these methods should also be sent to all other methods with the same name in the hierarchy (since it is impossible to determine the actual receiver method statically due to the mechanism of dynamic method binding)

  The concrete graph constraints can be looked up on the AGG home page.

5 Specification of object-oriented refactorings

Since programs (or design models) are specified as type graphs, refactorings can be expressed as *typed graph transformations*. A *graph transformation* $t : G \Longrightarrow_{p(m)} H$ is defined as a pair consisting of a *graph production rule* $p : L \rightarrow R$ and a *match* $m : L \rightarrow G$. 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 [14], one can easily extend this definition to come to a notion of *typed graph transformations* that respects the type constraints imposed by the type graph (without multiplicities).

As a concrete example, the transformation *Encapsulate Variable* in Figure 6 can be applied to a class containing a variable of a particular type. It changes the



Parameters and Variables

In	Out	Handler	Type	Name	Expression	OK
☐	☐	Java Expr	String	v		✓
☐	☐	Java Expr	String	s		✓
☐	☐	Java Expr	String	g		✓

Conditions

Expression	OK
s.equals("set"+v)	✓
g.equals("get"+v)	✓
	☐

Fig. 6 Graph transformation for the *Encapsulate Variable* refactoring. The upper middle pane represents the LHS, the upper right pane represents the RHS, and the upper left pane represents one of the NACs. The bottom panes are used to specify constraints between variables used in the NAC and LHS.

visibility of a variable in a class from public to private. It also introduces a new setter method and getter method for this variable in the class. The return type of the getter method, as well as the parameter type of the setter method, must be the same as the type of the encapsulated variable. The rest of the class 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.

Because the graphs that we use are attributed, the values of node and edge attributes in the graph may be modified by the transformation. This is for example the case in Figure 6 with the attribute `visibility` of variable node `1`, whose value is modified from `public` to `private`.

Another crucial feature of AGG is the ability to specify *negative application conditions* (NACs) [20] that capture the negated preconditions of a transformation. In the refactoring community, preconditions are frequently used to specify the applicability constraints of a refactoring [21–24].

In a graph transformation setting, NACs can be considered as a kind of forbidden subgraphs. For example, the transformation rule *Encapsulate Variable* contains the following NACs (only one of them is shown in Figure 6):

– NAC *No Setter* 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 relating the name `s` of the method in the NAC to the corresponding setter method name `v` in the RHS using the condition `s.equals("set"+v)`.

– NAC *No Getter* forbids the existence of a getter method in the class where the variable is to be encapsulated. An attribute condition relates the name `s` of the method in the NAC to the corresponding getter method name `g` in the RHS using the condition `g.equals("get"+v)`.
– NACs *No Getter In Ancestors* and *No Setter in Ancestors* are the same as *No Getter* and *No Setter*, but for all the ancestor classes of the class containing the variable to be encapsulated.
– NACs *No Getter In Descendants* and *No Setter in Descendants* are the same as *No Getter* and *No Setter*, but for all the descendant classes of the class containing the variable to be encapsulated.

Besides the *Encapsulate Variable* refactoring explained above, we implemented many other refactorings from Martin Fowler's refactoring catalog [1] as typed attributed graph transformations with NACs. The complete list is given below. The most interesting refactorings are presented in the following figures, leaving out most of the NACs.

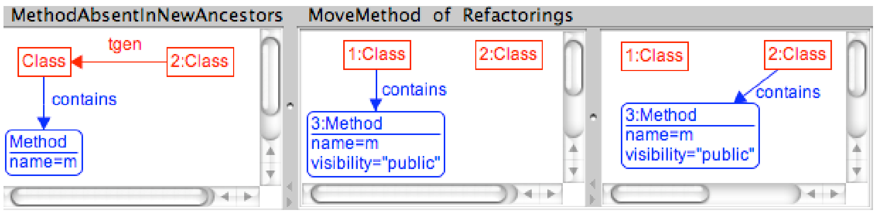
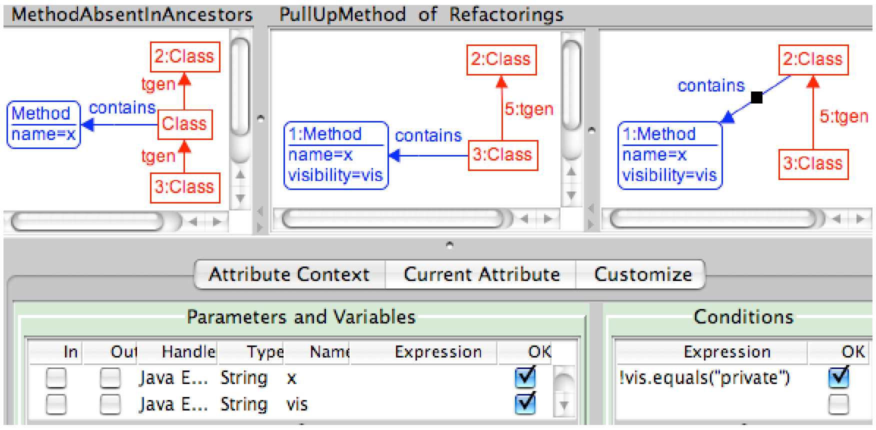


Fig. 7 Graph transformation rule for *Move Method*. Only one of its NACs is shown in the left pane.

– *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 7. Note that this rule is an oversimplification as it does not capture the difference between dynamic message sends and static message sends. To be complete, moving a method to another class that does not have a common ancestor with the source class should also entail the replacement of all dynamic messages to and from this method by static messages. In the current implementation of this refactoring, we opted for a more conservative definition where moving a method is prohibited if there are still dynamic message sends to or from this method.
– *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 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 shown in Figure 8. An attribute condition is used to prevent private methods from being pulled up.

- *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 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 9.
- *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 10.
- *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*.
- *Add Parameter* adds a new parameter to a given method.
- *Remove Parameter* removes an unused parameter from a given method.



In	Out	Handle	Type	Name	Expression	OK
☐	☐	Java E...	String	x		☑
☐	☐	Java E...	String	vis		☑

Expression	OK
!vis.equals("private")	☑
	☐

Fig. 8 Graph transformation rule for *Pull Up Method*. The attribute condition `!vis.equals("private")` specifies that only public or protected methods can be pulled up.

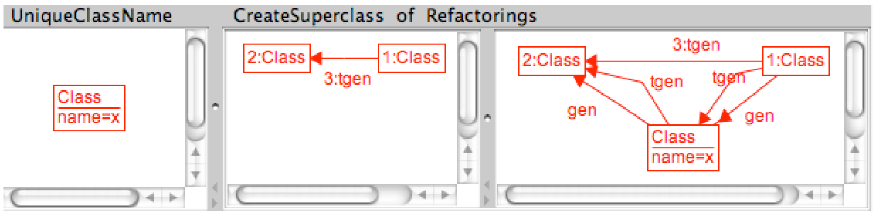


Fig. 9 Graph transformation rule for *Create Superclass*. Note that this transformation retains all existing transitive generalization edges, and introduces some extra ones.

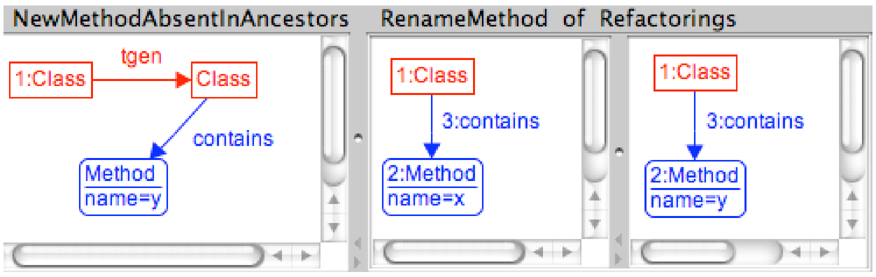


Fig. 10 Graph transformation rule for *Rename Method*.

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 expected conflicts between parallel refactorings can be detected by critical pair analysis. For this purpose, we chose a rather abstract graph representation that abstracts from all implementation details. Moreover, 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. For example, in the case of *Pull Up Method*, many more preconditions are required than the ones we actually implemented: the method should not directly access attributes from its defining class; the method should not call other methods in its defining class that are not understood by its superclass; the method should not perform super calls. For a detailed treatment of all these preconditions for this and other refactorings, we refer to [25].

Although, in theory, some of the simplifications we made may lead to false negatives during conflict detection, in practice, it turned out that all of the conflicts we expected to occur were actually detected. Furthermore, also unexpected conflicts were reported, such as the conflicts between renaming and move refactorings. Roughly considering these conflicts one could argue that even renamed variables/methods could be moved and should not cause conflicts. But we will see that the analysis will report a conflict, since the move refactoring binds the variable/method name which is changed after renaming. Thus, the analysis can sharpen the view on interdependencies between different refactorings.

6 Conflict analysis of refactoring rules

Critical pair analysis is known from term rewriting and can be used to check if a rewriting system can contain conflicting computations. Critical pair analysis has been generalized to graph rewriting in [17] and is formally presented for typed attributed graph transformation in [18]. Critical pairs formalize the idea of showing a conflicting situation in a minimal context. From the set of all critical pairs we can extract the objects and links which cause conflicts or dependencies. Let us now take a closer look at the idea of critical pair analysis. We start by providing some definitions.

Definition 1 (conflicting graph transformations) *Two graph transformations $t_1 : G \Longrightarrow_{p_1(m_1)} H_1$ and $t_2 : G \Longrightarrow_{p_2(m_2)} H_2$ are **in conflict** if t_1 cannot be performed after t_2 (i.e., rule p_1 cannot be applied to H_2) or vice versa (i.e., rule p_2 cannot be applied to H_1).*

Definition 2 (critical pair) *A critical pair is a pair of conflicting graph transformations $t_1 : G \Longrightarrow_{p_1(m_1)} H_1$ and $t_2 : G \Longrightarrow_{p_2(m_2)} H_2$ such that G is a minimal graph. G is minimal if there is not a proper subgraph G' of G such that there are conflicting transformations $t'_1 : G' \Longrightarrow_{p_1(m'_1)} H'_1$ and $t'_2 : G' \Longrightarrow_{p_2(m'_2)} H'_2$ with $m'_i(x) = m_i(x)$ for all $x \in L_{p_i}$ and $i = 1, 2$.*

To construct minimal critical graphs we basically consider all overlapping graphs of the left-hand sides of two rules with the obvious matches. If one of the rules contains NACs, extensions of the left-hand sides by parts of the corresponding NACs have to be considered for the construction of overlapping graphs in addition. The reasons why graph rules can be in conflict are threefold:

1. One rule application deletes a graph object (i.e., a node or edge) which is in the match of another rule application.
2. One rule application generates graph objects that give rise to a graph structure that is prohibited by a NAC of another rule application.
3. One rule application changes attributes being in the match of another rule application.

As an example of two graph transformation rules that are in conflict, consider *Pull Up Method* of Figure 8 and *Move Method* of Figure 7. These are in conflict, because they give rise to a number of critical pairs. One of these conflicts is visualised in Figure 11. Intuitively, the conflict arises because we move a method to a new class, while in parallel we pull up the same method to another class.

AGG supports 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 (see, for example, Figure 3).

In the case of *Pull Up Method* versus *Move Method* explained above, four critical pairs are reported. Two of the critical graphs computed by AGG for this situation are shown in Figure 15. They show critical overlapping graphs of the left-hand sides of the rules in Fig. 8 and 7. Both critical graphs report similar conflict situations that correspond to the conflict illustrated in Figure 11. The additional two conflicts not depicted are less interesting, since they report possible conflicts that cannot occur in our setting. This is due to the fact that AGG's critical pair algorithm abstracts away from concrete attribute interrelations. Since arbitrary Java expressions can be used for attribute conditions and computations, it just reports general conflicts on attribute usage, i.e., one rule application changes an attribute that another rule application uses. Acting in this way, it happens that some of the possible conflicts reported can never become real conflicts. Most of this kind of

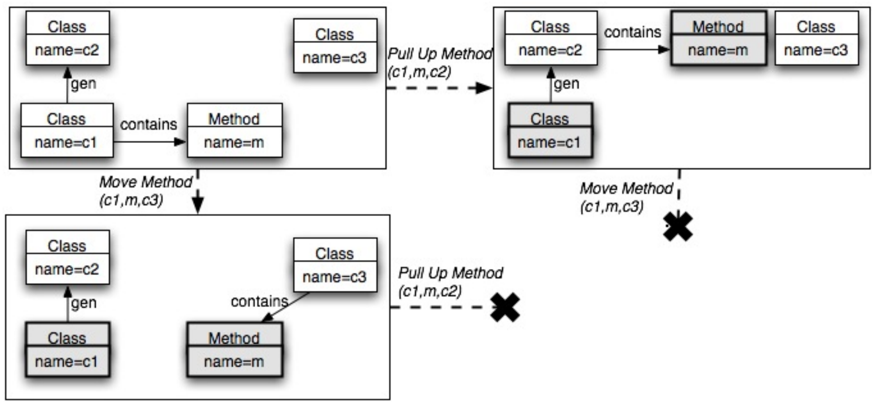


Fig. 11 Example of a conflict between graph transformations *Move Method* and *PullUp-Method*.

potential conflicts can be filtered out by specifying additional multiplicity constraints in the type graph and by further graph constraints postulating existence or non-existence of certain graph structures. In this way, the underlying meta model can be better adapted. More details to this topic can be found in the user manual on the AGG web page.

We applied the critical pair analysis algorithm of AGG to the selection of refactorings presented in Section 5. We observed that, for many pairs of refactorings, duplicate critical pairs were reported for the same conflict. Therefore, we improved the algorithm to disregard meaningless critical pairs by taking into account the upper bounds of the multiplicity constraints in the type graph of Figure 4. The results of this improved algorithm are shown in Figure 3. All critical pairs can be considered in detail on the AGG web page.

It is important to note here that the critical pairs that are detected by the algorithm rely on the chosen meta model (type graph) 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.

Nevertheless, the obtained results correspond mainly 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*.

For a detailed discussion of the analysis we performed on the computed critical pairs, we refer to [26]. We will report our most important observations here. A first observation is that parallel applications of the same rule at the same match are always in conflict. But this conflict is always solvable by performing only one of these equal rule applications. In other words, the diagonal of the critical pair table would always contain critical pairs which are obvious. Because of this, we decided to filter them out in Figure 3 in order not to unnecessarily clutter the results.

A second important observation is the presence of *asymmetric conflicts* in some cases. Especially conflicts of this kind were unexpected in the beginning of our analysis, since they are less obvious. An asymmetric situation indicates that it is possible to apply two transformations in a particular order, but not the other way around. Such information is very important to us, as it can be considered as a special kind of sequential dependency that allows us to reduce the set of refactorings that should be suggested to the software developer in a given context. More specifically, if we know that refactoring T_1 can be applied after refactoring T_2 but not the other way around, then we will only propose T_2 in the list of suggested refactorings. The asymmetric conflicts that can be found in the conflict table of Figure 3 are that *Add Parameter* and *Remove Parameter* can be applied before *Move Method*, *Pull Up Method* or *Rename Method* but not after.

As a third important observation of the critical pair analysis, we can conclude that there is a preferred order in which to apply refactorings in order to reduce the number of actual conflicts in a refactoring sequence. For example, based on the asymmetry in the critical pair table of Figure 3 we can avoid many conflicts by applying the following heuristics. Apply refactorings *Add Parameter* and *Remove Parameter* as early as possible, because in the table there are more conflicts reported in columns 7 and 8 than in rows 7 and 8. Apply renamings (*Rename Class*, *Rename Method* and *Rename Variable*) as late as possible, because in the table there are more conflicts reported in rows 9, 10 and 11 than in columns 9, 10 and 11.

7 Conflict analysis of refactoring applications

In AGG, it is possible to check which of the refactorings are applicable to a concrete input graph G: A refactoring is applicable if there exists at least one match of its left-hand side (taking into account the NACs) in G. The list of all refactorings that are applicable to the graph in Figure 5 is shown in Figure 12. It is obtained by using AGGs menu item “Check Rule Applicability”. *Pull Up Variable* and *Remove Parameter* are reported as non-applicable because, in the considered input graph, none of the subclasses have variables, and because all methods having parameters are called by others, thus prohibiting their removal.

Considering a specific graph like the one in Figure 5, not all reported critical pairs are relevant in this context, since not all refactorings are applicable. Therefore, AGG supports the analysis of conflicts in concrete instance graphs by selecting only the relevant critical pairs and showing how the corresponding conflict graphs are matched to the instance graph. To analyse our sample refactorings closer we take the concrete instance graph of Figure 5, apply refactorings T_1, T_2 and T_3 of Section 2, and get the resulting instance graph shown in Figure 13. Figure 14 shows all critical pairs that are relevant in the context of this instance graph. Looking closer we see that concrete conflicts are reported for applying *Move Method* twice as well as for applying *Pull Up Method* in combination

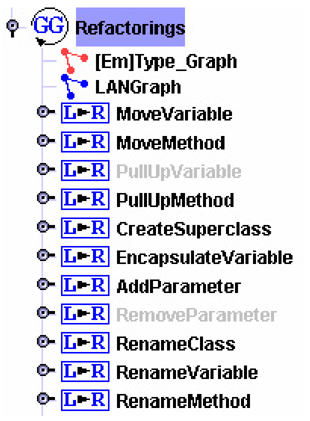

Refactorings
[Em]Type_Graph
LANGraph
L▸R MoveVariable
L▸R MoveMethod
L▸R PullUpVariable
L▸R PullUpMethod
L▸R CreateSuperclass
L▸R EncapsulateVariable
L▸R AddParameter
L▸R RemoveParameter
L▸R RenameClass
L▸R RenameVariable
L▸R RenameMethod

Fig. 12 Refactorings that are applicable to the LAN graph of Figure 5 are shown in black, the others are shown in gray.

with *Move Method*. In other words, the conflicts we expected in Section 2 are also derived by our formal dependency analysis.

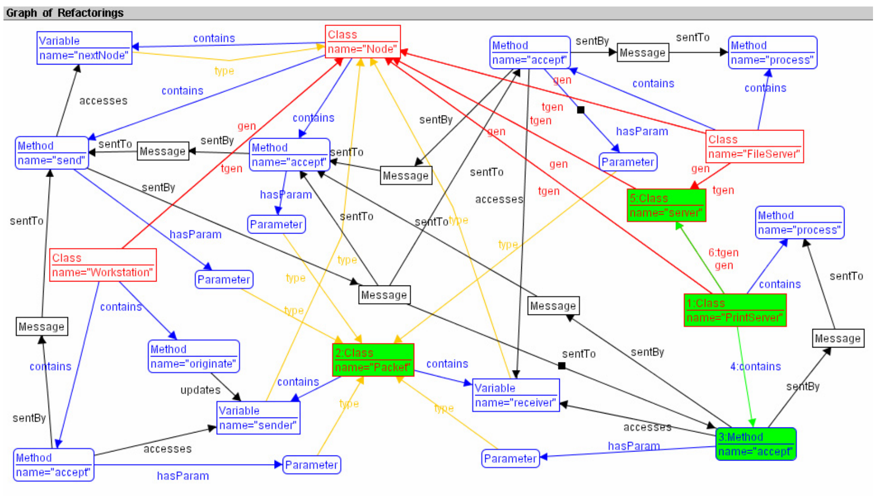
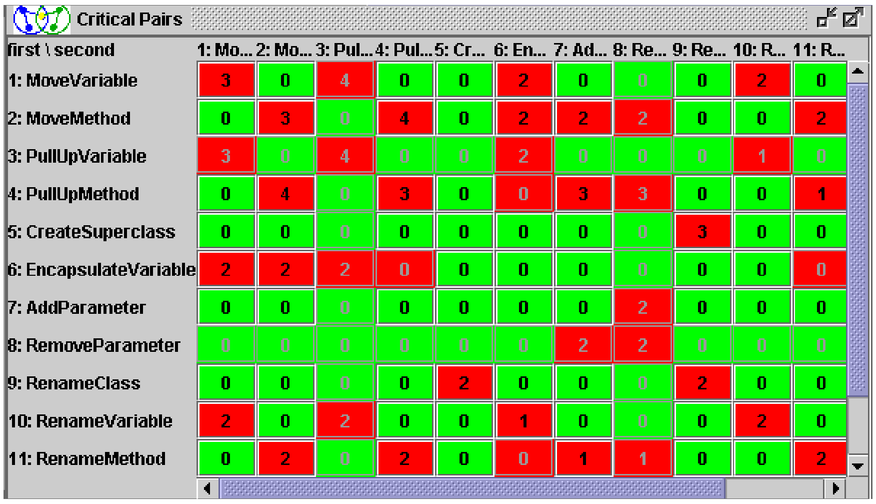


Fig. 13 The instance graph in Figure 5 after applying refactorings T_1, T_2 and T_3. Observe the use of *tgen*-edges to denote the transitive generalization relationship: there is a *tgen*-edge from *FileServer* and *PrintServer* to their indirect superclass *Node*.

Now we consider the actual conflicts between refactorings *Move Method* and *Pull Up Method* closer. There are 4 possible conflict situations reported for the instance graph in Figure 13. Two of them lead to relevant conflicts. The corresponding minimal conflict graphs are shown in Figure 15. Taking the second conflict



Critical Pairs

first \ second	1: Mo...	2: Mo...	3: Pul...	4: Pul...	5: Cr...	6: En...	7: Ad...	8: Re...	9: Re...	10: R...	11: R...
1: MoveVariable	3	0	4	0	0	2	0	0	0	2	0
2: MoveMethod	0	3	0	4	0	2	2	2	0	0	2
3: PullUpVariable	3	0	4	0	0	2	0	0	0	1	0
4: PullUpMethod	0	4	0	3	0	0	3	3	0	0	1
5: CreateSuperclass	0	0	0	0	0	0	0	0	3	0	0
6: EncapsulateVariable	2	2	2	0	0	0	0	0	0	0	0
7: AddParameter	0	0	0	0	0	0	0	2	0	0	0
8: RemoveParameter	0	0	0	0	0	0	2	2	0	0	0
9: RenameClass	0	0	0	0	2	0	0	0	2	0	0
10: RenameVariable	2	0	2	0	0	1	0	0	0	2	0
11: RenameMethod	0	2	0	2	0	0	1	1	0	0	2

Fig. 14 Critical pairs relevant for the instance graph in Figure 13. Rows and columns 3 and 8 are displayed in gray because *Pull Up Variable* and *Remove Parameter* are not applicable (see Figure 12). All other changes with respect to the critical pair table of Figure 3 are also displayed in gray. More specifically, in the instance graph, no critical pair is reported for the combinations (4,6), (6,4), (6,11) and (11,6).

graph (indicated by (2) in the Figure), we can embed it into the instance graph of Figure 13 in different ways. One of these embeddings, corresponding to the conflict between T_4 and T_6 is highlighted in Figure 13 in green (gray).

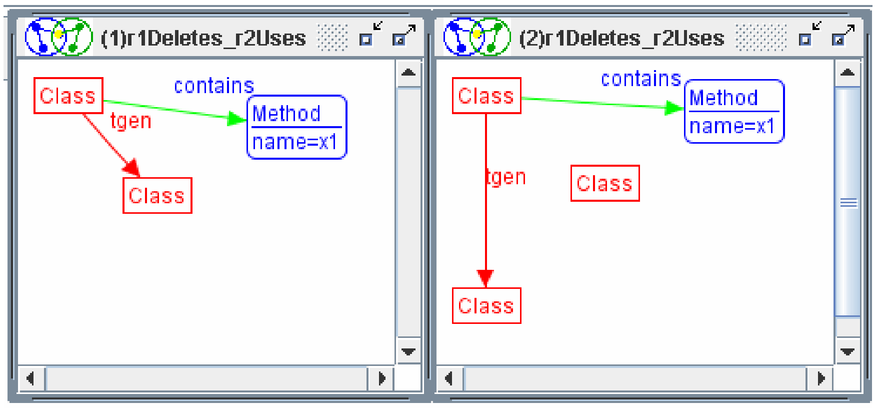


Fig. 15 Conflicts of *Move Method* and *Pull Up Method* which can occur in the graph in Figure 13

Besides analysing pairs of refactorings (at abstract or concrete level), we can of course also apply the refactoring transformations directly in AGG. Figure 16 shows the result of applying the sequence of refactorings $T_8, T_1, T_9, T_2, T_3, T_4, T_7$ of Section 2. Note that T_5 and T_6 have not been applied in this sequence because they have a critical pair conflict with T_4. In this case, the most obvious resolution strategy would be to replace T_5 and T_6 by a single new transformation "Move method `accept` from class `Server` to class `Packet`".

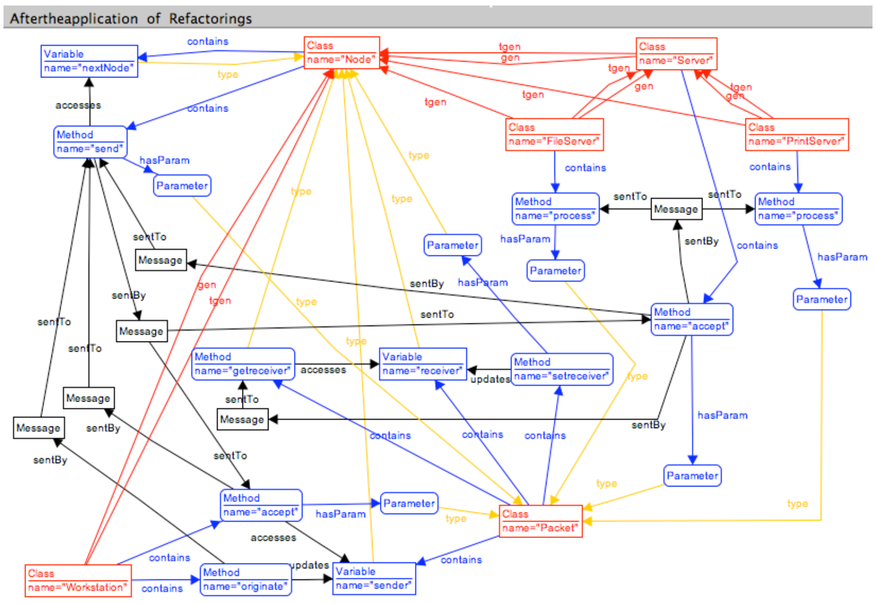


Fig. 16 Result of applying the sequence of refactorings $T_8, T_1, T_9, T_2, T_3, T_4, T_7$ to the LAN graph of Figure 5.

Please note that the sequential dependencies we analysed in our sample refactoring scenario in Table 1 are not observable in our formal specification of refactorings based on graph transformations. This is mainly due to the fact that we modelled only a restricted variant of the *Pull Up Method* and *Pull Up Variable* refactorings. More specifically, in the graph transformation rules we only specified the case where a method/variable is pulled up from a single subclass. One would need a similar rule for pulling up from two subclasses, from three subclasses, and so on. In the general case, we need an additional mechanism that allows us to specify an infinite set of transformation rules. Although this seems to be feasible from a theoretical point of view using amalgamated graph transformation [27], this concept is not yet supported by AGG.

8 Related work

In [14], the formalism of critical pairs was explained and related to the formal property of confluence of typed attributed graph transformations. In [13], critical pair analysis is used to detect conflicting requirements in independently developed use case models. In [12], critical pair analysis has been used to increase the efficiency of parsing visual languages by delaying conflicting rules as far as possible. In [28], graph transformation dependency analysis has been used for the purpose of detecting and resolving inconsistencies in design models.

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 [29]. Two other approaches that rely on graph transformation to tackle the problem of software merging were proposed by Westfechtel [30] and Mens [31]. 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. Also the application to refactoring transformations is new.

Refactoring is a very active research domain [2]. 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 [5,32–35]. To our knowledge, no attempt has been made to try and detect conflicts between refactorings applied in parallel.

A recent research trend is to apply refactoring techniques to *models* as opposed to programs. Boger *et al.* developed a refactoring browser integrated with a UML modelling tool [4]. It supports refactoring of class diagrams, state chart diagrams, and activity diagrams. Sunyé *et al.* formally defined some state chart refactorings using OCL pre- and post conditions [3]. Van Gorp *et al.* proposed a UML extension to express the pre- and post conditions of program refactorings using OCL [5,36], enabling an OCL empowered CASE tool to verify nontrivial pre and post conditions, to compose sequences of refactorings, and to use the OCL query engine to detect bad code smells. Such an approach is desirable as a way to refactor design models independent of the underlying programming language. Correa and Werner built further on these ideas, and implemented refactorings in OCL-script, an extension of OCL [7]. Porres implemented model refactorings as rule-based update transformations in SMW, a scripting language based on Python [6]. Zhang *et al.* developed a model transformation engine that integrates a model refactoring browser that automates and customises various refactoring methods for either generic models or domain-specific models [37].

9 Conclusion and Future Work

In the context of software refactoring, and to a lesser extent model refactoring, there are plenty of tools available that automate the process of applying refactoring transformations. Such tool support is missing, however, when it comes to suggesting a set of refactorings that can be used to improve the software structure, and assisting the developer to select the most appropriate refactoring. One of the reasons for this lack of tool support is the fact that there can be many implicit dependencies between refactorings. A pair of refactorings may be mutually exclusive, the application of a refactoring may depend on another one, or it may prohibit the application of another one.

The goal of this paper was to gain a deeper insight in these refactoring dependencies, and provide formally-founded tool support to analyse them. To achieve this, we represented software (programs or models) as typed attributed graphs that respect a type graph representing the object-oriented meta model. We specified refactorings as parameterised typed attributed graph transformation rules. Refactoring preconditions were specified by means of so-called negative application conditions.

To analyse dependencies between refactorings, we fed the above specifications into AGG, a state-of-the-art graph transformation tool. Its most salient feature (for the purpose of this article) is its built-in critical pair analysis algorithm. In this article we successfully explored how critical pair analysis can help a software developer to detect and analyse conflicts and dependencies between refactorings. It provides a first, but crucial, step towards better automated tool support for refactoring.

Obviously, a lot of work remains to be done. For example, some of the conflict situations that we expected to occur were not detected because our specification of refactorings was not sufficiently complete. In the experiments we carried out, each refactoring was specified by a single graph transformation rule. A full specification of some refactorings would require more than one rule, and sometimes even an infinite number of rules.[5] Currently, complex refactorings which are described by a set of rules to be applied in a controlled order are possible in AGG, but on the basis of Java programs only. Thus in this case, a refactoring is performed as “programmed graph transformation”. Furthermore, we are exploring whether and how we could use and implement graph transformation schemes and amalgamated graph transformations for this purpose [27].

While it was possible to automate the parallel dependency analysis of refactorings to a large extent with AGG, sequential dependency analysis still remains a largely manual process. Therefore, we are currently trying to include better support in AGG to automate this process.

Another open issue is how to deal with conflicts after they have been detected. From a formal point of view, one can rely on the technique of confluence analysis. We are currently exploring how to use this technique to incorporate conflict resolution strategies for refactorings.

For the proof of concept performed in this article, we deliberately made a number of simplifications to the object-oriented meta model that was represented as a type graph. To be more realistic, we need to enhance this type graph to model other kinds of object-oriented constructs such as local variables, super sends, interfaces, and so on. This will also require changes to the refactoring rules, and may even imply new refactorings. Note that a more sophisticated type graph will require the

[5] The same problem has been identified in [32].

use of type graphs with inheritance (similar to the way specialisation is used in the UML meta model). Therefore, this feature needs to be added to AGG as well.

Last but not least, the critical pair analysis takes a lot of time to compute. In order to make the approach more high-performance and, as such, more scalable to real-world situations, we are currently trying to improve the efficiency of the critical pair algorithm. Initial results on how to achieve this have been reported in [38].

References

1. Fowler, M.: Refactoring: Improving the Design of Existing Code. Addison-Wesley (1999)
2. Mens, T., Tourwé, T.: A survey of software refactoring. Transactions on Software Engineering **30** (2004) 126–139
3. G. Sunyé, Pollet, D., LeTraon, Y., J.-M. Jézéquel: Refactoring UML models. In: Proc. UML 2001. Volume 2185 of Lecture Notes in Computer Science., Springer-Verlag (2001) 134–138
4. Boger, M., Sturm, T., Fragemann, P.: Refactoring browser for UML. In: Proc. 3rd Int'l Conf. on eXtreme Programming and Flexible Processes in Software Engineering. (2002) 77–81 Alghero, Sardinia, Italy.
5. Van Gorp, P., Stenten, H., Mens, T., Demeyer, S.: Towards automating source-consistent UML refactorings. In Stevens, P., Whittle, J., Booch, G., eds.: UML 2003 - The Unified Modeling Language. Volume 2863 of Lecture Notes in Computer Science., Springer-Verlag (2003) 144–158
6. Porres, I.: Model refactorings as rule-based update transformations. In Stevens, P., Whittle, J., Booch, G., eds.: UML 2003 - The Unified Modeling Language. Volume 2863 of Lecture Notes in Computer Science., Springer-Verlag (2003) 159–174
7. Correa, A., Werner, C.: Applying refactoring techniques to UML/OCL models. In: Proc. Int'l Conf. UML 2004. Volume 3273 of Lecture Notes in Computer Science., Springer Verlag (2004) 173–187
8. Tourwé, T., Mens, T.: Identifying refactoring opportunities using logic meta programming. In: Proc. 7th European Conf. Software Maintenance and Re-engineering (CSMR 2003), IEEE Computer Society Press (2003) 91–100
9. van Emden, E., Moonen, L.: Java quality assurance by detecting code smells. In: Proc. 9th Working Conf. Reverse Engineering, IEEE Computer Society Press (2002) 97–107
10. Marinescu, R.: Using object-oriented metrics for automatic design flaws in large scale systems. In Demeyer, S., Bosch, J., eds.: Object-Oriented Technology (ECOOP'98 Workshop Reader). Volume 1543 of Lecture Notes in Computer Science., Springer-Verlag (1998) 252–253
11. Simon, F., Frank Steinbrückner, Lewerentz, C.: Metrics based refactoring. In: Proc. European Conf. Software Maintenance and Reengineering, IEEE Computer Society Press (2001) 30–38
12. Bottoni, P., Taentzer, G., Schürr, A.: Efficient parsing of visual languages based on critical pair analysis and contextual layered graph transformation. In: Proc. IEEE Symp. Visual Languages. (2000) 59–60
13. Hausmann, J.H., Heckel, R., Taentzer, G.: Detection of conflicting functional requirements in a use case-driven approach. In: Proc. Int'l Conf. Software Engineering, ACM Press (2002) 105–115

14. Heckel, R., Jochen Malte Küster, Taentzer, G.: Confluence of typed attributed graph transformation systems. In: Graph Transformation. Volume 2505 of Lecture Notes in Computer Science., Springer-Verlag (2002) 161–176
15. Demeyer, S., Janssens, D., Mens, T.: Simulation of a LAN. Electronic Notes in Theoretical Computer Science **72** (2002)
16. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Patterns: Elements of Reusable Object-Oriented Languages and Systems. Addison-Wesley (1994)
17. Plump, D.: Hypergraph Rewriting: Critical Pairs and Undecidability of Confluence. In Sleep, M., Plasmeijer, M., van Eekelen, M.C., eds.: Term Graph Rewriting. Wiley (1993) 201–214
18. Ehrig, H., Prange, U., Taentzer, G.: Fundamental theory for typed attributed graph transformation. In Parisi-Presicce, F., Bottoni, P., Engels, G., eds.: Proc. 2nd Int'l Conf. Graph Transformation (ICGT'04), Rome, Italy. Volume 3256 of Lecture Notes in Computer Science. Springer (2004) 161–177
19. Corradini, A., Montanari, U., Rossi, F.: Graph processes. Fundamenta Informaticae **26** (1996) 241–265
20. Habel, A., Heckel, R., Taentzer, G.: Graph Grammars with Negative Application Conditions. Fundamenta Informaticae **26** (1996) 287–313
21. Opdyke, W.F.: Refactoring: A Program Restructuring Aid in Designing Object-Oriented Application Frameworks. PhD thesis, University of Illinois at Urbana-Champaign (1992)
22. Roberts, D., Brant, J., Johnson, R.E.: A refactoring tool for Smalltalk. Theory and Practice of Object Systems **3** (1997) 253–263
23. Roberts, D.B.: Practical Analysis for Refactoring. PhD thesis, University of Illinois at Urbana-Champaign (1999)
24. Tichelaar, S., Ducasse, S., Demeyer, S., Nierstrasz, O.: A meta-model for language-independent refactoring. In: Proc. Int'l Symp. Principles of Software Evolution, IEEE Computer Society Press (2000) 157–169
25. Tichelaar, S.: Modeling Object-Oriented Software for Reverse Engineering and Refactoring. PhD thesis, University of Bern (2001)
26. Mens, T., Taentzer, G., Runge, O.: Detecting structural refactoring conflicts using critical pair analysis. Electronic Notes in Theoretical Computer Science (2004)
27. Taentzer, G.: Parallel and Distributed Graph Transformation: Formal Description and Application to Communication-Based Systems. PhD thesis, TU Berlin (1996) Shaker Verlag.
28. Mens, T., Van Der Straeten, R., D'Hondt, M.: Detecting and resolving model inconsistencies using transformation dependency analysis. In Nierstrasz, O., Whittle, J., Harel, D., Reggio, G., eds.: Model Driven Engineering Languages and Systems. Volume 4199 of Lecture Notes in Computer Science., Springer-Verlag (2006) 200–214
29. Mens, T.: A state-of-the-art survey on software merging. Transactions on Software Engineering **28** (2002) 449–462
30. Westfechtel, B.: Structure-oriented merging of revisions of software documents. In: Proc. Int'l Workshop on Software Configuration Management, ACM Press (1991) 68–79
31. 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. Volume 1779 of Lecture Notes in Computer Science., Springer-Verlag (2000) 127–143
32. Van Eetvelde, N., Janssens, D.: Extending graph rewriting for refactoring. In: Graph Transformations. Volume 3526 of Lecture Notes in Computer Science., Springer-Verlag (2004) 399–415 Proc. Second Int'l Conf. Graph Transformation (ICGT), Rome, Italy, September-October 2004.

33. Bottoni, P., Parisi-Presicce, F., Taentzer, G.: Specifying Integrated Refactoring with Distributed Graph Transformation. In Pfaltz, J., Nagl, M., Boehlen, B., eds.: Application of Graph Transformations with Industrial Relevance (AGTIVE'03). Volume 3062 of Lecture Notes in Computer Science., Springer (2004) 220–235
34. Bottoni, P., Parisi-Presicce, P., Taentzer, G.: Specifying Coherent Refactoring of Software Artefacts with Distributed Graph Transformations. In v. Bommel, P., ed.: Transformation of Knowledge, Information, and Data: Theory and Applications, Idea Group Publishing (2005)
35. Mens, T., Van Eetvelde, N., Demeyer, S., Janssens, D.: Formalizing refactorings with graph transformations. Software Maintenance and Evolution: Research and Practice **17** (2005) 247–276
36. Schippers, H., Van Gorp, P., Janssens, D.: Leveraging UML profiles to generate plugins from visual model transformations. Electronic Notes in Theoretical Computer Science (2004)
37. Jing Zhang, Yuehua Lin, J.G.: Generic and Domain-Specific Model Refactoring using a Model Transformation Engine. In: Model-driven Software Development - Research and Practice in Software Engineering. Springer Verlag (2005)
38. Lambers, L., Ehrig, H., Orejas, F.: Efficient detection of conflicts in graph-based model transformation. In: Proc. International Workshop on Graph and Model Transformation (GraMoT'05). Volume 152 of Electronic Notes in Theoretical Computer Science., Elsevier Science (2006) 97–109