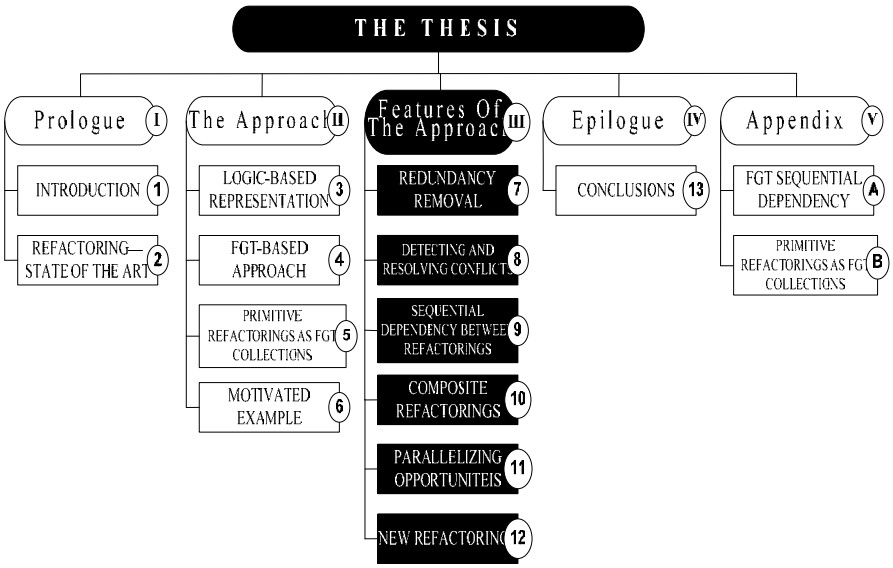
Part III

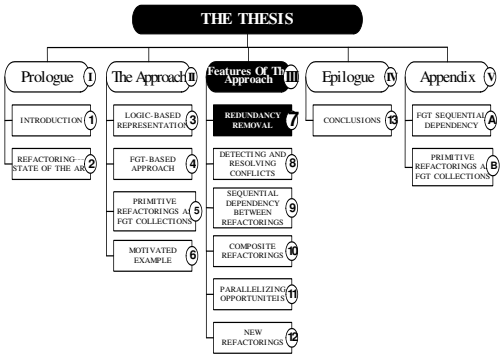
Features Of The Approach



CHAPTER 7

REDUNDANCY REMOVAL

7.1 Introduction



Applying refactorings on a system can be a time-consuming process, especially when the refactorings are to be applied to a large system. The cost is caused by checking the precondition of the refactoring and by running the required transformation operations on the system. For example, one of the precondition conjuncts of the **deleteMethod** primitive refactoring is that the method should not be referenced anywhere in the system. In this case, the refactoring tool has to check the entire system to look for references to that method. In addition to checking refactoring precondition, some refactorings cause a lot of changes (restructuring) to the system, and this in turn implies executing multiple transformation operations. For example, in the primitive refactoring **moveMethod** described in section 5.3.2.2 many transformation operations are needed to implement the refactoring. The cost is correspondingly higher in the case of composite refactorings, since these may have a significant number of precondition conjuncts that need to be checked, and may significantly restructure the system.

In some cases, a collection of refactorings may embody redundancies. Redundancy occurs whenever a subset of transformation actions undertaken to refactor a system turns out to be unnecessary. In an extreme case, the entire refactoring may have no effect at all on the original system. Redundancy might mean that needless work and effort are done by the refactoring tool, as the following two sections describe.

Previous approaches do not allow for the removal of such redundancies, because refactoring is implemented as a sequence of code blocks (*black box*). No meta-information is available to the refactoring tool to indicate what each part of the code does, and consequently, the tool has no ability to optimise the code.

One of the advantages of dealing with refactoring as a collection of FGTs is that opportunities become available to remove such redundancies. We call this process a *reduction process*. The

final effects of the refactoring on the system after the reduction process is the same as the final effects without any reduction. Two types of reductions can be identified: absorbing reductions and cancelling reductions. In the following two sections, each one of the two types will be discussed.

7.2 Absorbing Reduction

This kind of reduction occurs when two FGTs can be absorbed by one that has the same effect as the two. For example, suppose that the user wants to add a new method *m1* in class *P.A*. To do this the following FGT is used:

addObject(*P, A, m1, _, _, type(basic, void, 0), public, [], method*)

To apply this FGT on the system the refactoring tool has to check its set of precondition conjuncts as described in section 4.2.1.1.B.

Suppose that the user then decides to rename the method *m1* in the class *P.A* to another name *m2*. To do this the following FGT is used:

renameObject(*P, A, m1, _, [], method, m2*)

To apply this FGT on the system the refactoring tool also has to check its set of precondition conjuncts as described in section 4.2.1.2.B.

In accomplishing these tasks, suppose that the refactoring tool carries out the following steps:

a. Check the set of precondition conjuncts of the FGT addObject. Suppose that the refactoring tool performs this check with effort *E1*.

b. Apply the FGT addObject to the system. Suppose that the refactoring tool performs the required transformations with effort *E2*.

c. Check the set of precondition conjuncts of the FGT renameObject. Suppose that the refactoring tool performs this checking with effort *E3*.

d. Apply the FGT renameObject on the system. Suppose that the refactoring tool performs the required transformations with effort *E4*.

The total effort ($T_{effort1}$) required by the refactoring tool to accomplish the previous scenario is therefore:

$$T_{effort1} = E1 + E2 + E3 + E4$$

Alternatively, the refactoring tool can test for redundancies implied by the two FGTs and, if found, make a suitable reduction. To process the scenario, the refactoring tool carries out the following steps:

a. Build the FGT-DAGs of an FGT-list. Suppose that the refactoring tool builds the FGT-DAGs with effort *E5*,

b. Execute the reduction algorithm on the generated FGT-DAGs. In our example, the tool will discover that a redundancy is implicit in the two FGTs. As a result the two FGTs will be absorbed into one FGT that has the same effect as the two:

   addObject(*P, A, m2, _, _, type(basic, void, 0), public, [], method*)

   Suppose that the refactoring tool performs the reduction with effort *E6*.

c. Check the set of the precondition conjuncts of the FGT addObject (with effort *E7*).

d. Apply the FGT on the model (with effort *E8*).

The total effort ($T_{effort2}$) required of the refactoring tool to accomplish the previous scenario is then:

$$T_{effort2} = E5 + E6 + E7 + E8$$

Assume that *E5* and *E6* are likely to be small because they are simple internal processes inside the refactoring tool that, in most of the cases, do not need to reference the underlying representation of the system. Let α =E5 + E6, then

$$T_{effort2} = \alpha + E7 + E8$$

To compare $T_{effort1}$ with $T_{effort2}$ note that $E1 = E7$ and $E2 = E8$ so that

$$T_{effort1} - T_{effort2} = E3 + E4 - \alpha$$

Assuming that α is much smaller than ($E3 + E4$), the total effort of the tool with reduction would be much less than the total effort without reduction ($T_{effort2} << T_{effort1}$). However, for a further discussion of these matters refer to 7.7.

Table 7.1 gives the absorbing reductions that may exist between the different pairs of FGTs. Each pair of FGTs that can be reduced is called a *reduction-pair*. The left column of the table shows the reduction-pairs, and the right column of the table shows the suitable FGT that absorbs the pair in the left column. Information in the table is stored as facts in the Prolog database. Examples of these facts are shown in Figure 7.1.

Table 7.1: Absorbing reduction

No	Reduction-Pairs	Absorbed By
1.	renameObject(*P,C,M,PR,LT,parameter,X*) → renameObject(*P,C,M,X,LT,parameter,Y*)	renameObject(*P,C,M,PR,LT,parameter,Y*)
2.	renameObject(*P,C,M,_,_,attribute,X*) → renameObject(*P, C,X,_,_,attribute,Y*)	renameObject(*P,C,M,_,_,attribute,Y*)
3.	renameObject(*P,C,M,_,LT,method,X*) → renameObject(*P,C,X,_,LT,method,Y*)	renameObject(*P,C,M,_,LT,method,Y*)
4.	renameObject(*P,C,_,_,_,class,X*) → renameObject(*P,X,_,_,_,class,Y*)	renameObject(*P,C,_,_,_,class,Y*)
5.	changeOAMode(*P,C,M,PR,LT,ObjT,X,Y*) → changeOAMode(*P,C,M,PR,LT,ObjT,Y,Z*)	changeOAMode(*P,C,M,PR,LT,ObjT,X,Z*)
6.	changeODefType(*P,C,M,PR,LT,ObjT,X,Y*) → changeODefType(*P,C,M,PR,LT,ObjT,Y,Z*)	changeODefType(*P,C,M,PR,LT,ObjT,X,Z*)
7.	addObject(*P,C,M,X,T1,T2,T4,T5,parameter*) → renameObject(*P,C,M,X,T5,parameter,Y*)	addObject(*P,C,M,Y,T1,T2,T4,T5, parameter*)
8.	addObject(*P,C,X,_,T2,T3,T5,_,attribute*) → renameObject(*P,C,X,_,_,attribute,Y*)	addObject(*P,C,Y,_,T2,T3,T5,_,attribute*)
9.	addObject(*P,C,X,_,T2,T3,T5,T6,method*) → renameObject(*P,C,X,_,T6,method,Y*)	addObject(*P,C,Y,_,T2,T3,T5,T6,method*)
10.	addObject(*P,X,_,_,T2,T3,T5,_,class*) → renameObject(*P,X,_,_,_,class,Y*)	addObject(*P,Y,_,_,T2,T3,T5,_,class*)
11.	addObject(*P,C,M,PR,T1,X ,T2,T3,ObjT*) → changeODefType(*P,C,M,PR,T3,ObjT,X,Y*)	addObject(*P,C,M,PR,T1,Y ,T2,T3,ObjT*)
12.	addObject(*P,C,M,PR,T1,T2 ,X,T4,ObjT*)→ changeOAMode(*P,C,M,PR,T4,ObjT,X,Y*)	addObject(*P,C,M,PR,T1,T2 ,Y,T4, ObjT*)
13.	changeOAMode(*P,C,M,PR,LT,ObjT,X,Y*) → deleteObject(*P,C,M,PR,ObjT*)	deleteObject(*P,C,M,PR,ObjT*)
14.	changeODefType(*P,C,M,PR,LT,ObjT,X,Y*) → deleteObject(*P,C,M,PR,ObjT*)	deleteObject(*P,C,M,PR,ObjT*)
15.	renameObject(*P,C,M,PR,LT,parameter,PR1*) → deleteObject(*P,C,M,PR1,LT,parameter*)	deleteObject(*P,C,M,PR,LT,parameter*)
16.	renameObject(*P,C,M,_,_,attribute,M1*) → deleteObject(*P,C,M1,_,_,attribute*)	deleteObject(*P,C,M,_,LT,attribute*)
17.	renameObject(*P,C,M,_,LT,method,M1*) → deleteObject(*P,C,M1,_,LT,method*)	deleteObject(*P,C,M,_,LT,method*)
18.	renameObject(*P,C,_,_,_,class,M1*) → deleteObject(*P,M1,_,_,_,class*)	deleteObject(*P,C,_,_,_,class*)

19.	renameRelation(*L1,P,C,M,PR,LT,Ftype,P1,C1,M1,PR1, LT1,Totype,Ltype,L2*) → renameRelation(*L2,P,C,M,PR,LT,Ftype,P1,C1,M1,PR1, LT1,Totype,Ltype,L3*)	renameRelation(*L1,P,C,M,PR,LT,Ftype,P1,C1, M1, PR1, LT1,Totype,Ltype,L3*)
20.	addRelation(*L1,P,C,M,PR,LT,Ftype,P1,C1,M1,PR1,LT1, Totype,Ltype*) → renameRelation(*L1,P,C,M,PR,LT,Ftype,P1,C1,M1,PR1, LT1,Totype,Ltype,L2*)	addRelation(*L2,P,C,M,PR,LT,Ftype,P1,C1,M1, PR1,LT1,Totype,Ltype*)
21.	renameRelation(*L1,P,C,M,PR,LT,Ftype,P1,C1,M1,PR1, LT1,Totype,Ltype,L2*) → deleteRelation(*L2,P,C,M,PR,LT,Ftype,P1,C1,M1,PR1, LT1,Totype,Ltype*)	deleteRelation(*L1,P,C,M,PR,LT,Ftype,P1,C1,M1, PR1,LT1,Totype,Ltype*)

Here is a detailed explanation of each pair of the absorbing reduction:

- Reduction-pairs 1-4 in the table concern the FGT renameObject. If a first FGT renames the *object* element from *name1* to *name2*, and then a second FGT renames the same *object* element from *name2* to *name3,* then these two FGTs can be absorbed by one which will rename the *object* from *name1* to *name3*.

- Reduction-pair 5 concerns the FGT changeOAMode. If a first FGT changes the access mode of the object element from *X* to *Y*, and then a second FGT changes the access mode of the object element form *Y* to *Z* then these two FGTs can be absorbed by one which will change the access mode of the object from *X* to *Z*.

- Reduction-pair 6 is the same as the previous one but for changeODefType.

- Reduction-pairs 7-10 concern the FGTs addObject and renameObject. If the FGT addObject adds a specific object with name *X*, and then a second FGT renameObject changes the name of the same object from *X* to *Y,* then these two FGTs can be absorbed by addObject that adds the object with name *Y* from the beginning.

- Reduction-pair 11 concerns the FGTs addObject and changeODefType. If the FGT addObject adds a specific object with a definition type *X*, and then a second FGT changeODefType changes the definition type of the same object from *X* to *Y* then these two FGTs can be absorbed by one addObject that adds the same object with the definition type *Y* from the beginning.

- Reduction-pair 12 is the same as the previous one but for changeOAMode.

- Reduction-pair 13 concerns the FGTs changeOAMode and deleteObject. If the FGT changeOAMode changes the access mode of specific object from *X* to *Y*, and then a second FGT deleteObject deletes the same object from the system, then there is no need to change

the access mode of the object that is going to be deleted in a later stage so these two FGTs can be absorbed by one deleteObject. Note that because the access mode of the object not appears in the FGT deletObject, then in this case, we just remove the FGT changeOAMode from the collection.

- Reduction-pair 14 is the same as the previous one but for changeODefType.
- Reduction-pair 15-18 concern with the FGTs renameObject and deleteObject. If the FGT renameObject changes the name of specific object from *X* to *Y*, and then a second FGT deleteObject deletes object *Y* from the system then in this case, there is no need to change the name of the object that is going to be deleted in a later stage so these two FGTs can be absorbed by one deleteObject that will delete the object with name *X* (old name).
- Reduction-pair 19 is the same as reduction-pairs 1-4 but for renameRelation.
- Reduction-pair 20 is the same as reduction-pairs 7-10 but for FGTs addRelation and renameRelation.
- Reduction-pair 21 is the same as reduction-pairs 15-18 but for FGTs renameRelation and deleteRelation.

7.3 Cancelling Reduction

This kind of reduction occurs when two FGTs cancel each other. For example, suppose that a user adds a new method *m1* in class *P.A*. To do this, he uses the following FGT:

addObject(*P, A, m1, _, _, type(basic, void, 0), public, [], method*)

To apply the FGT, the refactoring tool has to check the relevant set of precondition conjuncts. Suppose that the method *m1* in the class *P.A* is subsequently deleted. To do this the following FGT is used:

deleteObject(*P, A, m1, _, [], method*)

It is clear that there is a reduction between the two FGTs. The refactoring tool will discover that there is a cancelling reduction between the two FGTs. As a result, the two FGTs will be removed from the refactoring collection. In this case, the only effort needed of the tool is to build the FGT-DAGs and carry out the reduction process. If we assume that *E1*, *E2* stands for

checking precondition conjuncts and applying FGT addObject and *E3*, *E4* stands for checking precondition conjuncts and applying FGT deleteObject then the effort of the refactoring tool without reduction will be:

$$T_{effort1} = E1 + E2 + E3 + E4$$

While the effort of the refactoring tool in case of using the reduction will be

$$T_{effort2} = \alpha$$

where α is the effort to build the FGT-DAGs and execute reduction process. As assumed before that α is small because it is an internal process within the tool. Therefore, it can be concluded that

$$T_{effort2} << T_{effort1.}$$

Table 7.2 gives the various possibilities for the cancelling reductions between different pairs of FGTs. Information in the table is stored as facts in the Prolog database, part of these facts are shown in Figure 7.1.

Table 7.2: Cancelling reduction

No	Reduction-Pairs
1.	deleteObject(*P,C,M,PR,LT,ObjT*) → addObject(*P,C,M,PR,_,_,_,LT,ObjT*)
2.	addObject(*P,C,M,PR,_,_,_,LT,ObjT*) → deleteObject(*P,C,M,PR,LT,ObjT*)
3.	deleteRelation(*R1,P,C,M,PR,LT,Ftype,P1,C1,M1,PR1,LT1,Totype,Ltype*) → addRelation(*R1,P,C,M,PR,LT,Ftype,P1,C1,M1,PR1,LT1,Totype,Ltype*)
4.	addRelation(*R1,P,C,M,PR,LT,Ftype,P1,C1,M1,PR1,LT1,Totype,Ltype*) → deleteRelation(*R1,P,C,M,PR,LT,Ftype,P1,C1,M1,PR1,LT1,Totype,Ltype*)

Here is a detailed explanation of each pair of the cancelling reduction:

- Reduction-pairs 1-2 in the table concern the FGTs deleteObject and addObject. To delete an *object* element from the system using FGT deleteObject and then to add the same *object* to the system using FGT addObject clearly has no effect on the system. The same also apply when add an *object* element to the system using FGT addObject and then to delete the same *object* element from the system using FGT deleteObject. As a result, the two FGTs need to be removed by the refactoring tool.

- Reduction-pairs 3-4 in the table are similar to the previous reduction-pair, but involve addRelation and deteteRelation.

```
……..
reduction(renameObject(P,C,_,_,_,class,X),renameObject(P,X,_,_,_,class,Y),
renameObject(P,C,_,_,_,class,Y)).

reduction(addObject(P,C,X,_,T2,T3,T4,T5,T6,method), renameObject(P,C,X,_,
T6,method,Y), addObject(P,C,Y,_,T2,T3,T4,T5,T6,method)).

reduction(addObject(P,C,M,PR,_,_,_,_,LT,ObjT),deleteObject(P,C,M,PR,LT,ObjT),
'Cancel Both').
……..
```

Figure 7.1: Part of the reduction facts as implemented in Prolog

7.4 Advantages of Reduction Process

The reduction process has the following advantages:

1. The number of FGTs and number of refactoring precondition conjuncts are reduced, thus increasing the efficiency of refactoring. Clearly, when an FGT is cancelled or absorbed by the reduction process, then its set of precondition conjuncts will also be cancelled or absorbed. One advantage of distributing the precondition conjuncts of the refactoring into two levels (FGT-level and refactoring-level) is the ability to cancel or absorb these precondition conjuncts by the reduction process.

2. The number of sequential dependencies between the different FGTs inside the refactoring will be reduced. This will increase the number of FGT-DAGs for that refactoring which means that the parallelizing opportunities at the time of refactoring will be increased.

3. Pseudo-conflicts may be eliminated. For example, suppose that we have the following two refactorings:

R_x: {……..
……..
addObject (*P, A , m1, _, _, type(basic, void, 0), public, [], method*)
……..
deleteObject (*P, A, m1, [], method*)
……..
……..}

R_y: {……..
……..
addObject (*P, A, m1, _, _, type(basic, void, 0), public, [], method*)
……..
..……....}

Trying to apply refactorings R_x and R_y concurrently on the system (or in the order R_y then R_x) could cause a conflict between the two refactorings because both of the two refactorings try to add the same method *m1* in the class *P.A* (conflicts between concurrent refactorings will be discussed in the next chapter). To solve this conflict one of the two refactorings could be cancelled or R_x could be executed before R_y.

In fact, the conflict between refactorings R_x and R_y is pseudo-conflict. If a reduction process discovers that there is a cancelling reduction between FGTs addObject and deleteObject in R_x and removes these two FGTs from the collection of FGTs of refactoring R_x, then there will be no conflict between R_x and R_y.

7.5 Reduction Algorithm

A reduction algorithm has been developed that takes an arbitrary FGT-DAG as input, and removes all redundancies in this data structure. This algorithm may be invoked to remove possible redundancies from some FGT-DAGs that represent a refactoring. Its use also will be seen in chapter 10 in the context of composite refactorings.

It can easily be verified that FGTs in each reduction-pair are sequentially dependent. If they appear as adjacent nodes in an FGT-DAG then they may be redundant. The reduction algorithm is based on this. It simply traverses an FGT-DAG and searches for adjacent reduction-pairs. When one is found, then the algorithm makes the suitable reduction and appropriately restructures the rest of nodes in that FGT-DAG. As will be seen, this reduction may result in new FGT-DAGs being created out of parts of the original FGT-DAG in which the redundancy was found.

As mentioned before, the refactoring may consist of more than one FGT-DAG. Since these are sequentially independent, the different instances of the reduction algorithm can work concurrently on each FGT-DAG.

Algorithm 7.1 gives the pseudo-code for the **reduction** algorithm. The algorithm takes as a parameter an FGT-DAG. It traverses the FGT-DAG from root to leaves in a depth-first fashion, searching for occurrences of reduction-pairs ($node_i$ and $node_j$) using the reduction facts. If a reduction-pair is found then the corresponding reduction is made and the links between the remaining FGTs in the FGT-DAG are changed properly.

If one of the cancelling reduction-pairs is found, then the reduction-pair ($node_i$, $node_j$) will be removed from the FGT-DAG. All links into and out of $node_i$ and $node_j$ will also be removed. In addition, the algorithm will check for sequential dependencies between each father of $node_i$ and the sons of both $node_i$ and $node_j$. If there is a sequential dependency then a new link will be created between the relevant father and son nodes. The same is done with respect to each father of $node_j$ and sons of both $node_i$ and $node_j$. As will be illustrated below, the above process may result in one or more FGT-DAGs being formed from parts of the old FGT-DAG.

If an absorbing reduction-pair is found, then the reduction-pair ($node_i$, $node_j$) will be removed from the FGT-DAG as well as all links related to these nodes. A new node called $node_x$ as specified by the relevant reduction fact is then inserted into the FGT-DAG. Again, the algorithm will check for sequential dependencies, in this case between each father of $node_i$ (and of $node_j$) and the newly created $node_x$. If a sequential dependency is found, then a new link will be created between the relevant father node and $node_x$. Similarly, a check for sequential dependencies will be made between $node_x$ and each son of $node_i$ and each son of $node_j$. Again, wherever a sequential dependency is found, a new link will be created between $node_x$ and the relevant son node. Also in this case, it is possible that the above process results in one or more FGT-DAGs being formed from parts of the old FGT-DAG.

Algorithm 7.1 (Reduction algorithm)

reduction (IN-DAG)
Input: IN-DAG: An FGT-DAG
Output: RED-DAGS: A redundancy-free set of FGT-DAGs

Insert IN-DAG into RED_DAGS
For each unexamined pair of adjacent nodes ($node_i$, $node_j$) in RED-DAGS **do** {
  //Search **reduction facts** for a match between ($node_i$, $node_j$)
  If ($node_i$, $node_j$) is a cancelling reduction-pair
    then {
      Let F = set of father nodes of $node_i$ and father nodes of $node_j$ (excluding $node_i$)
      Let S = set of son nodes of $node_i$ (excluding $node_j$) and son nodes of $node_j$
      Let FS = F X S
      For each pair ($node_f$, $node_s$) in FS **do** {
        If ($node_f$, $node_s$) sequentially dependent
          then insert link from $node_f$ to $node_s$

} //end **For**

Remove ($node_i$, $node_j$) from IN-DAG

Remove all links into and out of $node_i$, and $node_j$

} //end **If**

Else If ($node_i$, $node_j$) is an absorbing pair

then {

Let $node_x$=absorbing FGT of ($node_i$, $node_j$)

Let F = set of father nodes of $node_i$ and father nodes of $node_j$ (excluding $node_i$)

Let S = set of son nodes of $node_i$ (excluding $node_j$) and son nodes of $node_j$

For each $node_f$ in F **do** {

If ($node_f$, $node_x$) sequentially dependent

then insert link from $node_f$ to $node_x$

} //end for

For each $node_s$ in S **do** {

If ($node_x$, $node_s$) sequentially dependent

then insert link from $node_x$ to $node_s$

} //end **For**

Remove ($node_i$, $node_j$) from IN-DAG

} //end **Else If**

Collect all FGT-DAGs produced by the foregoing into RED-DAGS

} //end **For**

Return RED-DAGS

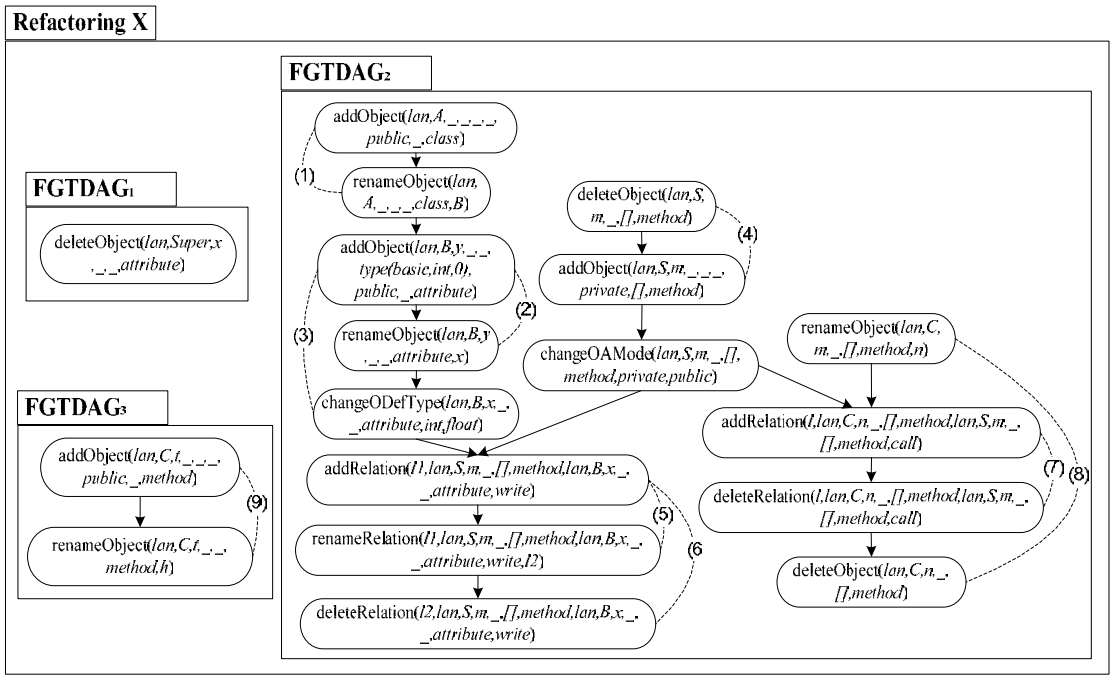
Note that the for-loop in the algorithm is not specific about the order in which adjacent nodes are examined. It does require, however, that new adjacent pairs that may be added into the FGT-DAGs in RED-DAGS have to be examined. The actual order to be used is an implementation issue. In the prototype tool, a top-down approach has been followed.

7.6 Example

To illustrate the reduction idea, a fictitious example as shown in Figure 7.2 is used. In the example, refactoring X consists of three independent FGT-DAGs ($FGT\text{-}DAG_1$, $FGT\text{-}DAG_2$ and $FGT\text{-}DAG_3$). The **reduction** algorithm will work on each one of the three FGT-DAGs separately.

For FGT-DAG$_2$ for example, the order in which the algorithm examines node pairs is indicated by the numbering on the dashed lines of Figure 7.2(a). Figure 7.2(b) shows the reduction action that the algorithm takes for each reduction-pair.

Figure 7.3 shows refactoring X after being reduced. Note that the number of reductions depends on the type of FGTs in the refactoring. The number of FGTs in the example has been reduced from 18 FGTs to 6 FGTs. Note also that the number of FGT-DAGs has been increased from 3 to 5 after reduction.



(a)

Reduction Actions

(1) Absorbed by addObject(*lan,B,_,_,_,_,public,_,class*)
(2) Absorbed by addObject(*lan,B,x,_,_,type(basic,int,0),public,_,attribute*)
(3) Absorbed by addObject(*lan,B,y,_,_,type(basic,float,0),public,_,attribute*)
(4) Delete both
(5) Absorbed by addRelation(*l2,lan,S,m,_,[],method,lan,B,x,_,_,attribute,write*)
(6) Delete both
(7) Delete both
(8) Absorbed by deleteObject(*lan,C,n,_,[],method*)
(9) Absorbed by addObject(*lan,C,h,_,_,_, public,[],method*)

(b)

Figure 7.2: Reduction inside refactoring

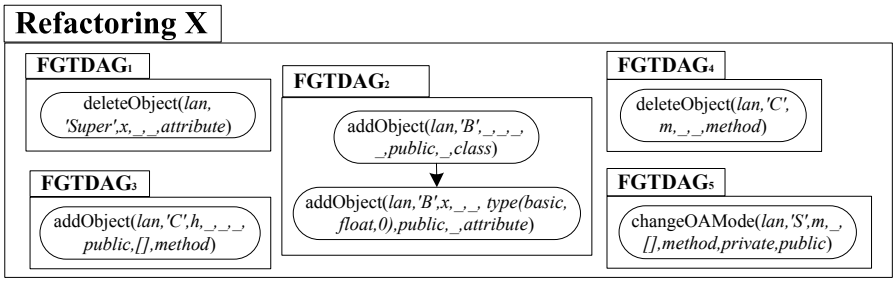


Figure 7.3: Refactoring X after reduction

7.7 Efficiency Considerations

The possibility that a given FGT-DAG may embody redundancies, and that these may be removed, is certainly of theoretical interest. How such redundancies may come about in practice is an open question. It may be, for example, that they are specified by a naive user attempting to use FGTs to specify a transformation or refactoring. Alternatively, redundancies may arise in a multi-user environment where different FGT-lists are merged.

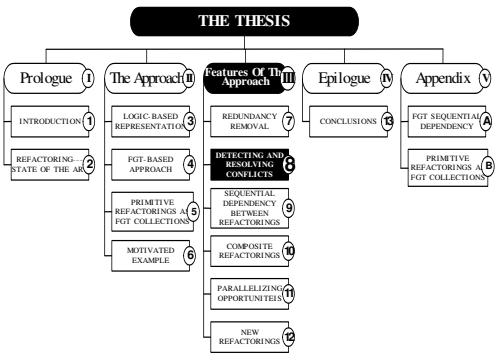
The question of whether or not it is efficient to remove redundancies from an FGT-DAG is also context-dependent. There are definite gains to be had in reducing the number of changes to the underlying system. If the system is large, its underlying representation is correspondingly large and unnecessarily searches into the data are best avoided. In contrast, the database of facts recording redundancy pairs and sequential dependencies is not system-dependent and can be accessed relatively efficiently for the purposes of setting up or changing FGT-DAGs. However, the overall cost of setting up FGT-DAGs and reducing them is also dependent on the originating FGT-list.

Notwithstanding these context-dependent efficiency considerations, in forthcoming chapters, all relevant FGT-DAGs will be considered to be redundancy-free.

Chapter 8

DETECTING AND RESOLVING CONFLICTS

8.1 Introduction



This chapter and the next consider two refactorings, R_i and R_j. How these refactorings came into being is not relevant. What is assumed to be known is the pre- and postconditions of the two refactorings. In addition, it is assumed that the internal composition of each refactoring is known in terms of a set of FGT-DAGs. The two respective chapters then enquire into the question of whether the two refactorings are related in a manner that constrains the way in which they can be applied to a system. There are three possible answers to this question.

1. They are entirely unrelated. In this case, they can be applied in any order to a system (However, without information about precisely what changes are made to the system while they are being applied, it cannot be asserted that they may safely run concurrently.)

2. There is some order in which the two refactorings have to be applied—either R_i then R_j or vice versa. This is the subject of enquiry in chapter 9.

3. It is not possible to apply both refactorings on any system. This is the subject of enquiry in the present chapter—namely, the matter of detecting conflicts, and possibly resolving them.

Conflict between refactoring R_i and refactoring R_j occurs when it is the case that applying them in a given order will make the later one inapplicable. By this, is meant that when the first refactoring is applied to the system it will change the state of the system in a way that makes the precondition of the second one inapplicable. Thus, the postcondition of the first will conflict with the precondition of the second.

For example, suppose in a multi-developer environment, two developers try to apply refactorings R_i and R_j to the same system. Assume that the system has a package P with one class C. Assume—as shown in Figure 8.1—that part of the transformations that the two refactorings intend to make on the system are as the follows: refactoring R_i adds a new class A

in the package *P* and refactoring R_j changes the name of the existing class *C* in the package *P* to a new name *A*. Note that part of the precondition conjuncts of the two refactorings is the non-existence of a class with name *A* in the package *P*.

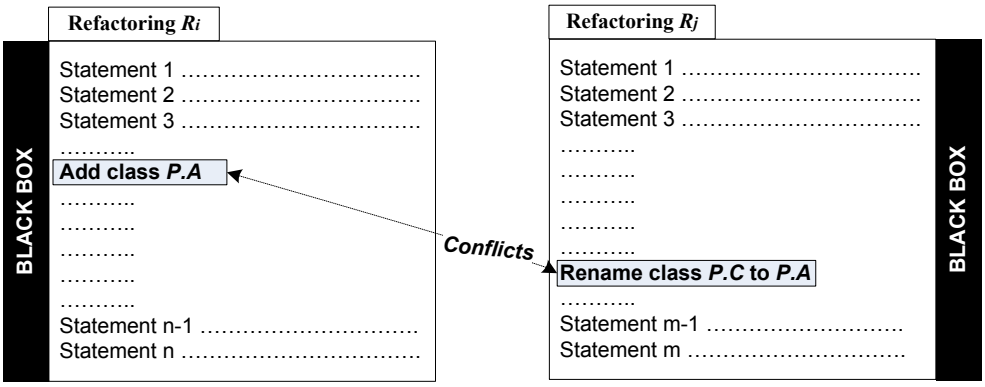


Figure 8.1: Conflicts between refactorings R_i & R_j

In the previous example, three possible scenarios may be envisaged:

1. Apply refactoring R_i then refactoring R_j. In this case, the tool will check the precondition conjuncts of R_i, discovering that they are satisfied because class *A* is not defined in the package *P*. Therefore, the tool will apply refactoring R_i to the system which means that the class *A* will be defined in the package *P* by this refactoring. Then at the time of applying refactoring R_j, the tool will check the precondition conjuncts of R_j, discovering that they do not hold because class *A* is defined now in the package *P*.

2. Apply refactoring R_j then refactoring R_i. In this case, the tool will check the precondition conjuncts of R_j, discovering that they are satisfied because class *A* is not defined in the package *P*. Therefore, the tool will apply refactoring R_j to the system which means that the class *A* will be defined in the package *P* by this refactoring. Then at the time of applying refactoring R_i, the tool will check the precondition conjuncts of R_i, discovering that they do not hold because class *A* is defined now in the package *P*.

3. Apply refactorings R_i and R_j simultaneously. In this case when the refactoring tool checks the precondition conjuncts of refactoring R_i and R_j, it will find that the precondition conjuncts of the two refactorings are satisfied because class *A* is not defined in package *P* at that time. Then the tool will start applying the two refactorings to the system simultaneously which will end up with an inconsistency because at the end of the two refactorings, package *P* will have two classes with the same name *A*.

To avoid such problems, a refactoring tool should have the capability to detect and resolve conflicts that may occur between multiple refactorings.

Tools based on previous refactoring approaches can potentially be designed to detect that there is a conflict between two refactorings. One approach is based on checking the pre- and postconditions of the two refactorings [38, 39, 52, and 55]. In the example presented in Figure 8.1, part of the precondition conjuncts of refactorings R_i and R_j is the non-existence of class *A* in package *P*. In addition, part of the postcondition conjuncts of the two refactorings is the existence of class *A* in package *P*. From this information, the refactoring tool could infer that there is a conflict between the two refactorings. Another approach proposed in the graph transformation community. The approach is based on the technique of critical pair analysis [55, 57, and 58].

However, it is difficult for such a tool to detect which specific parts of the two refactorings cause the conflict, and therefore to take possible corrective steps that could potentially resolve the conflict.

8.2 Conflicts in FGT-Based Approach

To detect and resolve conflicts between multiple refactorings in an FGT-based approach, it is sufficient to detect and resolve conflicts at the level of those FGTs which make up these refactorings. Consider the refactorings R_i and R_j discussed in the previous example but now shown in Figure 8.2 as a collection of FGTs ordered in FGT-DAGs. In such a scenario, the refactoring tool can check for conflicts between every pair of FGTs in the two refactorings.

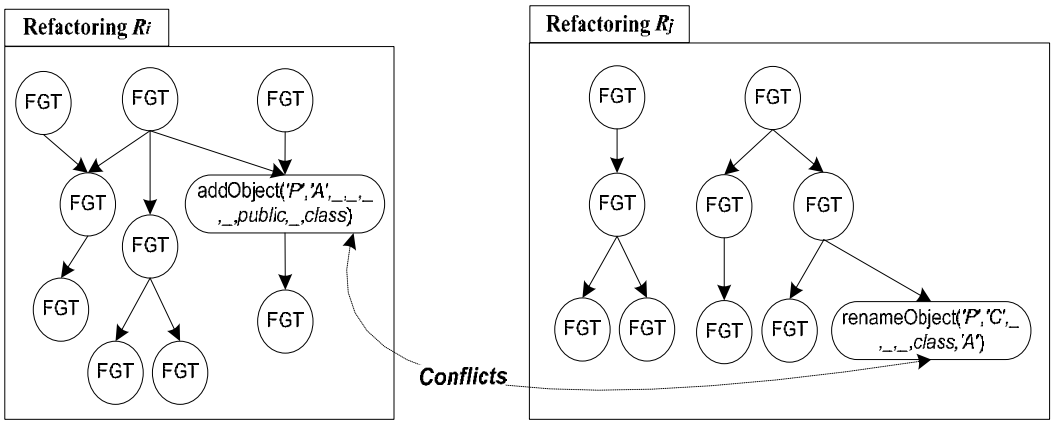


Figure 8.2: Conflict detection in FGT-based approach

This approach is close to what is called the **operation-based merge** approach [16, 21, 32, 50, 62, and 80] that is used to find conflicts between multiple versions of software that need to be merged after being changed/evolved by multiple developers. To detect the merge conflicts in such approaches, there is no need to compare all versions in their entirety—it suffices to compare only the evolution operations that have been applied to obtain each of the versions. In the present context, these evolution operations are comparable to FGTs that make up the refactorings. The literature suggests that this **operation-based merge** approach is more efficient and solves various problems that occur in other approaches (such as the **text-based merge** approach [32, 44, 45, and 48], in which software artifacts are considered as text or binary files). It is, however, out of the scope of this thesis to go into the details of these different merge approaches.

Should a refactoring tool allow a naive user to define a set of FGT-DAGs as constituting a new primitive refactoring (as explained in chapter 12), it is conceivable that these FGTs might contain mutual conflicts. The tool could, in principle, be designed to trace and report such conflicts. However, for the purposes of this thesis, it will be assumed that FGTs (and thus the associated FGT-DAGs) that make up a primitive refactoring do not conflict with one another. Nevertheless, the possibility arises that multiple primitive refactorings might be submitted to the tool for implementation. The tool ought to be able to detect and resolve conflicts that might exist between two (or more) such refactorings.

The various possibilities of conflicts that may occur between different FGTs have been pre-catalogued, as shown in Figure 8.3 and explained in more detail in Tables 8.1 and 8.2. This information is stored as facts in the Prolog database, examples of which are shown in Figure 8.4.

The arcs with arrows at both ends in Figure 8.3 represent bi-directional conflicts—i.e. conflicts that may occur between FGTs in both directions. Both directions mean that applying the two FGTs in either order will cause a conflict. For example, the following two FGTs obviously have a conflict in both directions:

FGT_1: addObject(*P, A,* _, _, _, _, *public,* _, *class*)
FGT_2: renameObject(*P, C,* _, _, _, *class, A*)

Applying FGT_1 first will prohibit applying FGT_2. This is because FGT_1 will add a new class with name *A* to the package *P* and after that, FGT_2 will try to rename another class *C* in package *P* to a new name *A*. Clearly, applying FGT_2 first will also prohibit subsequently applying FGT_1.

The arcs with a single arrow at one end in Figure 8.3 represent uni-directional conflicts—i.e. conflicts that occur between two FGTs in one direction only. Consider, for example:

FGT_1: renameObject (*P, C, _, _, _, class, A*)
FGT_2: deleteRelation (*isa, P, C, _, _, _, class, P, D, _, _, _, class, extends*)

Clearly, applying FGT_1 first will prohibit applying FGT_2 because after changing the name of the class *C* to *A* as per FGT_1, FGT_2 will not be able to find class *C*—i.e. its set of precondition conjuncts are no longer satisfied. On the other hand, applying FGT_2 first will not cause any conflict.

To resolve the conflict after being detected, a resolution procedure is defined for each type of conflict. Conflicts between the different FGTs (conflict-pairs) are categorized into three groups according to the approach used to resolve these conflicts:

1. **Ordering-conflicts** refer to conflicts that can be resolved by applying the two refactorings in a specific order.

2. **Cancelling-conflicts** refer to conflicts that can be resolved by cancelling (withdrawing) one of the two refactorings. The developer will be asked to choose one of the two refactorings to be cancelled.

3. **Removable-conflicts** refer to conflicts that can, in principle, be resolved by modifying one of the two FGTs that participate in the conflict. Suppose that FGT_x is from refactoring X and FGT_y is from refactoring Y; and suppose that the developer is asked to modify FGT_x. In doing so, the following should be taken into account:

   a. All FGTs that sequentially depend on FGT_x (i.e. descendants of FGT_x in the FGT-DAG of refactoring X) should also be modified to reflect the changes done on FGT_x. However, these changes should not introduce new sequential dependencies or redundancies in the FGT-DAGs of refactoring X.

   b. Changes that the developer makes on FGT_x should not produce new conflicts with ancestors of FGT_x in the relevant FGT-DAG.

   c. Changes that the developer makes on FGT_x should not produce conflicts with FGTs located in different FGT-DAGs of refactoring X—i.e. the FGTs constituting refactoring X should remain conflict-free.

d. The modified FGT_x should not have a conflict with any FGTs in refactoring Y that have already been checked to date.

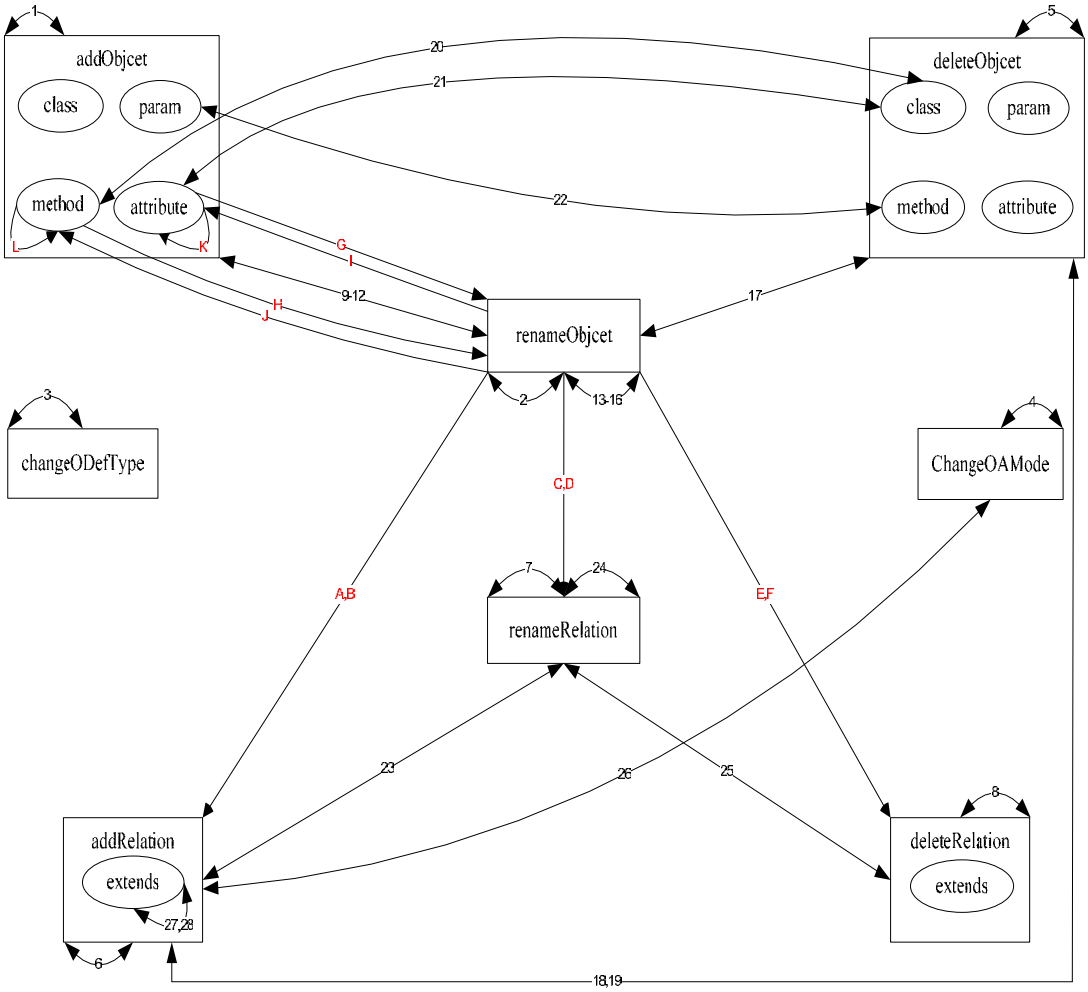


Figure 8.3: Possible conflicts between FGTs

```
.....
fgtConflict(addobject(P,C,M,X,_,_,_,_,_,OT), addObject(P,C,M,X,_,_,_,_,_,OT)).
fgtConflict(renameObject(P,C,_,_,_,attribute,X), renameObject(P,C,_,_,_,attribute,X)).
fgtConflict(addObject(P,C,X,_,_,_,_,_,_,attribute), deleteObject(P,C,_,_,_,class)).
fgtConflict(changeOAMode(P,C,M,PR,LT,OT,X,Y), addRelation(_,_,_,_,_,_,_,_,P,C,M,PR,LT,OT,_)).
fgtConflict(renameObject(P,C,M,PR,LT,OT,X), deleteRelation(_,P,C,M,PR,LT,OT,_,_,_,_,_,_,_)).
.....
```

Figure 8.4: A Selection of fgtConflict facts as implemented in Prolog

It is feasible to identify FGT pairs that constitute removable-conflicts. It is also possible to offer guidelines about how one of the FGTs in the pair may be changed without taking account of the overall context of the FGTs in the pair. However, there is no guarantee that a removable-conflict can indeed be resolved in *every specific context*. It is beyond the scope of this thesis to explore conditions in the surrounding context of a removable-conflict pair that will guarantee that the conflict can indeed be removed. Also left as a matter for future research, is the associated problem of algorithmically resolving removable-conflicts.

8.3 FGT's Conflicts-Pairs

In the following two subsections (8.4.1 and 8.4.2), a detailed description of each type of the bi-directional and uni-directional conflict is given. To clarify the discussion, a simplified UML class diagram of a *College* system is used. The system, shown in Figure 8.5, is in a package called *College* and has three classes: *Teacher*, *Student* and *PostGradStudent*.

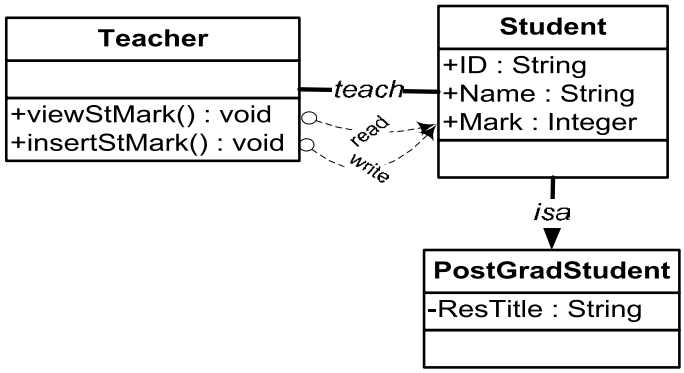


Figure 8.5: A simplified UML class diagram of a college system

8.3.1 Bi-Directional Conflict

A bi-directional conflict is a conflict that may occur between FGTs in both directions. Both directions mean that applying the two FGTs in either order will cause a conflict. In the following, a discussion of each bi-directional conflict catalogued in Table 8.1 is given. It will be seen that these conflicts are never classifiable as ordering-conflicts—only as cancelling or removable-conflicts.

Table 8.1: Bi-directional FGT Conflict-Pairs

No	FGT_x	FGT_y
1.	addobject(*P,C,M,X,_,_,_,_,OT*)	addObject(*P,C,M,X,_,_,_,_,OT*)
2.	renameObject(*P,C,M,PR,LT,OT,_*)	renameObject(*P,C,M,PR,LT,OT,_*)
3.	changeODefType(*P,C,M,PR,LT,ObjT,X,_*)	changeODefType(*P,C,M,PR,LT,ObjT,X,_*)
4.	changeOAMode(*P,C,M,PR,LT,ObjT,X,_*)	changeOAMode(*P,C,M,PR,LT,ObjT,X,_*)
5.	deleteObject(*P,C,M,PR,LT,ObjT*)	deleteObject(*P,C,M,PR,LT,ObjT*)
6.	addRelation(*RelL,P,C,M,PR,FLT,Ftype,P1,C1,M1 ,PR1,TLT,Totype,Ltype*)	addRelation(*RelL,P,C,M,PR,FLT,Ftype,P1,C1, M1,PR1,TLT,Totype,Ltype*)
7.	renameRelation(*X,P,C,M,PR,FLT,Ftype,P1, C1,M1,PR1,TLT,Totype,Ltype,_*)	renameRelation(*X,P,C,M,PR,FLT,Ftype,P1,C1, M1,PR1,TLT,Totype,Ltype,_*)
8.	deleteRelation(*RelL,P,C,M,PR,FLT,Ftype, P1,C1,M1,PR1,TLT,Totype,Ltype*)	deleteRelation(*RelL,P,C,M,PR,FLT,Ftype,P1,C1,M1,P R1,TLT,Totype,Ltype*)
9.	addObject(*P,C,M,X,_,_,_,LT,parameter*)	renameObject(*P,C,M,_, LT,parameter,X*)
10.	addObject(*P,C,X,_,_,_,_,_,attribute*)	renameObject(*P,C,_,_,_,attribute,X*)
11.	addObject(*P,C,X,_,_,_,_,LT,method*)	renameObject(*P,C,_,_, LT,method,X*)
12.	addObject(*P,X,_,_,_,_,_,_,class*)	renameObject(*P,_,_,_,_,class,X*)
13.	renameObject(*P,C,M,_, LT,parameter,X*)	renameObject(*P,C,M,_, LT,parameter,X*)
14.	renameObject(*P,C,_,_,_,attribute,X*)	renameObject(*P,C,_,_,_,attribute,X*)
15.	renameObject(*P,C,_,_, LT,method,X*)	renameObject(*P,C,_,_, LT,method,X*)
16.	renameObject(*P,_,_,_,_,class,X*)	renameObject(*P,_,_,_,_,class,X*)
17.	renameObject(*P,C,M,PR,LT,OT,X*)	deleteObject(*P,C,M,PR,LT,OT*)
18.	addRelation(*_,P,C,M,PR,LT,Ftype,_,_,_,_,_,_,_*)	deleteObject(*P,C,M,PR,LT,Ftype*)
19.	addRelation(*_,_,_,_,_,_,_,P,C,M,PR,TL,Ttype,_*)	deleteObject(*P,C,M,PR,TL,Ttype*)
20.	addObject(*P,C,X,_,_,_,_,LT,method*)	deleteObject(*P,C,_,_,_,class*)
21.	addObject(*P,C,X,_,_,_,_,_,attribute*)	deleteObject(*P,C,_,_,_,class*)
22.	addObject(*P,C,M,X,_,_,_,LT,parameter*)	deleteObject(*P,C,M,_,LT,method*)
23.	renameRelation(*X,P,C,_,_,_,class,P1,C1,_,_,_, class,association,Y*)	addRelation(*Y,P,C,_,_,_,class,P1,C1,_,_,_,class, association*)
24.	renameRelation(*_,P,C,_,_,_,class,P1,C1,_,_,_, class,association,X*)	renameRelation(*_,P,C,_,_,_,class,P1,C1,_,_,_,class, association,X*)
25.	renameRelation(*X,P,C,M,PR,FLT,Ftype,P1, C1,M1,PR1,TLT,Totype,LType,Y*)	deleteRelation(*X,P,C,M,PR,FLT,Ftype,P1,C1,M1,PR1, TLT,Totype,LType*)
26.	changeOAMode(*P,C,M,PR,LT,OT,X,Y*)	addRelation(*_,_,_,_,_,_,_,P,C,M,PR,LT,OT,_*)
27.	addRelation(*_,P,C,_,_,_,class,P1,C1,_,_,_,class, extends*)	addRelation(*_,P2,C2,_,_,_,class,P1,C1,_,_,_,class, extends*)
28.	addRelation(*_,P,C,_,_,_,class,P1,C1,_,_,_,class, extends*)	addRelation(*_,P1,C1,_,_,_,class,P,C,_,_,_,class, extends*)

- Conflicts 1-8 between FGT_x and FGT_y in the table occur in the case that FGT_x and FGT_y are the same, applying the first one on the system will prohibit applying the second one. For example, in the *College* system, an attempt to apply the following two FGTs in either order will cause a conflict:

  FGT_x: addobject(*College, Student, Age, _, _, _, _, _, attribute*)
  FGT_y: addobject(*College, Student, Age, _, _, _, _, _, attribute*)

It is clear that the two FGTs try to add the same attribute *Age* to the class *Student*—something that cannot happen more than once.

In general, it can easily be seen that attempting to apply any FGT more than once in succession will always cause a conflict. Indeed, it is in the very nature of an FGT to *transform* the state of a system which satisfies its precondition to a *different* state in which the precondition is no longer satisfied.

These conflicts are classifiable as cancelling-conflicts. One of the two refactorings should be cancelled to resolve the conflict.

- Conflicts 9-12 between FGT_x and FGT_y occur when FGT_x tries to add an object X to the system and FGT_y tries to change the name of another object to X, or *vice versa*. This means that the system will have two objects with the same name X which is prohibited.

  The type of these conflicts is removable-conflict. If the context permits it, one of the two FGTs can be modified in a way that resolves the conflicts between the two FGTs. For example, in the *College* system trying to apply the following two FGTs successively in either order will cause a conflict:

  FGT_x: addobject(*College, Teacher, listSTMarks, _, _, _, _, [], method*)
  FGT_y: renameObject(*College, Teacher, viewSTMark, _, [], method, listSTMarks*)

  The conflict could perhaps be resolved by choosing to rename the *viewSTMark* method to, say, *displaySTMark*, should the context permit this.

- Conflicts 13-16 between FGT_x and FGT_y occur when FGT_x tries to change the name of an object *X* to a new name *Y* and FGT_y tries to change the name of another object *Z,* defined in the same scope as object *X,* to the same new name *Y*. This means that the system will have two objects with the same name *Y* defined within the same scope which is prohibited. This would happen if and attempt was made to apply the following FGTs:

  FGT_x: renameObject(*College, Stuednt, ID, _, _, attribute, StPinf*)
  FGT_y: renameObject(*College, Stuednt, Name, _, _, attribute, StPinf*)

  The type of these conflicts is removable-conflict. If the context permits it, one of the two FGTs can be modified in a way that resolves the conflicts between the two FGTs.

- Conflict 17 between FGT_x and FGT_y occurs when FGT_x tries to change the name of an object from Y to X and FGT_y tries to delete that object using the old name Y or vice versa.

For example, in the *College* system, an attempt to apply the following two FGTs in either order will cause a conflict:

FGT_x: renameObject(*College, Student, Mark, _, _, attribute, Grade*)
FGT_y: deleteObject(*College, Student, Mark, _, _, attribute*)

Applying FGT_x first will change the name of the attribute *Mark* to a new name *Grade*. Then at the time of applying FGT_y attribute *Mark* will not be found. Applying FGT_y first will delete the attribute *Mark* from the class *Student*, then at the time of applying FGT_x attribute *Mark* will not be found.

This conflict is classifiable as cancelling-conflict. One of the two refactorings should be cancelled to resolve the conflict.

- Conflict 18 and 19 between FGT_x and FGT_y occur when FGT_x tries to add a *relation* between two different objects in the system and FGT_y tries to delete one of the two objects which participates in that relation or vice versa. For example, in the *College* system, an attempt to apply the following two FGTs in either order will cause a conflict:

  FGT_x: addRelation(*_, College, Teacher, viewSTMark, _, [], method, College, Student, Name, _, _, attribute, read*)
  FGT_y: deleteObject(*College, Student, Name , _, _, attribute*)

  Applying FGT_x first will add the *read* relation between the method *Teacher.viewSTMark* and the attribute *Student.Name,* indicating that the method *viewSTMark* has *read* access to attribute *Name*. This means that the attribute *Student.Name* is now referenced from another object in the system which means that it could not be deleted by FGT_y. Conversely, applying FGT_y first will delete the attribute *Student.Name* that will prohibit applying FGT_x because at that time one of the participating objects in the *relation* will not be defined.

  These conflicts are classifiable as cancelling-conflicts. One of the two refactorings should be cancelled to resolve the conflict.

- Conflict 20-22 between FGT_x and FGT_y occur when FGT_x tries to add an object in a container and FGT_y tries to delete that container or vice versa. An example would be if FGT_x tries to add a member (attribute or method) in a class while FGT_y tries to delete that class. Adding a member in a class will prohibit deleting that class. The same when FGT_x tries to define a new parameter in a method *m* and FGT_y tries to delete that method using the old signature of the method (the parameter list before adding the new parameter).

These conflicts are classifiable as cancelling-conflicts. One of the two refactorings should be cancelled to resolve the conflict.

- Conflict 23 is the same as conflicts 9-12 but with *association* relations. It occurs when one of the FGT tries to add a new *association* relation with label *X* between two classes while another FGT tries to change the label of another *association* relation that exists between the same two classes to a new label *X*. This means that the two classes will have two *association* relations with the same label which is prohibited. For example, in the *College* system, an attempt to apply the following two FGTs in either order will cause a conflict:

  FGT_x: renameRelation(*teach, College, Teacher, _, _, _, class, College, Student, _, _, _, class, association, supervise*)

  FGT_y: addRelation(*supervise, College, Teacher, _, _, _, class, College, Student, _, _, _, class, association*)

  Note that the reason for just considering *association* relations and excluding the other types of relations in this type of conflict is the following:

  c. For *read*, *write*, *call* and *type* relations: There is no label for these relations as explained in section 3.3 and also it is prohibited for two objects to have more than one *relation* of the same type (*read, write, call* or *type*).

  d. For *extends* relation it is also prohibited for two classes to have more than one *extends* relation between them at the same time.

  The type of this conflict is removable-conflict. If the context permits it, one of the two FGTs can be modified in a way that resolves the conflicts between the two FGTs.

- Conflict 24 occurs when FGT_x and FGT_y try to change the label of two different *association* relations that exists between two specific objects to the same label. This means that the two associations will have the same label which is prohibited. The other types of relations are excluded from this conflict for the same reasons explained above.

  The type of this conflict is removable-conflict. If the context permits it, one of the two FGTs can be modified in a way that resolves the conflicts between the two FGTs.

- Conflict 25 between FGT_x and FGT_y occurs when FGT_x tries to change the label of a relation (*association or extends*) that exists between two classes from *X* to *Y* and FGT_y tries to delete that relation using the old label *X* or vice versa. For example, in the *College* system, an attempt to apply the following two FGTs in either order will cause a conflict:

FGT_x: renameRelation(*teach, College, Teacher, _, _, _, class, College, Student, _, _, _, class, association, supervise*)

FGT_y: deleteRelation(*teach, College,Teacher,_,_,_,class,College,Student,_,_,_, class, association*)

This conflict is classifiable as cancelling-conflict. One of the two refactorings should be cancelled to resolve the conflict.

- Conflict 26 between FGT_x and FGT_y occurs when FGT_x tries to change the access mode of specific object *A* from a less restricted mode to a more restricted one and FGT_y tries to add a relation where the destination of the relation is the object *A* and the relation requires that the access mode of the object *A* should be the less restricted one. In the *College* system, an attempt to apply the following two FGTs in either order will cause a conflict:

  FGT_x: changeOAMode(*College, Student, Name , _, _, attribute, public, private*)

  FGT_y: addRelation(*_, College, Teacher, viewStMark, _, [], method, College, Student, Name, _, _, attribute, read*)

  Applying FGT_x first will change the access mode of the attribute *Student.Name* from *public* to *private* which will prohibit applying FGT_y. Alternatively, applying FGT_y first will add a *read* relation between the method *Teacher.viewStMark* and the attribute *Student.Name* this will prohibit applying FGT_x.

  This conflict is classifiable as cancelling-conflict. One of the two refactorings should be cancelled to resolve the conflict.

- Conflict 27 between FGT_x and FGT_y occurs when the two FGTs try to add an *extends* relation between two classes where the subclass in the two FGTs is the same class *A*, this means that class *A* will have multiple superclass (multiple inheritance) which is not allowed.

  This conflict is classifiable as cancelling-conflict. One of the two refactorings should be cancelled to resolve the conflict.

- Conflict 28 between FGT_x and FGT_y occurs when FGT_x tries to add an *extends* relation between class *A* and *B* where *A* is the source (superclass) of the relation and *B* is the destination (subclass) of the relation. At the same time, FGT_y tries to add an *extends* relation between class *A* and *B* where *B* is the source (superclass) of the relation and *A* is the destination (subclass) of the relation which is prohibited.

This conflict is classifiable as cancelling-conflict. One of the two refactorings should be cancelled to resolve the conflict.

8.3.2 Uni-Directional Conflict

A uni-directional conflict is a conflict that may occur between two FGTs, but only if they are applied to the system in a specific order. In the following a discussion of each type of uni-directional conflict as catalogued in Table 8.2 is given. Illustrative use of the simplified UML class diagram of a *College* system is continued. Note that all the uni-directional conflicts are ordering-conflicts. A tool should determine the specific order that should be followed to resolve the conflict between the two refactorings. In reference to the conflict-pairs in Table 8.2, if FGT_y is applied first, then the conflict will be resolved.

Table 8.2: Uni-directional FGT conflict-pairs

No	FGT_x	FGT_y
A.	renameObject(*P,C,M,PR,LT,OT,*X)	addRelation(_,*P,C,M,PR,LT,OT,*_,_,_,_,_,_,_)
B.	renameObject(*P,C,M,PR,LT,OT,X*)	addRelation(_,_,_,_,_,_,_,*P,C,M,PR,LT,OT,*_)
C.	renameObject(*P,C,M,PR,LT,OT,X*)	renameRelation(_,*P,C,M,PR,LT,OT,*_,_,_,_,_,_,_,_)
D.	renameObject(*P,C,M,PR,LT,OT,X*)	renameRelation(_,_,_,_,_,_,_,*P,C,M,PR,LT,OT,*_,_)
E.	renameObject(*P,C,M,PR,LT,OT,X*)	deleteRelation(_,*P,C,M,PR,LT,OT,*_,_,_,_,_,_,_)
F.	renameObject(*P,C,M,PR,LT,OT,X*)	deleteRelation(_,_,_,_,_,_,_,*P,C,M,PR,LT,OT,*_)
G. *	addObject(*P1,C1,X,*_,_,_,_,_,*attribute*)	renameObject(*P2,C2,*_,_,_,*attribute,X*)
H.*	addObject(*P1,C1,X,*_,_,_,_,_,*method*)	renameObject(*P2,C2,*_,_,_, *method,X*)
I.*	renameObject(*P1,C1,*_,_,_,*attribute,X*)	addObject(*P2,C2,X,*_,_,_,_,_,*attribute*)
J. *	renameObject(*P1,C1,*_,_,_,*method,X*)	addObject(*P2,C2,X,*_,_,_,_,_, *method*)
K.*	addObject(*P1,C1,X,*_,_,_,_,_,*attribute*)	addObject(*P2,C2,X,*_,_,_,_,_,*attribute*)
L.*	addObject(*P1,C1,X,*_,_,_,_,_,*method*)	addObject(*P2,C2,X,*_,_,_,_,_, *method*)

* Note: Assume *P1.C1* is one of the ancestor's of *P2.C2*.

- Conflicts A and B between FGT_x and FGT_y occur when FGT_x tries to change the name of object from *X* to *Y* and FGT_y tries to add a *relation* between two objects where the object used in FGT_x is the source or the destination object in the relation, and the relation continues to use the old name of the object *X*. Applying FGT_x first then FGT_y will cause a conflict but applying the two FGTs in a reverse order will not cause any conflicts. For example, in the following two FGTs in the *College* System will result in a conflict:

  FGT_x: renameObject(*College, Student , Name, _, _, attribute, StName*)

  FGT_y: addRelation(*_, College, Teacher, viewStMark, _, [], method, College, Student, Name, _, _, attribute, read*)

Applying FGT_x first will change the name of the attribute in the class *Student* from *Name* to *StName* which will prohibit applying FGT_y that uses the old name which is not defined at this point. However, applying FGT_y first then FGT_x will not cause any conflicts because at the time of applying FGT_y the two objects used in the relation are defined. The ID of the two objects will be used to store the *relation* in the facts' database, as:

read(*RelID, _, viewStMarkID, NameID*)

Subsequently, FGT_x may change the name of the attribute *Student.Name* to *Student.StName* but this will not cause a conflict at the level of the stored ID information.

- Conflicts C-F are similar to the conflicts in the previous point but instead of adding a new relation they change the label or delete an existing relation. In both of the cases, they use the old name of the object.

- Conflicts G and H between FGT_x and FGT_y occur when FGT_x tries to add a new member with name *X* to the class *C1* and FGT_y tries to change the name of another member in the class *C2* to the name *X* where class *C1* is an ancestor of class *C2*. Applying FGT_x first will prohibit applying FGT_y because FGT_y in this case will try to redefine an inherited member which is not allowed. For example, a conflict will occur in the following two FGTs in the *College* System:

  FGT_x: addObject(*College, Student, Major, _, _, _, _, _, attribute*)
  FGT_y: renameObject(*College, PostGradStudent, ResTitle, _, _, attribute, Major*)

  Applying FGT_x first will add the attribute *Major* to the *Student* class which then prohibits applying FGT_y because the attribute *Major* is inherited from a superclass and it is not allowed to redefine it. Applying FGT_y first will not cause any conflict because after changing the name in the subclass to *Major* there is no problem to add an attribute with the same name in a superclass.

- Conflicts I-L are almost the same as conflicts in the previous point. In these conflicts, one of the two FGTs tries to define a new member or rename an existing member in the subclass X where the new name is the same as one that is already defined by the second FGT in one of the class X's ancestors, which means that the inherited member from the superclass is being redefined and this is prohibited.

8.4 Conflict Algorithm

Because each refactoring is represented as a collection of FGTs ordered in FGT-DAGs it is possible to identify, at an FGT-level, where conflicts might occur in two refactorings. At a later stage, such conflicts will need to be resolved.

In this section, a conflict detection and resolving algorithm (**detectResolveConflict**) is defined. The algorithm is based on detecting and resolving conflicts of FGTs that constitute these refactorings. To do so, the algorithm uses the information given in Table 8.1 and 8.2, which is stored in the database of the tool as **fgtConflicts** facts.

Algorithm 8.1 gives the pseudo-code of the **detectResolveConflict** algorithm. The algorithm takes as input two refactorings X and Y. The algorithm initially assumes that X and Y can be applied to the system in either order. It ends with one of the following verdicts:

1. The (possibly user-modified) refactorings are conflict-free, and can be applied in any order.
2. The (possibly user-modified) refactorings are conflict-free provided they are applied in a specified order.
3. The refactorings are in conflict and the user has withdrawn one of them.

The algorithm works in a nested loop fashion. For each FGT-DAG$_i$ of refactoring X, the algorithm starting from the root, taking each FGT$_{ii}$ in FGT-DAG$_i$ and checking if there is a conflict between it and all the FGT$_{jj}$ in every FGT-DAG$_j$ of refactoring Y. The traversal of FGT-DAGs is also in a top-down fashion, starting at the root. Every pair (FGT$_{jj,}$ FGT$_{ii}$) or (FGT$_{ii,}$ FGT$_{jj}$) is checked for a match with the **fgtConflict** facts. If a match is found this means that there is a conflict between FGT$_{ii}$ and FGT$_{jj}$.

It is emphasized that the algorithm traverses the FGT-DAGs in the two refactorings under consideration from top to bottom. This allows one to detect the first occurrence of conflict in the two FGT-DAGs, perhaps to resolve this conflict and, if required, to modify the FGTs in the FGT-DAG$_i$ that are sequentially dependent on a modified FGT. This means that when the algorithm reaches FGT$_{ii}$ of FGT-DAG$_i$ of refactoring X, all the FGTs before FGT$_{ii}$ (ancestors of FGT$_{ii}$) in FGT-DAG$_{ii}$ are conflict-free (there are no conflicts between any of them with any other FGTs in refactoring Y). This is illustrated in Figure 8.6.

Algorithm 8.1 (Conflict detection & resolving algorithm)

detectResolveConflict(Ref X, Ref Y)

Input: Ref X: a conflict-free redundancy-free set of FGT-DAGs of refactoring X
Ref Y: a conflict-free redundancy-free set of FGT-DAGs of refactoring Y

Output: Detect & Resolve conflicts between refactorings X & Y

For each FGT_{ii} in $FGT\text{-}DAG_i$ (*starting from the root*) in Ref X **do** {
  For each FGT_{jj} in $FGT\text{-}DAG_j$ (*starting from the root*) in Ref Y **do** {
    If there is a match between the pair (FGT_{jj}, FGT_{ii}) and an **fgtConflict** fact **then** {
      switch (conflict-pair(FGT_{ii}, FGT_{jj})) {
        ordering-conflict: { Determine the correct order of the two refactorings that resolves the conflict between FGT_{ii} and FGT_{jj}. If this order is opposite to an order determined in a previous iteration of the algorithm, behave as for a cancelling-conflict }
        removable-conflict: { Ask the developer to modify FGT_{ii} to resolve the conflict, accounting for all matters mentioned above. This includes modifying, if necessary, FGTs in the sub-DAG rooted in FGT_{ii}. If such modification is not allowed by the context, then behave as for the cancelling-conflict }
        cancelling-conflict { Ask the developer to choose one of the refactorings X or Y. Delete the chosen refactorings from the system. End the detectResolveConflict procedure }
      } //end **switch**
    } //end **If**
  } //end **for**
} //end **for**

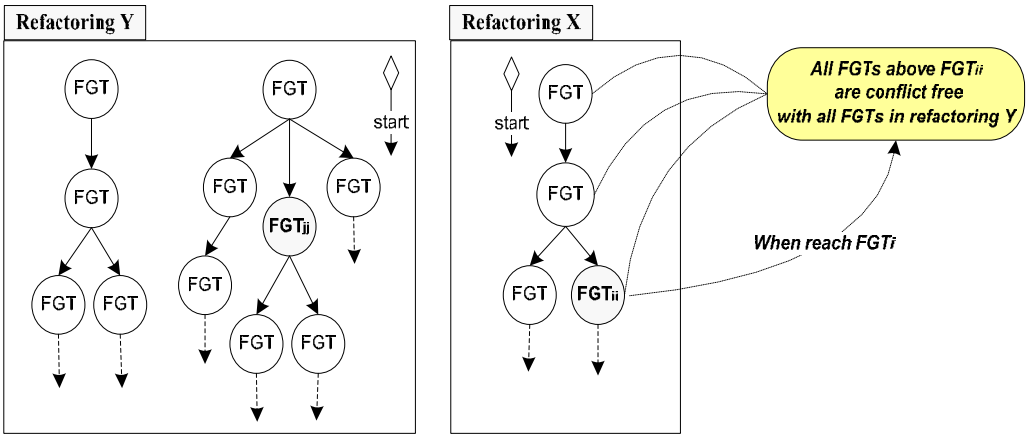


Figure 8.6: Conflict detection & resolving algorithm

To resolve the conflict when it is detected, the algorithm determines the type of the conflict between the pair FGT_{ii} and FGT_{jj}. Three different types of conflicts are identified:

1. If the conflict is an ordering-conflict, then the algorithm relies on the order of FGT_{ii} and FGT_{jj} in the matched **fgtConflict** fact to determine the correct order of the two refactorings. For example, in reference to the *College* system, suppose that FGT_{ii} and FGT_{jj} in Figure 8.6 are as the following:

FGT_{ii}: addRelation(_, *College, Teacher, viewStMark, _, [], method, College, Student, Name, _, _, attribute, read*)

FGT_{jj}: renameObject(*College, Student, Name, _, _, attribute, StName*)

In this example, **detectResolveConflict** algorithm will find a match between the pair (FGT_{ii}, FGT_{jj}) and the fact

fgtConflict(renameObject(*P,C,M,PR,LT,OT,X*), addRelation(_,_,_,_,_,_,_,*P,C,M,PR,LT,OT,*_))

This conflict is explained in row B of Table 8.2. According to the information stored in the refactoring tool's database, the conflict is an ordering-conflict. To resolve this conflict, therefore, the refactoring that contains the FGT which matches with the second argument of the **fgtConflict** fact should be applied first. In the example, FGT_{ii} (from refactoring X) matches with the second argument of the fact. This means that refactoring X should be applied first, and then refactoring Y. Using this scenario, when apply refactoring X, the *read* relation will be added between the two objects using the old name of the attribute *College.Student.Name.* Thereafter, by applying refactoring Y, the name of the attribute *College.Student.Name* will be changed to *College.Student.StName.* The conflict is resolved.

Note, however, that it is possible that at a later stage, another ordering-conflict is detected. If this conflict can only be resolved by an opposite ordering to the one already determined then there is a deadlock between the two refactorings. The only way to resolve the deadlock is to withdraw one of the refactorings, as in the case of a cancelling-conflict.

2. If the conflict is a removable-conflict, then the algorithm asks the developer to modify FGT_{ii} in a way that will resolve the conflicts between the two FGTs. The algorithm then will traverse FGT-DAG_i from FGT_{ii} down and modify all FGTs that sequentially depends on FGT_{ii} to reflect the modification of FGT_{ii}. For example, related to the *College* system, suppose that FGT_{ii} and FGT_{jj} in Figure 8.6 are as the following:

FGT_{ii}: addObject(*College, PostGradStudent, ResField, _, _, type(basic, string, 0), public, _, attribute*)

FGT_{jj}: renameObject(*College, PostGradStudent, ResTitle, _, _, attribute, ResField*)

In this example, **detectResolveConflict** algorithm will find a match between the pair ($FGT_{ii,}$ FGT_{jj}) and the fact

fgtConflict(addObject(*P,C,X,_,_,_,_,_,attribute*), renameObject(*P,C,_,_,_,attribute,X*) **)**

this means that there is a conflict between the two FGTs. This is because as a result of the two FGTs, the class *PostGradStudent* will have two attributes with the same name *ResField* ,which is a conflict. This conflict is explained in row 10 of Table 8.1. According to the information stored in the refactoring tool's database, the conflict is a removable-conflict. To resolve this conflict, the algorithm will ask the user to choose another name to be used for the attribute in FGT_{ii} instead of *ResField* (*ResSubject* for example). Then the tool will modify FGT_{ii} to be:

addObject(*College, PostGradStudent,* ***ResSubject****, _, _, type(basic, string, 0), public, _, attribute*)

instead of

addObject(*College, PostGradStudent,* ***ResField****, _, _, type(basic, string, 0), public, _, attribute*)

After that, the algorithm will traverse FGT-DAG_i from the node FGT_{ii} down until it reaches the leaves, changing each occurrence of the attribute *College.PostGradStudent.ResTitle* to *College.PostGradStudent.ResSubject*. Note, however, that it is possible that the modification is not allowed by the context. Then the only way to resolve the deadlock is to withdraw one of the refactorings, as in the case of a cancelling -conflict.

3. If the conflict is a cancelling-conflict, then the algorithm will ask the developer to choose one of the refactorings X or Y to be cancelled. The algorithm will delete the chosen refactoring from the system and terminate the execution of the algorithm. For example, suppose that FGT_{ii} and FGT_{jj} in Figure 8.6 are as the following:

FGT_{ii}: addRelation(_,*College, Teacher, viewStMark, _, [], method, College, Student, Name, _, _, attribute, read*)

FGT_{jj}: DeleteObject(*College, Student, Name, _, _, attribute*)

In this example, FGT_{ii} tries to add a *read* relation from the method *College.Teacher.viewStMark* to the attribute *College.Student.Name*. At the same time FGT_{jj} tries to delete the attribute *College.Student.Name* which leads to a conflict. This conflict is explained in row 19 of Table 8.1. According to the information stored in the database of the refactoring tool, the conflict is one of the cancelling-conflicts. To resolve this conflict, the algorithm will ask the user to choose one of the two refactorings (X or Y) to be cancelled.

8.5 LAN Motivated Example

Consider the motivated LAN example. Assume a multi-user system, such that one user wants to apply the refactoring **pullUpMethod** to pull up the method *accept* from the subclasses *FileServer*, *PrintServer* to their superclass *Server*, and another user wants to move the *accept* method from the *FileServer* class to the *Packet* class. The latter user may be motivated, for example, by the fact that the *accept* method can directly access the variable *receiver* in the class *Packet*. Clearly that there is a conflict between the two refactorings **moveMethod** and **pullUpMethod** because there is no possibility to move the method *accept* from the *FileServer* to the class *Packet* and at the same time pull it up from the *FileServer* to the superclass *Server*.

In order to discover this fact algorithmically by the FGT-based tool, the **detectResolveConflict** procedure will be called and the two refactorings, shown in Figure 8.7, will be sent as parameters to the procedure. After executing the procedure, a match is found between the following pair of FGTs (FGT_{ii}, FGT_{jj}) and one of the **fgtConflict** facts:

FGT_{ii}: deleteObject(*Lan, FileServer, accept, _, [Packet], method*)

FGT_{jj}: deleteObject(*Lan, FileServer, accept, _, [Packet], method*)

where FGT_{ii} is from refactoring **pullUpMethod** and FGT_{jj} is from refactoring **moveMethod**. The **fgtConflict** fact that the match occurs with is:

fgtConflict(deleteObject(*P,C,M,PR,LT,ObjT*), deleteObject(*P,C, M,PR,LT,ObjT*))

This conflict is explained in row 5 of Table 8.1. According to the information stored in the refactoring tool's database, the conflict is one of the cancelling-conflicts. To resolve this conflict, the algorithm will require that one of the two refactorings (**moveMethod** or **pullUpMethod**) be cancelled.

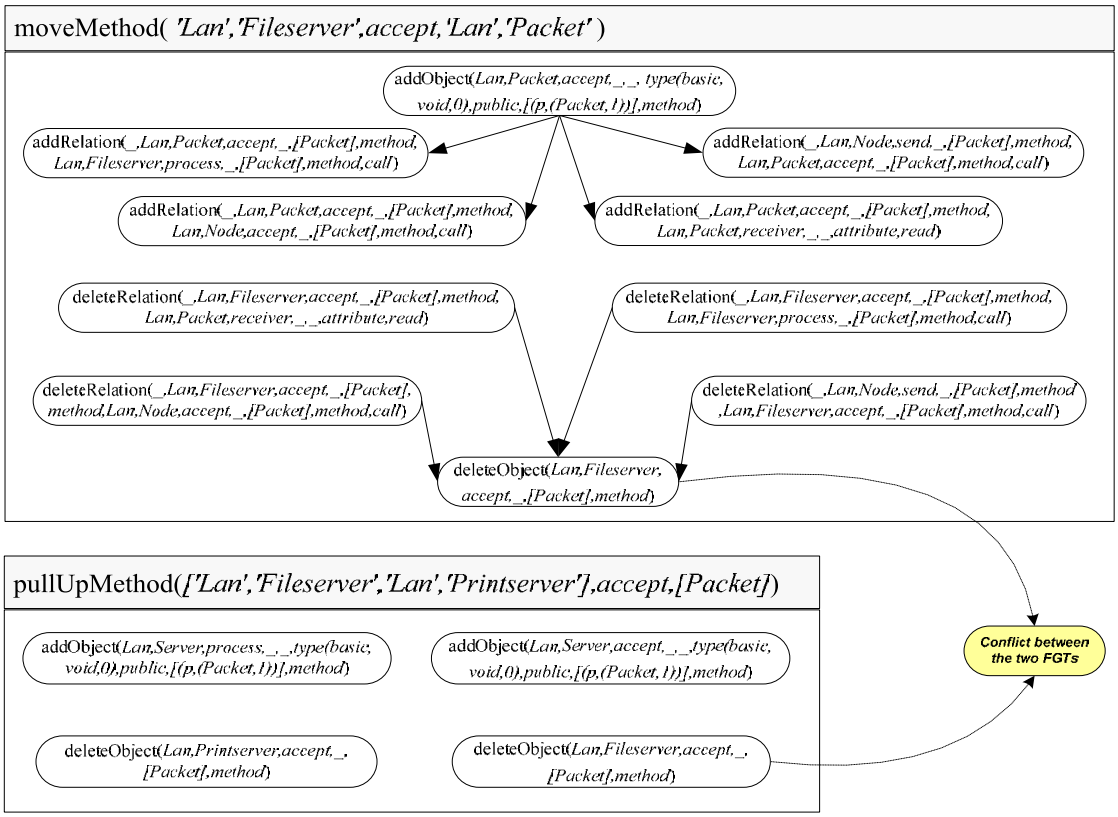


Figure 8.7: Conflicts between refactorings moveMethod & pullUpMethod

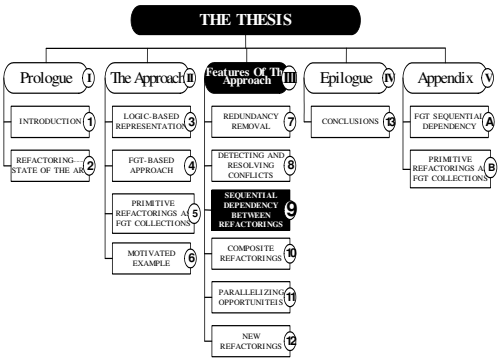
8.6 Reflections on Conflicts

The conflicts discussed in this chapter should not be confused with sequential dependencies—neither at the refactoring-level nor at the FGT-level. These conflicts deal with what may not happen before applying a refactoring (or FGT). For example, Table 8.2 (entries A-E) specify that an object may not be renamed before changing (i.e. adding, deleting, or renaming) a relation associated with the object. This does not mean that a relation involving an object has to be changed before renaming the object. What has to happen before apply some refactoring or FGT is, broadly speaking, a sequential dependency issue, and this will be discussed in the next chapter.

Chapter 9

SEQUENTIAL DEPENDENCY BETWEEN REFACTORINGS

9.1 Introduction



As in the previous chapter, the concern is with two arbitrary refactorings, R_i and R_j. However, in this case, it is assumed that R_i and R_j are conflict-free. The first question to consider is the following: Is it possible to apply the two refactorings in any order on any system that satisfies the preconditions of both? The answer is, of course, that it is possible, since the refactorings are assumed to be conflict-free. A second question then, is whether it could be possible to apply R_j after applying R_i in a system that initially satisfied R_i's precondition, but not R_j's. There is one of two possible answers:

1. Yes, it would be possible, provided that the initial state of the system is such that it satisfies those precondition conjuncts of R_j that are not realized as a result of applying R_i first.

2. No, it is not possible, because there is some inherent contradiction between the pre- and postconditions of R_i and R_j—even though they are conflict-free. In this case, the deadlock problem arises.

It should be emphasized the assumption of conflict freedom between R_i and R_j is, initially, strict—i.e. it is assumed that there is no ordering-conflict that can be resolved if one refactoring is applied after the other. In considering the deadlock problem in section 9.5, however, this restriction will be lifted.

Refactoring R_j is sequentially dependent on refactoring R_i ($R_i \rightarrow R_j$) if some or all of the precondition conjuncts of refactoring R_j are satisfied by applying refactoring R_i first. Thus, the definition of refactoring sequential dependency is similar to the definition of FGT sequential dependency, namely:

Refactoring R_j sequentially depends on refactoring R_i if and only if the postcondition conjuncts of refactoring R_i satisfies one or more precondition conjuncts of refactoring R_j.

As an example, suppose that a system has a package with name *P* and that a user intends to apply two refactorings R_i and R_j. Assume—as shown in Figure 9.1—that the two refactorings involve the following respective transformations on the system: Refactoring R_i adds a new class *A* in the package *P*; and refactoring R_j adds a new attribute *Attn* in the class *P.A* which is added in refactoring R_j. Clearly, refactoring R_j sequentially depends on refactoring R_i.

A batch of refactorings may be produced in a multi-developer environment in which groups of developers work on the same system. Potentially, such a batch of refactorings may be large with many sequential dependencies between the different refactorings. It would therefore be useful to have an automated way of discovering such sequential dependencies between refactorings.

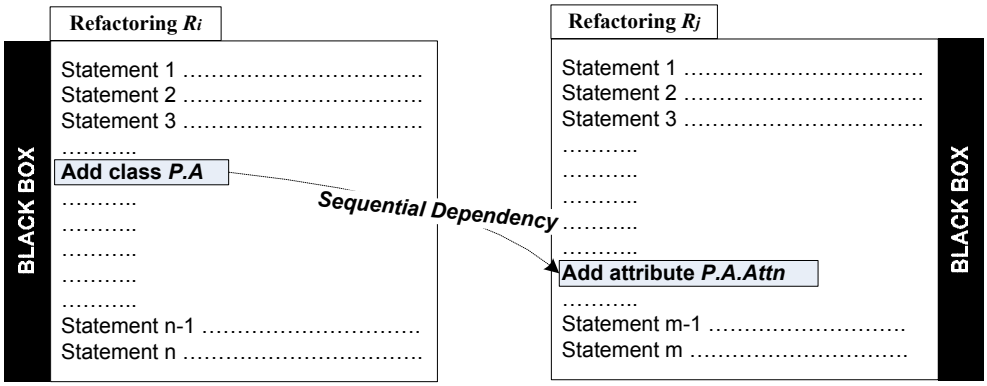


Figure 9.1: Sequential dependency between refactorings R_i & R_j

9.2 Sequential Dependency in Previous Approaches

A straightforward approach to apply a set of refactorings in a batch to a system is simply to traverse the batch to find a candidate whose precondition conjuncts are satisfied by the system and then to apply it to the system. Then search the batch again, looking for a new candidate and so on until the list is finished or none of the remaining refactorings can be applied. Each time the new refactoring's precondition conjuncts will be checked against the system under consideration, which means that, potentially, the tool has to make many references to the system. Note that in practice, the description of the system may be very large, and references to such a large system therefore runs the risk of becoming costly.

Such an approach suffers from the classical disadvantages of "greedy" algorithms: non-optimal behaviour. For example, in the example presented in Figure 9.1, if the tool by chance chooses refactoring R_j first, then its precondition conjuncts will not be satisfied. Refactoring R_i will then be chosen, its precondition conjuncts checked, and processing then proceeds (This means that R_i will be applied to the system). Subsequently, the tool will go back to refactoring R_j to check its precondition conjuncts again, which implies duplication of work and effort in referencing the underlying system.

In order to solve the above problem, various authors have proposed alternative approaches to find sequential dependencies between refactorings by trying to find such relations without a need to check the underlying system under consideration, thus reducing the time needed to reference the underlying system. In [39, 52, 55, and 70], an approach is proposed that infers sequential dependency relations between the different refactorings by comparing their pre- and postcondition conjuncts without a need to reference the state of the system under consideration.

In the example presented in Figure 9.1, the existence of class *A* in package *P* is one of the conjuncts in the precondition of refactoring R_j. Also one of the conjuncts of the postcondition of refactoring R_i is precisely the existence of class *A* in package *P*. Because the postcondition of refactoring R_i satisfies a conjunct of the precondition conjuncts of refactoring R_j, a tool can infer that R_j sequentially depends on R_i.

While depending on pre- and postcondition conjuncts of refactorings does indeed establish whether or not there is a sequential dependency between two refactorings, the following should be noted:

1. Sometimes it is impossible to infer the sequential dependency between the two refactorings by considering the pre- and postcondition conjuncts in isolation. Figure 9.2 shows an example of two refactorings *Ri* and R_j where part of the two refactorings does the following respective transformations on the system: Refactoring R_i adds a new class *A* in the package *P*; and refactoring R_j changes the name of the class *P.A* to another name *P.C* as indicated in the example.

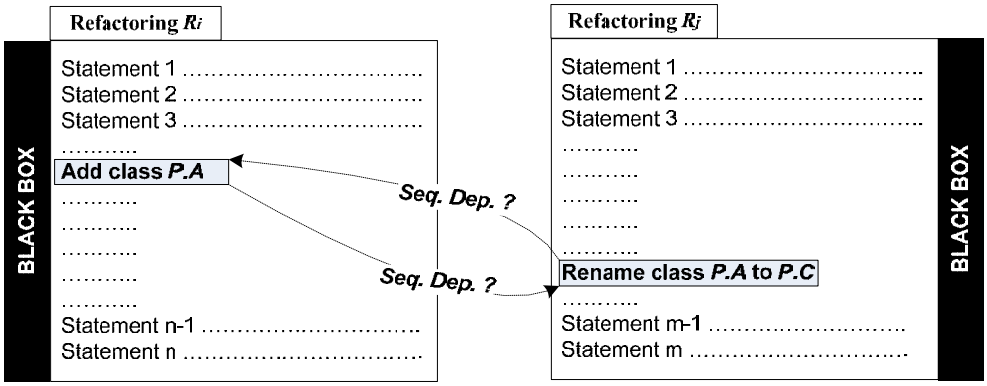


Figure 9.2: Ambiguous sequential dependency

Considering just these transformations of the two refactorings the following pre- and postcondition of the two refactorings can be distinguished:

Precondition of R_i*:* class *A* not defined in package *P*

Postcondition of R_i*:* class *A* is defined in package *P*

Precondition of R_j*:* class *A* is defined in package *P*, class *C* is not defined in package *P*

Postcondition of R_j*:* class *A* is not defined in package *P*, class *C* is defined in package *P*

From the information that can be inferred from the previous pre- and postcondition conjuncts of the two refactorings, two scenarios can be distinguished:

- It is clear that part of the postcondition conjuncts of refactoring R_i is included in the precondition conjuncts of refactoring R_j. From this, the tool can infer that R_j is sequentially dependent on R_i ($R_i \rightarrow R_j$).

- It is also clear that part of the postcondition conjuncts of refactoring R_j is included in the precondition conjuncts of refactoring R_i. From this, the tool can infer that R_i is sequentially dependent on R_j ($R_j \rightarrow R_i$).

Therefore, there is an ambiguity about the sequential dependency relation between the two refactorings if they are viewed in isolation of the underlying system. As far as can be established, authors of approaches such as in [39, 52, 55, and 70] do not address this problem. The ambiguity may be resolved by checking the underlying system to discover the real state of the system. This makes it possible to choose the correct sequential dependency relation that is imposed by the system under consideration. In the previous example, if the class *P.A* is already defined in the system then scenario **b** is the correct scenario where ($R_j \rightarrow R_i$). On the other

hand, if class *P.A* is not defined in the system then scenario **a** is the correct scenario where ($R_i \rightarrow R_j$).

2. Since refactoring tools to date typically implement each refactoring as black-boxed sequence of coding statements, it is not possible to establish at what specific points in the respective code blocks the two refactorings become sequentially dependent. This leads to the following shortcomings:

a. It is not possible to exploit any possibilities for implementing the refactorings in parallel.

b. It is not possible to exploit any possibilities for removing redundancy between the different refactorings.

Of course, this latter problem does not arise when the refactorings happen to correspond to FGTs. However, primitive refactorings may, in general, include many actions (FGTs). This is true also of composite refactorings.

9.3 Sequential Dependency between FGT-Based Refactorings

Finding sequential dependencies between refactorings can be based on finding sequential dependencies at the level of the FGTs that constitute the refactorings. As shown in Figure 9.3, representing refactorings R_i and R_j—discussed in the previous example—as collections of FGTs ordered in FGT-DAGs means that the sequential dependency between every pair of FGTs in the two refactorings can be checked. The figure shows that there is a sequential dependency between the two FGTs:

FGT_{ii}: addObject(*P, A, _, _, _, _, public, _, class*)
FGT_{jj}: addObject(*P, A, Attn, _, _, type(basic, int, 0), public, _, attribute*)

where FGT_{jj} sequentially depends on FGT_{ii}. Finding just one case of sequential dependency between a pair of FGTs in the two refactorings is sufficient to establish the sequential dependency between the two refactorings. In the example, the fact that refactoring R_j sequentially depends on refactoring R_i, means that refactoring R_i needs to be applied first.

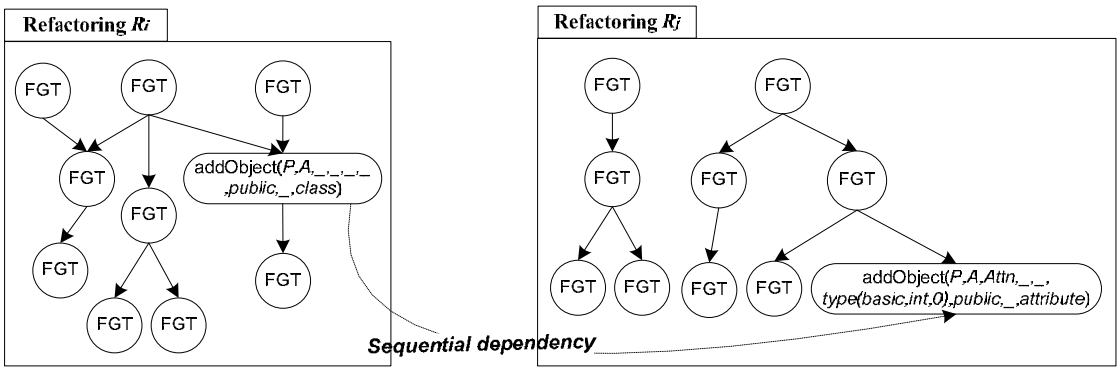


Figure 9.3: Sequential dependency in FGT-based approach

To do that, the various possibilities of sequential dependency that may occur between the different FGTs have to be examined. Recall that these have been pre-catalogued as shown in Figure 4.1 and explained in more details in Appendixes A.1 and A.2. As explained in sections 4.3.2 and 4.3.3, two categories of sequential dependencies between FGTs were identified: Uni-directional FGT sequential dependencies and Bi-directional FGT sequential dependencies.

Note that, one of the advantages of distinguishing between two types of FGT sequential dependencies is that the information can be used to solve the ambiguity problem discussed in the previous section with respect to refactorings. If there is a bi-directional sequential dependency between two FGTs that appear within two respective refactorings, this means that there is ambiguity between the two refactorings as well. In this case, the underlying system should be referenced in order to establish which refactoring should be applied first and which one second in a given context, or whether, in fact, there is a deadlock problem, as discussed in section 9.5 below.

Thus, by identifying the type FGTs involved in two refactorings (uni-directional or bi-directional), the tool can determine whether or not there is a need to reference the underlying system (i.e. in the case of bi-directional sequential dependency). This is the basis for the sequential dependency algorithm, discussed in the next section.

9.4 Sequential Dependency Algorithm

Representing refactorings as a collection of FGT-DAGs allows one to establish exactly at which part of the two refactorings the sequential dependency between the two occurs. In this section, an algorithm (**sequentialDependency**) to find the sequential dependency between two refactorings is defined. The algorithm is base on finding the sequential dependency at the level of FGTs which constitute these refactorings. It uses the **uniDirSD** and **biDirSD** facts already stored in the database of the tool.

As mentioned before, access on the underlying system is a time-consuming process if the system is large. Therefore, the algorithm has been designed to minimize such access. This criterion is taken into consideration in the proposed **sequentialDependency** algorithm by working in three phases:

Phase one: In this phase, the algorithm tries to find the sequential dependency relations between the two refactorings using the **uniDirSD** facts. It takes each FGT from the first refactoring (FGT_{ii}) and checks if it has a **uniDirSD** with any other FGTs of the second refactoring (FGT_{jj}). Finding a single match is enough for the algorithm to determine the sequential dependency between the two refactorings. Note that in this phase the algorithm takes a decision without having to access the underlying system. If the algorithm does not find any of the **uniDirSD** between any pair of FGTs in the two refactorings then the algorithm goes to the next phase.

Phase two: In this phase, the sequential dependency algorithm tries to find the sequential dependency relation between the two refactorings using the **biDirSD** facts. It takes each FGT from the first refactoring (FGT_{ii}) and checks if it has a **biDirSD** with any other FGTs of the second refactoring (FGT_{jj}). If one is found, then the algorithm has to check the underlying system to resolve the ambiguity. If the algorithm does not find any of the **biDirSD** between any pair of FGTs in the two refactorings then the algorithm goes to the next phase.

Phase three: In this phase, the algorithm checks the refactoring-level pre- and postcondition conjuncts of the two refactorings to infer the sequential dependency between the two refactorings. Note that the approach used here is the same as the approach described in the second part of section 9.2 with a major difference: the concern here is just with refactoring-level pre- and postcondition conjuncts and not with the entire set of refactoring pre- and postcondition conjuncts.

It should be noted that the **sequentialDependency** algorithm given below can be used to establish the sequential dependency relationship between any two refactorings that are represented as FGT-DAGs. It is, however, a requirement that the refactorings be both conflict-free and deadlock-free. The matter of deadlock freedom is taken up in section 9.5 below, while the previous chapter has shown how conflict freedom can be established. Three different kinds of conflicts between pairs of refactorings were mentioned: ordering-conflicts (where conflict can be resolved by ordering one of the refactorings before the other); cancelling-conflicts (where conflict can only be resolved by withdrawing one of the refactorings); and removable-conflicts (where conflicts can be resolved by appropriately modifying FGTs that comprise one of the refactorings).

In general, the **sequentialDependency** algorithm can be used to establish the sequential dependency relationships between appropriately selected refactorings in a *batch* of refactorings. The outcome is then one or more **refactoring directed acyclic graph (REF-DAGs)**, as illustrated in Figure 9.4. Each node in a REF-DAG represents *one* of the refactorings and contains the FGT-DAGs of that refactoring. When the **sequentialDependency** algorithm finds that refactoring Y is sequentially dependent on refactoring X, then node X becomes the father of node Y in one of the REF-DAGs. If there is no sequential dependency between two different REF-DAGs then they can be processed and applied in parallel.

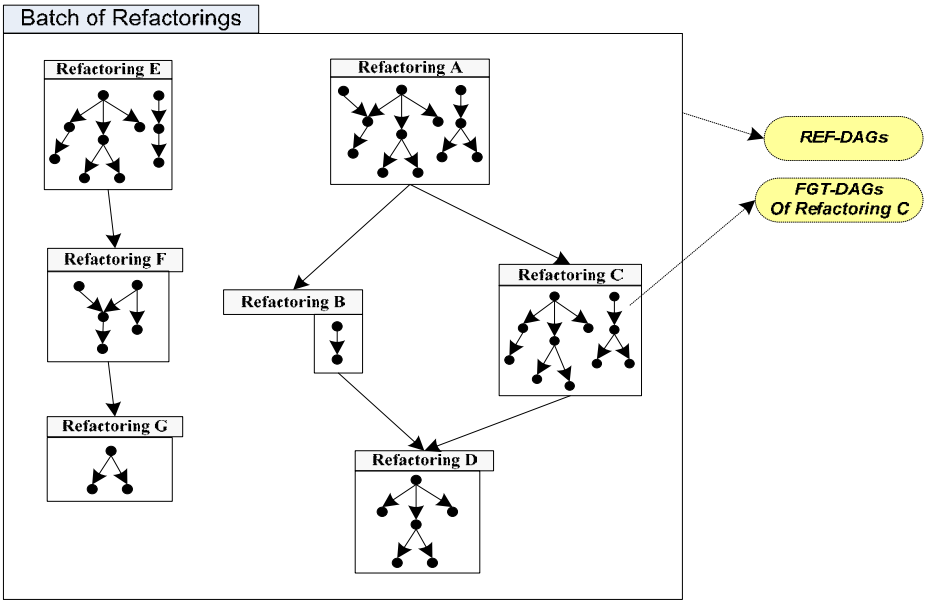


Figure 9.4: Refactoring Directed Acyclic Graphs (REF-DAGs)

Algorithm 9.1 (Sequential dependency algorithm)

sequentialDependency(Ref X, Ref Y)

Input: Ref X: a conflict-free redundancy-free set of FGT-DAGs of refactoring X
Ref Y: a conflict-free redundancy-free set of FGT-DAGs of refactoring Y

Assumption: Ref X & Ref Y are deadlock-free

Output: An indication that X→Y; or that Y→X; or that there is no sequential dependency relationship between X and Y

// Start of phase one

For each FGT_{ii} in $FGT\text{-}DAG_i$ (*starting from the root*) in X **do** {
  For each FGT_{jj} in $FGT\text{-}DAG_j$ (*starting from the root*) in Y **do** {
    Search **unDirSD** facts to find a match with the pair (FGT_{ii}, FGT_{jj}) or the pair (FGT_{jj}, FGT_{ii})
    If there is a match **then** {
      Determine sequential dependency between X & Y
      Return result } //end **If**
  } //end **for**
} //end **for**

// End of phase one. Start of phase two

For each FGT_{ii} in $FGT\text{-}DAG_i$ (starting from the root) in X **do** {
  For each FGT_{jj} in $FGT\text{-}DAG_j$ (starting from the root) in Y **do** {
    Search **biDirSD** facts to find a match with the pair (FGT_{ii}, FGT_{jj}) or the pair (FGT_{jj}, FGT_{ii})
    If there is a match **then** {
      Check the underlying system to determine the direction of the sequential dependency
      Return result } //end **If**
  } //end **for**
} //end **for**

// End of phase two. Start of phase three

{ Check pre- and postcondition conjuncts at the refactoring-level of the two refactorings,
  Return result }

// End of phase three

Algorithm 9.1 gives the pseudo-code for the **sequentialDependency** algorithm. The algorithm takes as input two refactorings X and Y. It then works in a nested loop fashion. For each $FGT\text{-}DAG_i$ of refactoring X, the algorithm starts from the root and takes each FGT_{ii} in $FGT\text{-}DAG_i$.

It checks if there is a sequential dependency between it and each FGT_{jj} in every FGT-DAG_j of refactoring Y, in each case also starting from the root of FGT-DAG_j. For every pair, of ($FGT_{jj,}$ FGT_{ii}) or ($FGT_{ii,}$ FGT_{jj}), the algorithm checks if there is a match with the **uniDirSD** facts. When a match is found this means that there is a sequential dependency between FGT_{ii} and FGT_{jj}, which means a sequential dependency between the two refactorings X and Y. Then the loop breaks and the algorithm returns the result to the calling procedure.

If the nested loop completes without finding any match, then the algorithm goes to the next phase by starting the nested loop again but this time searching for a match with the **biDirSD** facts. If the nested loop completes without finding any match, then the algorithm goes to the next phase to check the refactoring-level pre- and postcondition conjuncts of the two refactorings.

9.5 Deadlock Problem

A deadlock between two refactorings occurs when each one of the two refactorings sequentially depends on the other. In other words, each one of the two refactorings needs the other one to be applied to the system, to satisfy its precondition conjuncts. As a result, none of them can be applied to the system.

The idea is explained in Figure 9.5. The FGTs in the two refactorings X and Y have the following sequential dependency relations:

FGTx1 → FGTy1 (*This means that Refactoring Y is sequentially dependent on Refactoring X*)
FGTy2 → FGTx2 (*This means that Refactoring X is sequentially dependent on Refactoring Y*)

Because the sequential dependencies between the two refactorings go in both directions, it can be concluded that there is a deadlock situation.

Note that in all the algorithms presented in the previous chapters of the thesis, an assumption of deadlock freedom between the different refactorings is considered. The same assumption is also made in the rest of the thesis.

It should be noted that deadlocks can only arise if an irrational attempt is made to refactor an existing system. If users refer to the current system only, their requested refactorings will not result in deadlock with each other, even if they request refactorings on the system independently of one another. Any requested refactoring has to rely on the state of the system

to satisfy its precondition conjuncts, or on applying some other prior refactoring to the system first. In the latter case, the precondition conjuncts of the latter system have to be satisfied by the system's state, etc. A deadlock can only occur if a user mistakenly attempts to satisfy the precondition conjuncts of one refactoring by specifying another, whose precondition depends on the first. One way in which this could happen, for example, is if end users are given the ability to build their own refactorings—as discussed in chapter 12—and this result in unorganised dummy refactorings which have deadlock between each other.

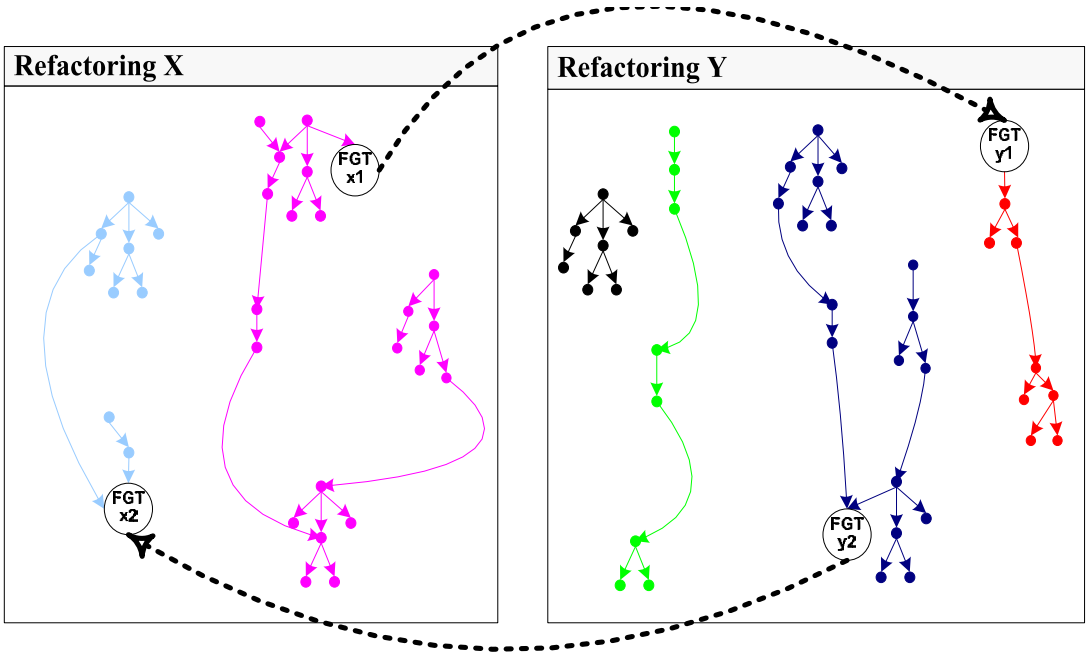


Figure 9.5: Deadlock problem

In order to detect the deadlock between two refactorings, **deadLockDetection** algorithm may be used. Algorithm 9.2 provides the pseudo-code for the deadLockDetection algorithm. The algorithm takes as input two refactorings X and Y. It then searches to find sequential dependencies between each pair of FGTs in the two refactorings. The algorithm works in the three phase manner as in the **sequentialDependency** algorithm described in section 9.4, with the following main difference:

When the algorithm finds the first sequential dependency between a pair of FGTs, it stores this sequential dependency relation and continues with the remaining of FGTs, searching for all other sequential dependency relations. Each time a new sequential dependency relation is discovered, it is checked with the stored one (the first discovered one). If it is in the opposite

direction this means that there is a deadlock, the algorithm is terminated and a "DeadLock" message is returned to the calling procedure; otherwise the algorithm will continue until all FGT pairs have been checked.

If no deadlock is discovered then the algorithm will start with the third phase. It will check the pre- and postcondition conjuncts at the refactoring-level to infer if there is a deadlock between the two refactorings or not. For this, the algorithm uses the following rule:

If (refactoring-level precondition conjuncts of refactoring X contains some of the postcondition conjuncts of refactoring Y) and (refactoring-level precondition conjuncts of refactoring Y contains some of the postcondition conjuncts of refactoring X), then there is a deadlock between the two refactorings.

In addition, the algorithm will check the sequential dependencies discovered in this phase with the stored one (if any) from the previous two phases.

Note that the assumption to date has been that X and Y are conflict-free in a strict sense—i.e. there is no ordering-conflict between X and Y. The deadLockDetection algorithm can be modified in an obvious way to detect possible deadlock between X and Y if this restriction is lifted. It would simply be a matter of noting the direction of the ordering-conflict at the start, and declaring a deadlock between X and Y if a sequential dependency is later discovered in the opposite direction.

Algorithm 9.2 (Deadlock detection algorithm)

deadLockDetection(Ref X, Ref Y)

Input: Ref X: a conflict-free set of FGT-DAGs of refactoring X

Ref Y: a conflict-free set of FGT-DAGs of refactoring Y

Output: An indication of whether or not X & Y are deadlocked

Let SDFound = false

// Start of phase one

For each FGT_{ii} in $FGT\text{-}DAG_i$ (*starting from the root*) in X **do** {

  For each FGT_{jj} in $FGT\text{-}DAG_j$ (*starting from the root*) in Y **do** {

    Search **unDirSD** facts to find a match with the pair ($FGT_{ii,}$ FGT_{jj}) or the pair ($FGT_{jj,}$ FGT_{ii})

    If there is a match **then** determine the sequential dependency (X→Y) or (Y→X)

    If (! SDFound) **then** {store the SD relation, SDFound=true}

      Else {compare the new SD relation with the stored one

        If it is in the opposite direction **then** return "DeadLock"}

  } //end **for**

} //end **for**

// End of phase one. Start of phase two

For each FGT_{ii} in $FGT\text{-}DAG_i$ (starting from the root) in X **do** {

  For each FGT_{jj} in $FGT\text{-}DAG_j$ (starting from the root) in Y **do** {

    Search **biDirSD** facts to find a match with the pair ($FGT_{ii,}$ FGT_{jj}) or the pair ($FGT_{jj,}$ FGT_{ii})

    If there is a match **then** check the underlying system to determine the direction of the SD

    If (!SDFound) **then** {store the SD relation, SDFound=true}

      Else {compare the new SD relation with the stored one

        If it is in the opposite direction **then** return "DeadLock"}

  } //end **for**

} //end **for**

// End of phase two. Start of phase three

Check pre- and postcondition conjuncts at the refactoring-level of the two refactorings

Determine the sequential dependencies between X & Y

Compare the discovered SDs with each other and with the stored one (*if there is*)

If there are two SDs in the opposite directions **then** return "DeadLock"

Return *"DeadLock-Free"*

// End of phase three

9.6 LAN Motivated Example

Consider the motivating example given in chapter 6. To find the sequential dependency between the three proposed refactorings (**pullUpMethod**, **createClass** and **encapsulateAttribute**), the sequential dependency algorithm will take FGT-DAGs of two refactorings each time to check if there is a sequential dependency between them. As a result of executing the algorithm, while checking the two refactorings **pullUpMethod** and **createClass** during phase one, the algorithm finds that FGT:

addObject(Lan, Server, accept, _, _, type(basic, void, 0), public, [(p, type(complex, Packet, 0))], method)

in refactoring **pullUpMethod** is sequentially dependent on FGT:

addObject(*Lan, Server, _, _, _, _, public, _, class*)

in refactoring **createClass**. As a result, the sequential dependency algorithm indicates that refactoring **pullUPMethod** is sequentially dependent on refactoring **createClass.** The resulting REF-DAG will be as shown in Figure 9.6.

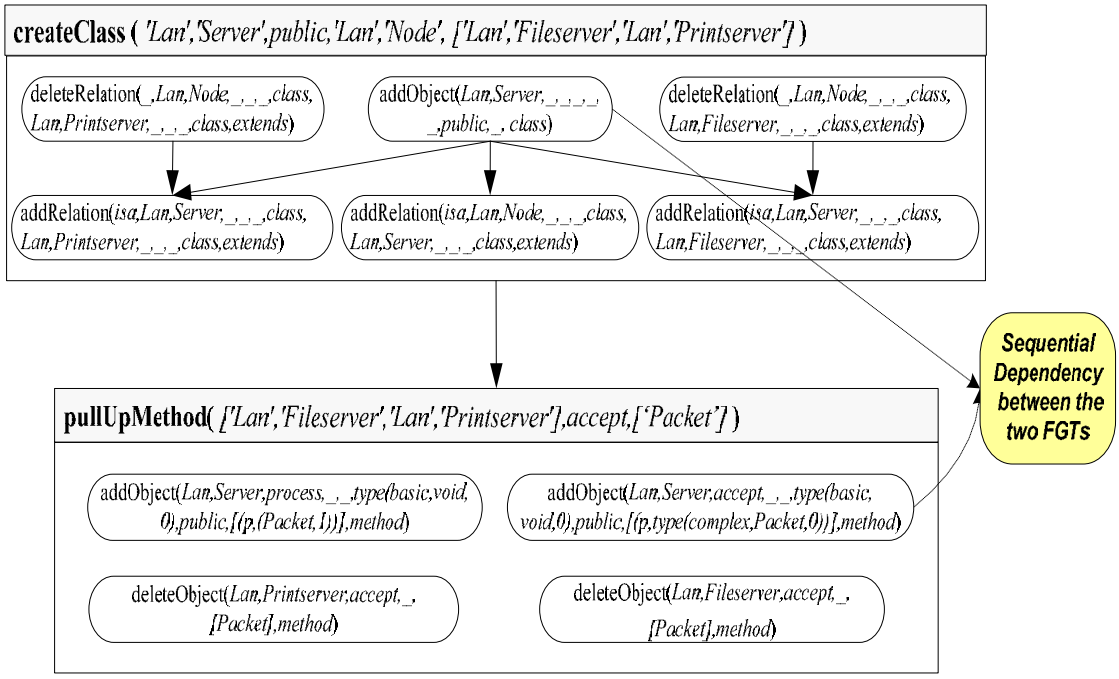
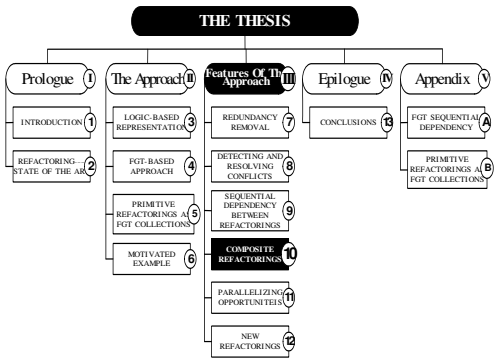


Figure 9.6: Sequential dependency between refactorings createClass & pullUpMethod

Chapter 10

COMPOSITE REFACTORINGS

10.1 Introduction



A developer who restructures a system, starts with some design goals in mind. In practice, it is likely that a single primitive refactoring will not meet the design goal in isolation. Instead, it may be necessary to jointly group primitive refactorings into a "batch" [38, 70], which is then applied to the model as one unit. Such a batch of primitive refactorings that addresses one or more of a developer's design goals is termed a composite refactoring. Of course, the composite refactoring created in this way could subsequently be combined with others, thus creating new ones, and so on [49].

Composite refactoring are conventionally specified as a *sequence* of primitive refactorings, the assumption being that they will be applied in that specific order. However, a necessary (but not sufficient) condition for successfully applying such a sequence to a system is that it should respect the so-called *sequential dependencies* between the constituent primitive refactorings. Briefly, if one or more precondition conjuncts of refactoring P are logically entailed by the postcondition of refactoring Q, then P is sequentially dependent on Q, denoted by Q→P.

For the purposes of the present discussion, it will be assumed that the primitive refactorings in a composite have been rationally selected—i.e. that there are no conflicts between the various primitive refactorings. (The previous chapters have also shown how such conflicts may be detected.) In order to illustrate relevant concepts, consider seven primitive refactorings, A…G, that are to be used in composite refactoring X. Suppose that they have the following sequential dependencies:

$$A \rightarrow B,\ A \rightarrow C,\ E \rightarrow F,\ F \rightarrow G,\ B \rightarrow D,\ C \rightarrow D$$

Assume, too, that the user has specified X as the following refactoring sequence:

$$X = \langle A, E, F, B, C, G, D \rangle$$

Note that this sequence respects the sequential dependencies: any refactoring that is sequentially dependent on another, will be processed after the latter. (As an aside, note that X embodies the following mutually independent refactoring subsequences:

<A→B, A→C, B→D, C→D> and <E→F, F→G>

In principle, these subsequences may be processed in parallel.)

The straightforward approach to apply such a composite to some system, is to focus on the primitive refactorings' pre- and postconditions in isolation, as shown in Figure 10.1. To applying the composite, the precondition of the first primitive refactoring, A, is checked against the system's state. If it is satisfied then refactoring A is applied. The precondition of the next primitive refactoring, E, is checked against the system, it is applied, and so on.

Dealing with composite refactorings in this way is vulnerable to the rollback problem: if, at some point, a precondition of one of the primitive refactorings in the composite is not satisfied, then the refactoring tool has to rollback all the primitive refactorings in the composite that had previously been applied to the system, so as to restore the system to its original state.

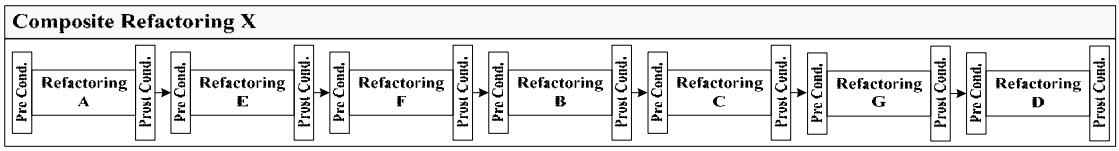


Figure 10.1: Straightforward approach

In [35, 38, 52, and 70], the concept of a composite precondition / postcondition, as illustrated in Figure 10.2, was proposed to deal with this rollback problem. The idea is to derive the *composite's* precondition and postcondition by considering the pre- and postconditions of its individual primitive refactorings. Note that the derived composite precondition conjuncts are not simply the conjunction of all precondition conjuncts of its constituent primitive refactorings. Doing so would neglect the transformations performed between the evaluation of the different conditions. For instance, assume that a composite X consists of two primitive refactorings (R_1 and R_2) where R_2 sequentially depends on R_1. Suppose the precondition of R_1 is P_1^P_2 and the precondition of R_2 is P_3^P_4. Suppose, also, that the postcondition of R_1 is P_3 (or, more generally, that it logically implies P_3, but not P_4). Then the precondition of the composite X is P_1^P_2^P_4. The same also applies for deriving the postcondition of the composite.

At the time of refactoring, the set of composite precondition conjuncts is checked against the system state before applying any primitive refactoring in the sequence to the system. If these are satisfied, then the primitive refactorings may be applied to the system in the sequence given, or indeed in any sequence that respects the sequential dependencies between primitive refactorings. Under such circumstances, no rollback will be necessary.

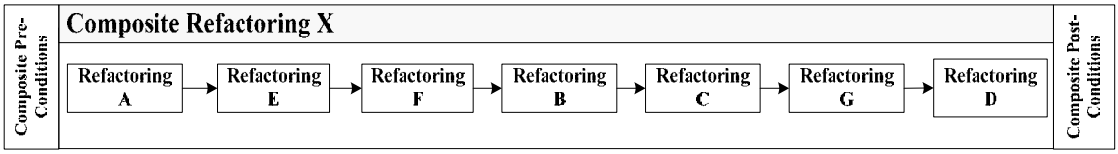


Figure 10.2: Composite refactoring in composite precondition approaches

The following section considers the implications of composite refactorings in the context of the FGT paradigm proposed in this thesis.

10.2 FGT-based Composite Refactoring

Recall that chapter 5 has catalogued the commonly mentioned primitive refactorings, together with their associated preconditions. That chapter also indicated how a given primitive refactoring that is to be applied on some system can be expressed as an equivalent FGT-list. As a consequence, an FGT-based procedure for composing several primitive refactorings, stored as FGT-lists, may seem obvious. Simply select the desired sequence of primitive refactorings that are to form a composite refactoring and place them in a list that respects their sequential dependency.

At this point, the composite-level's pre- and postcondition can be computed from the refactoring-level pre- and postconditions of all the refactorings inside the composite, exactly in the same way as described by previous authors. However, to determine whether or not the composite can be applied to the system without rollback, it is now no longer sufficient merely to check the composite-level precondition. In addition, the FGT-enabling preconditions of FGT-DAGs in the composite should also be checked against the system before deciding whether or not to apply the composite.

The fact that the FGT-enabling preconditions have to be computed means that some additional work has to be undertaken when using FGTs, if rollback is to be avoided. However, there are

potential gains to be had from this cost, but to exploit them; the FGT-lists of individual primitive refactorings need to be decomposed into equivalent FGT-DAGs as described in chapter 4. Under these circumstances, it is at least of theoretical interest to explore whether the FGT-DAGs can be merged further. The practical value of doing this will be considered at the end of the chapter.

The procedure **compositeRefactoring**, to be described below, merges the FGT-DAGs of primitive refactorings, and determines the composite- and FGT-enabling preconditions. The procedure assumes that the following holds:

a. All the refactorings (whether or not they are primitive refactorings) that the developer wants to include in the composite are redundancy-free and conflict-free.

b. The refactorings are in a sequence whose order respects the sequential dependencies between them.

In order to build the composite refactoring X described in the previous section, the procedure **compositeRefactoring** is called as follows:

compositeRefactoring([*Refactoring A, Refactoring E, Refactoring F, Refactoring B, Refactoring C, Refactoring G, Refactoring D*])

The procedure then executes the steps given below, which will be explained in more detail later:

Step 1: Generate the system-specific FGT-list corresponding to each primitive refactoring in the composite.

Step 2: Build a set of FGT-DAGs for each FGT-list.

Step 3: Determine the sequential dependency relationship between every pair of FGTs in the composite refactoring.

Step 4: Use refactoring-level pre- and postconditions to infer (a) possible undetected sequential dependencies between refactorings; and (b) composite-level pre- and postconditions.

Step 5: Remove all redundancies from the FGT-DAGs.

Step 6: Determine the FGT-enabling precondition of each FGT-DAG in the composite.

Clearly, the first two steps can be carried out by using the primitive refactoring procedures described in chapter 5 (in step 1); and the **build-FGT-DAG** algorithm found in section 4.3.4 (in step 2).

In the third step, it is obviously unnecessary to check dependency relationships between FGTs within each primitive refactoring's set of FGT-DAGs, since these are already reflected by the arcs in the FGT-DAGs. To check the dependency relationships between FGTs in different FGT-DAG sets, an adapted form of the algorithm in section 9.4 (to determine the sequential dependency between two refactorings) can be used. Recall that this algorithm operates on a pair of sets of FGT-DAGs and that it terminates upon finding a sequential dependency between a single pair of FGTs.

The first adaptation is to find *all* the sequential dependencies between FGTs in the two refactorings, and *not* just the first one. Under the assumption that the original list of primitive refactorings were rationally assembled, there will not be any circular paths (i.e. conflicts) generated by this process.

The second adaptation is that, when dealing with **biDirSD**, it is unnecessary to access the underlying system to determine the direction of the sequential dependency. Instead, the direction may be inferred from the order of primitive refactorings in the original list that is provided as input to the **compositeRefactoring** procedure.

Step 3 therefore, involves the following on each pair of primitive refactorings, say S and T, represented respectively as FGT-DAG sets: Consider all FGT pairs comprising of an FGT in S and an FGT in T. Use the facts **uniDirSD** and **biDirSD** to determine whether or not they are sequentially dependent, and if so, connect them by an appropriate sequential dependency arc.

The outcome of step 3 is schematically shown by the dashed arrows in Figure 10.3(a), connecting various FGTs across different refactorings.

The bold arrows between refactorings in Figure 10.3(a) indicate sequential dependencies between them. In this particular example, this may be directly inferred from the fact that sequential dependencies had been established in step 3 between one or more their constituent FGTs.

However, in general, it may be the case that there is no such FGT-level sequential dependency, but nevertheless, a sequential dependency that is related to the possibility that one (or more) refactoring-level postcondition establishes one (or more) refactoring-level precondition

conjuncts of another[1]. For example, the **pullUpAttribute** primitive refactoring requires, as a refactoring-level precondition, that the relevant attribute's access mode should not be *private* in any of the relevant subclasses. It may or may not be the case that the original sequence of primitive refactorings contains a primitive refactoring(s) to change everywhere the attribute's access mode to *public* or *protected*. If there are such primitive refactorings that change the access mode, then **pullUpAttribute** is sequentially dependent on them. If there are no such primitive refactorings, then the requirement that the attribute should not have a *private* access mode in any of the relevant subclasses, becomes a composite-level pre-requisite.

It is the task of step 4 to compare the refactoring-level pre- and postconditions of a primitive refactoring pair, S and T, to infer which of the pre- and postcondition conjuncts should serve as composite-level pre- and postconditions respectively, and which of them indicate a refactoring-level sequential dependency that was not established in step 3. Note that in the latter case, an application of the composite refactoring must ensure that the application of the various FGTs respects such refactoring-level sequential dependencies. Figure 10.3(b) shows the combined effect of steps 3 and 4: a new set of FGT-DAGs made up of the original sets of FGT-DAGs, together with their associated composite-level pre- and postconditions.

However, before applying the composite, step 5 should be executed to remove possible redundancies that have arisen as a result of combining FGT-DAGs in the previous steps. The **reduction** algorithm described in section 7.4 may be used for this purpose.

Note that in step 6, the determination of the FGT-enabling precondition of each FGT-DAG in the composite, necessarily has to take place after reduction. This is to account for actual FGTs that are to be used in the composite, rather than those that were directly implied by the FGT-list that had been derived from the input primitive refactoring list.

[1] Note that the matter is, in fact, somewhat more subtle. It is also possible that an FGT postcondition establishes part of a refactoring level's precondition; or that a refactoring-level postcondition establishes part of an FGT's precondition. Example 2 below will give an illustration of the first of these possibilities.

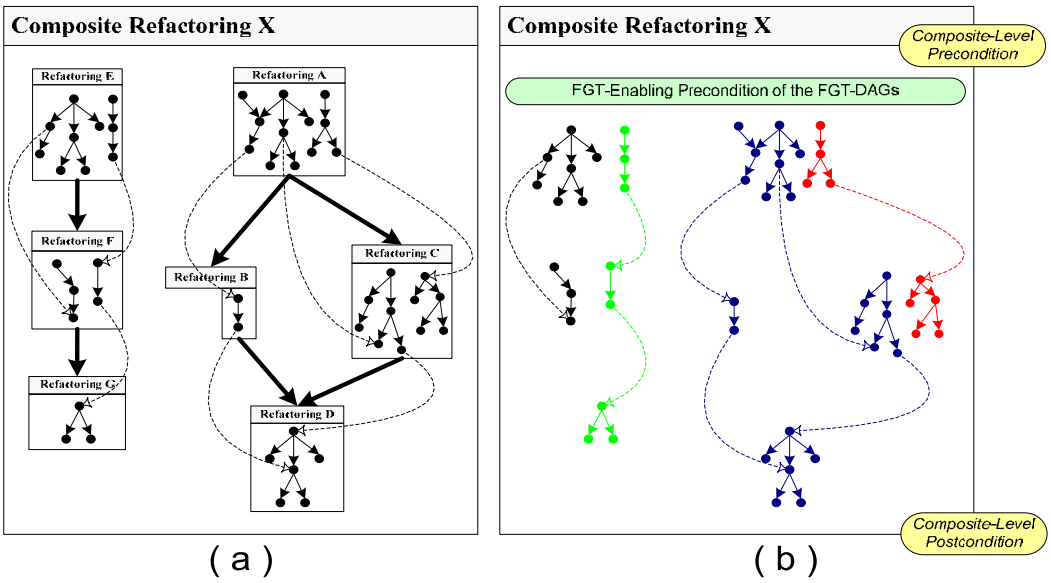


Figure 10.3: Composite refactoring in FGT approach

In summary, then, the **compositeRefactoring** procedure generates (a) a set of independent redundancy-free FGT-DAGs that reflects the actual transformations needed to achieve the composite refactoring; (b) a set of composite-level pre- and postconditions; and (c) the FGT-enabling precondition of each FGT-DAG in the composite.

At the time of refactoring, the composite refactoring is executed in two phases. In the first phase, two levels of preconditions are checked: (a) the composite-level preconditions, and (b) the FGT-enabling preconditions of the various FGT-DAGs.

If all relevant preconditions are satisfied then, in a second phase, the different FGTs are applied to the underlying system, again here, in the same order as they appear in the different FGT-DAGs. Processing the composite in two phases solves the rollback problem because the tool will not apply any FGT in the composite before checking that system to be refactored complies with the preconditions at the composite-level and FGT-enabling level.

The processed composite refactoring is like any other refactoring. All the operations that the approach offers for dealing with refactorings can also be carried out on composite refactorings (reduction, conflict detection & resolving, sequential dependency, and parallelization).

10.3 Examples

10.3.1 encapsulateAttribute Composite Refactoring

To illustrate the FGT approach for dealing with composite refactorings, the composite refactoring **encapsulateAttribute**—which is used to prevent direct accesses to a specific attribute—will be given as an example.

Figure 10.4(a) gives a UML class diagram for a simplified *College* system. The system has a package called *College* with three classes *Teacher*, *Student* and *Registration*. Note that the information extracted from the class diagram alone is not sufficient for refactoring. For example, if a method *m* is to be deleted from the class diagram using the primitive refactoring *deleteMethod*, then that method should be not referenced by any other object elements in the class diagram, and this kind of referencing information is not in the UML class diagram. The underlying logic representation of the class diagram should include this kind of extra information. To get such information, we have to refer to the code-level implementation of the system. Figure 10.4(a) shows such information represented as dashed arrows between the different object elements of the class diagram.

Suppose that one of the suggested enhancement to the class diagram of the *College* system is to encapsulate the attribute *Mark* in the *Student* class. This refactoring is useful for increasing modularity, by avoiding direct accesses of the local state of a *Student*. For this restructuring, the composite refactoring **encapsulateAttribute** can be constructed.

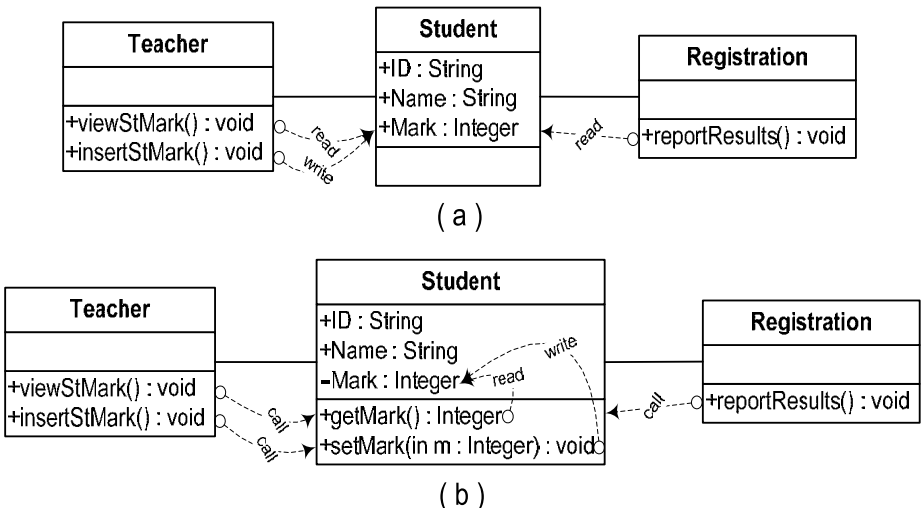


Figure 10.4: A simplified UML class diagram of a *college* system. (a) before and (b) after refactoring

The order of the primitive refactorings in the composite is shown in Figure 10.5. Note that the order reflects the sequential dependency that exist between the different refactorings in the composite. According to the order, a refactoring tool should first add the getter and setter methods. Then it should redirect the destination of all the *read/write* accesses from the attribute to them. After this stage, the attribute is not referenced by any object in the system. Therefore, the refactoring tool can change the access mode of the attribute from *public* to *private*.

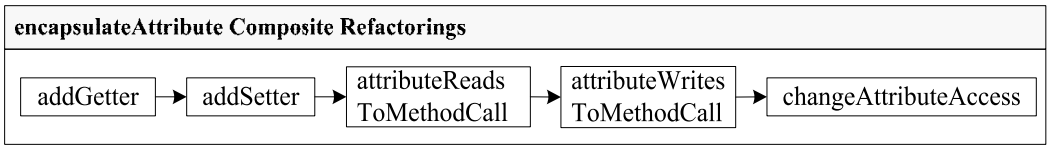


Figure 10.5: encapsulateAttribute composite refactoring

In the refactoring tool, in order to encapsulate the attribute *College.Student.Mark*, the procedure:

compositeRefactoring([*addGetter, addSetter, attributeReadsToMethodCall, attributeWritesToMethodCall, changeAttributeAccess*])

is used, where the arguments in the procedure refer to the primitive refactorings that are included in the composite **encapsulateAttribute.** (Note that, for conciseness, the arguments above are given in an abbreviated above. Their full form—as they should actually appear in the procedure call—is given in the middle column of Table 10.1.) As discussed above, the procedure will produce an FGT-list which represents the transformation actions to be performed as part of the encapsulation process. This FGT-list is shown in the right-hand column of Table 10.1.

The FGT-lists produced by each primitive refactoring in the composite are then allocated to one or more FGT-DAGs. Thereafter, the sequential dependencies between the different FGTs in the different primitive refactorings are found. These are shown as dashed arrows in Figure 10.6(a).

The solid arrows show the sequential dependencies between primitive refactorings that can be inferred from the dashed arrows. Note that the fact that there are only solid arrows between refactoring primitives whose FGTs show some sequential dependencies (i.e. the dashed arrows) is an indication that in this particular example, no primitive refactoring dependencies are induced by considering refactoring-level pre- and postconditions in step 4 of the

compositeRefactoring procedure. In fact, in this example, no composite-level preconditions are to be found.

Table 10.1: encapsulateAttribute refactoring

Comp-osite Ref.	Sequence Of Primitive Refactorings	Sequence Of FGTs For Each Primitive Refactoring
encapsulateAttribute(*'College', 'Student', 'Mark'*)	**addGetter**(*'College', 'Student', 'Mark'*)	**FGT1:** addObject(*College, Student, getMark, _, _, type(basic,int,0), public,[], method*) **FGT2:** addRelation(*_,College,Student,getMark,_,[],method,College, Student,Mark, _,_, attribute, read*)
	addSetter(*'College', 'Student', 'Mark'*)	**FGT3:** addObject(*College,Student,setMark,_,_, type(basic,void,0), public, [(p, type(basic,int,0))], method*) **FGT4:** addRelation(*_,College,Student,setMark,_,[int], method, College, Student, Mark,_,_, attribute, write*)
	attributeReadsToMethod-Call(*'College', 'Student', 'Mark', 'College', 'Student',getMark, []*)	**FGT5:** deleteRelation(*_,College,Teacher,viewStMark,_,[], method, College, Student, Mark,_,_, attribute,read*) **FGT6:** deleteRelation(*_,College,Registration, reportResults, _,[], method, College, Student, Mark,_,_,attribute, read*) **FGT7:**addRelation(*_,College,Teacher,viewStMark,_,[],method, College,Student,getMark,_,[],method,call*) **FGT8:** addRelation(*_,College,Registration,reportResults,_,[],method, College, Student, getMark,_,[], method, call*)
	attributeWritesToMethod-Call(*'College', 'Student', 'Mark', 'College', 'Student', setMark, [int]*)	**FGT9:** deleteRelation(*_,College,Teacher,insertStMark,_,[],method, College,Student, Mark,_,_,attribute,write*) **FGT10:** addRelation(*_,College,Teacher,insertStMark,_,[],method, College,Teacher, setMark, _,[int], method, write*)
	changeAttributeAccess(*'College', 'Student', 'Mark', private*)	**FGT11:** changeOAMode(*College,Student,Mark,_,_,attribute, public, private*)

Since there are no reductions to be made in step 5 of the **compositeRefactoring** procedure, Figure 10.6(b) shows the final result. It clearly indicates which FGTs may be applied independently to the system. For example, FGTs 1, 3, 5, 6 and 9 may be launched independently, while FGT 11 can only be applied once FGTs 5, 6 and 9 have completed.

Step 6 requires that the FGT-enabling precondition of each FGT-DAG in the composite has to be determined. Since there is no composite-level precondition in this example, the system should comply with the FGT-enabling precondition. If it does so, then the composite may be directly applied to the system without further checking of preconditions.

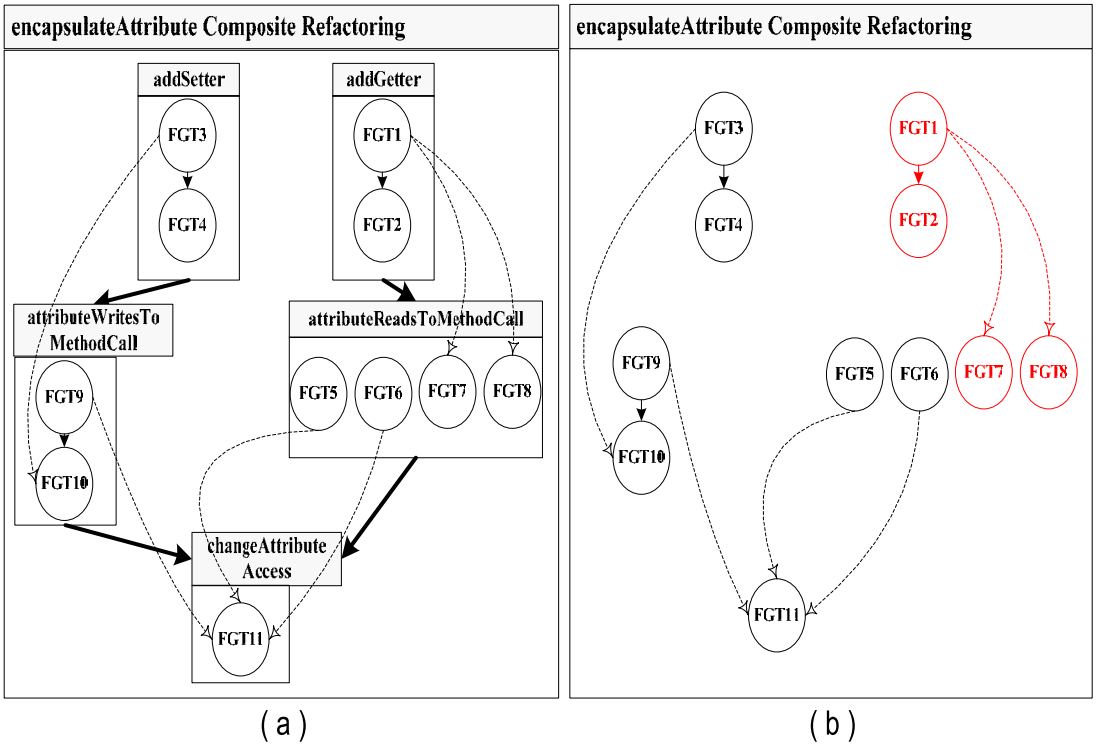


Figure 10.6: encapsulateAttribute composite refactoring in FGT approach

10.3.2 enh-pullUpAttribute Composite Refactoring

A second example is provided to illustrate two aspects relating to step 4 that were not illustrated in the previous example: the way in which the composite-level precondition is derived; and the way in which sequential dependencies arise when FGT postconditions establish refactoring-level preconditions.

Suppose that the developer wants to create a new composite refactoring called **enh-pullUpAttribute**. The aim of the composite is to pull up an attribute from a set of subclasses to their common superclass. One of the precondition conjuncts of the primitive refactoring **pullUpAttribute**, described in section 5.3.2.8, is that the access mode of the attribute in all the subclasses where it is defined should not be *private*. The new proposed composite refactoring in this example solves this problem by changing the access mode of the attribute from *private* to *protected* which will give the ability to pull up it. The composite also defines a getter and a setter method for the pulled up attribute after pulling it up to the superclass. The composite refactoring **enh-pullUpAttribute** consists of the following actions:

1. Change the access mode of the attribute from *private* to *protected* in all the subclasses where the access mode of the attribute is *private*. This is done by using the primitive refactoring **changeAttributeAccess.**

2. Pull up the attribute from all the subclasses where it defined to their common superclass. This is done by using the primitive refactoring **pullUpAttribute**

3. Add getter and setter methods for the attribute that is now located in the superclass. This is done by using the primitive refactorings **addGetter** and **addSetter**.

Figure 10.7(a) gives a UML class diagram for a simplified system. The system has a package *P* with three classes *A*, *B* and *C*. *C* is the superclass of *A* and *B*. class *A* has an attribute *x* with *private* access mode. Class *B* has an attribute *x* with *protected* access mode.

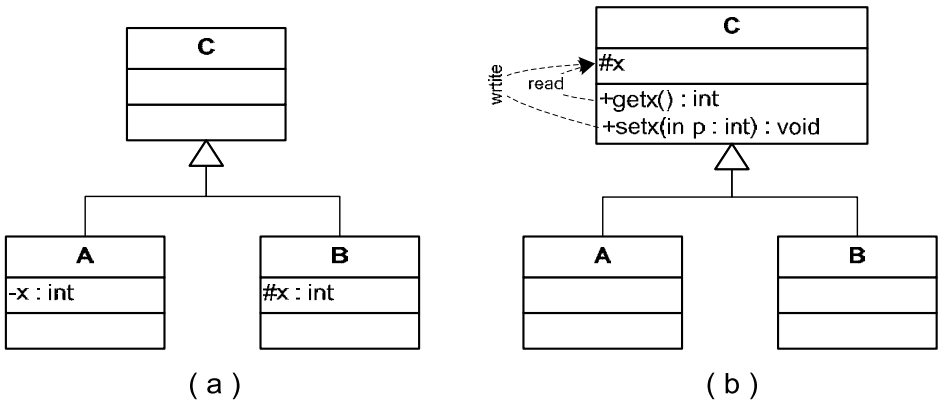


Figure 10.7: A simplified UML class diagram. (a) before and (b) after refactoring

Suppose that the developer wants to pull up the attribute *x* from the subclasses *A* and *B* to the superclass *C*. For this restructuring the composite refactoring **enh-pullUpAttribute** can be constructed. Note that, for this case, the primitive refactoring **pullUpAttribute** cannot be used directly because the access mode of the attribute *A.x* is *private* which means that the precondition of the refactoring will be not satisfied.

In the refactoring tool, in order to pull up the attribute *x*, the procedure:

compositeRefactoring([changeAttributeAccess(*'P', 'A', x, protected*), pullUpAttribute(*'P', 'C', x*), addGetter(*'P', 'C', x*), addSetter(*'P', 'C', x*)])

is used, where the arguments in the procedure refer to the primitive refactorings that are included in the composite **enh-pullUpAttribute**. The procedure will produce an FGT-list

which represents the transformation actions to be performed as part of the pull up process. This FGT-list is shown in the right-hand column of Table 10.2.

Table 10.2: enh-pullUpAttribute refactoring

Comp-osite Ref.	Sequence Of Primitive Refactorings	Sequence Of FGTs For Each Primitive Refactoring
enh-pullUpAttribute (*'P', 'C', x*)	changeAttributeAccess (*'P', 'A', x, protected*)	**FGT1**: changeOAMode(*P,A,x,_,_,attribute, private, protected*)
	pullUpAttribute (*'P', 'C', x*)	**FGT2**: addObject(*P,C,x,_,_, type(basic,int,0),protected,_,attribute*) **FGT3**: deleteObject(*P,A, x,_,_, attribute*) **FGT4**: deleteObject(*P,B, x,_,_, attribute*)
	addGetter(*'P', 'C', x*)	**FGT5:** addObject(*P,C,getx,_,_,type(basic,int,0), public,[],method*) **FGT6:** addRelation(*_,P,C,getx,_,[],method,P,C,x,_,_,attribute,read*)
	addSetter(*'P', 'C', x*)	**FGT7:** addObject(*P,C,setx,_,_,type(basic,int,0), public,[(p, type(basic, int,0))], method*) **FGT8:** addRelation(*_,P,C,setx,_,[int], method,P,C,x,_,_,attribute,write*)

The FGT-lists produced by each primitive refactoring in the composite are then allocated to one or more FGT-DAGs. Thereafter, the sequential dependencies between the different FGTs in the different primitive refactorings are found. These are shown as dashed arrows in Figure 10.8(a). The sequential dependencies between the primitive **pullUpAttribute**, **addGetter** and **addSetter** are inferred from these dashed arrows and represented as solid arrows in Figure 10.8(a).

In step 4 of the **compositeRefactoring** procedure, the refactoring-level pre- and postconditions are used to infer (a) possible undetected sequential dependencies between refactorings; (b) composite-level pre- and postconditions. In this example, the primitive refactoring **pullUpAttribute** inside the composite **enh-pullUpAttribute** has the following refactoring-level precondition conjuncts:

1. The attribute to be pulled up has the same type definition in all the subclasses in which it is defined.

2. The attribute to be pulled up does not have access mode *private* in any of the subclasses of the superclass

Let P.A denotes class A in package P, P.B denotes class B in package P, etc. Then, when **pullUpAttribute**'s refactoring-level precondition conjuncts are instantiated in terms of the objects in the system to be refactored, they become:

1. The attribute *x* has type *int* in both P.A and P.B.

2. The access mode of *x* in P.A is not *private*.

3. The access mode of *x* in P.B is not *private*.

Since the conjunct in line 2 is satisfied by the postcondition conjuncts of FGT1 in Table 10.2, the two primitive refactorings **changeAttributeAccess** and **pullUpAttribute** are sequentially dependent. This relation is represented by the solid arrow between the two refactorings in Figure 10.8(a).

The composite-level precondition conjuncts are those that remain after removing the conjunct in line 2, namely:

1. The attribute *x* has type *int* in both P.A and P.B.

2. The access mode of *x* in P.B is not *private*.

Since the **pullUpAttribute** is the only primitive refactoring in the composite which has refactoring-level preconditions, there are no other conjuncts in the composite-level precondition.

Since no reductions are possible when merging the various FGT-DAGs, Figure 10.8(b) shows the final result of the composite refactoring **enh-pullUpAttribute**. Note that in this case, it is represented as a single FGT-DAG. The composite-level precondition conjuncts are represented in a yellow box at the top of the composite. Note that the sequential dependencies between refactorings in the composite that were discovered by referring to refactoring-level pre- and postconditions are represented by making a link between all the leaf FGTs in the first primitive and all the root FGTs in the second primitive. In this example, the two primitive refactorings **changeOAMode** and **pullUpAttribute** have to be linked in this way. As a result, links are created between FGT1 and each of FGT2, FGT3, and FGT4. This ensures that at the refactoring time, none of the **pullUpAttribute's** FGTs will be executed until FGT1 is executed.

The FGT-enabling precondition conjuncts of the FGT-DAG are represented in a green box at the top of the composite. They can be inferred from its constituent FGTs as consisting of the following conjuncts:

1. The access mode of *x* attribute in P.A is *private*.
2. P.C exists.
3. There is no *x* attribute in P.C or in any of its ancestors.
4. P.B.*x* exists.
5. There is no *getx* method in P.C or in any of its ancestors.
6. There is no *setx* method in P.C or in any of its ancestors.

Checking the composite-level precondition as well as the FGT-enabling precondition of the FGT-DAG against the system in Figure 10.7(a) verifies that all the various conjuncts hold. Therefore the FGTs in Figure 10.8(b) may be applied to the system without the danger of rollback. The result is then the system depicted in Figure 10.7(b).

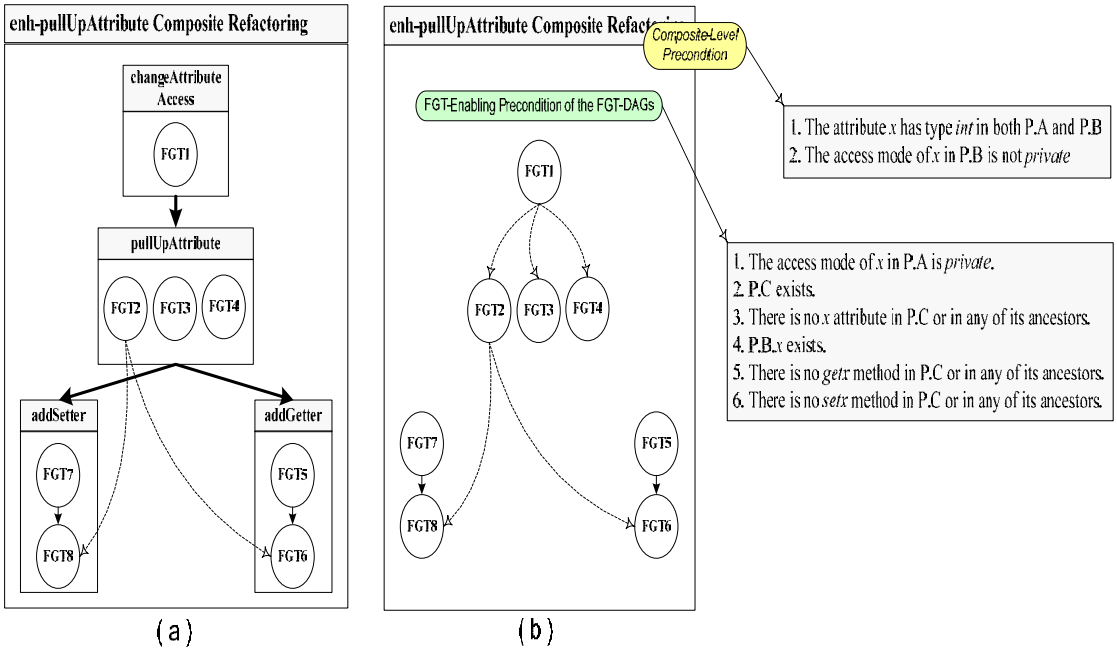


Figure 10.8: enh-pullUpAttribute composite refactoring

10.4 Reflection on this Chapter

This chapter has shown how composite refactorings can be built in the context of the FGT paradigm. It has been seen that FGT-DAGs can be merged, and that a composite-level precondition as well as FGT-enabling preconditions which are then used to avoid rollback. Furthermore, redundancies that may arise because of the merging of FGT-DAGs can be eliminated.

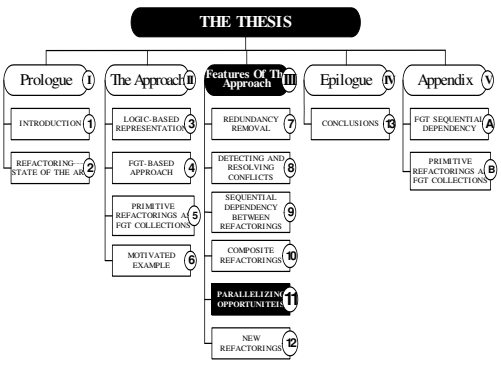
While the discussion has been in terms of an initial set of primitive refactorings, there is nothing to prevent the ideas developed in this chapter being carried over to compose composite refactorings from other composite refactorings. The approach developed in the previous chapter can then be used to determine sequential dependencies between such composites. As the size of composites grows, it may reasonably be conjectured that the scope for conflicts arising between them, and the scope for detecting reductions will increase. Again, these matters can be dealt with as described in chapters 7, 8 and 9.

Whether or not there will be a need for ever-larger and more complicated refactorings in the future, is a matter of conjecture. To the extent that there is, it would seem that the features described above will be of practical importance. What is clear, however, is that FGT-DAGs expose opportunities for parallel implementation. This will be discussed in the next chapter. The chapter after that will examine the implications of all of the above on providing end-user support for building new refactorings.

Chapter 11

PARALLELIZING OPPORTUNITIES

11.1 Introduction



In the FGT approach, opportunities for parallelizing are manifested at the time of refactoring and also during the process of reduction, detecting conflicts, determining sequential dependencies, and generating composites between refactorings. This is because of the ability of the approach to represent refactoring as a collection of FGTs, which are distributed among different FGT-DAGs according to their sequential dependency relations. These FGT-DAGs are independent and can be managed concurrently.

In previous approaches parallelizing are discussed at the level of refactorings. Given a chain of refactorings, Roberts in [70] pointed out that the dependency relationships between refactorings can be used to determine which sets of refactorings within the chain can be performed in parallel, and which ones must be performed sequentially. Each chain can be assigned to a separate processor.

Parallelizing in the proposed approach goes one level down by expressing parallelizing at the FGTs level, which offers the possibility of parallelizing the transformation inside one refactoring. The benefit of this can be easily seen, especially in respect of large refactorings such as composite refactorings with many FGTs inside it. For example, Figure 10.6(b) showed that the **encasulateAttribute** refactoring for the *Mark* attribute in the *College* system ends up with two FGT-DAGs, which can be applied and processed in parallel.

While it is beyond of the scope of this thesis to define parallel versions of the different algorithms developed throughout the thesis, the next section suggests, in overview, some of the ways in which parallelization can be exploited in the various FGT-related algorithms.

11.2 Parallelizing Opportunities

Parallelizing opportunities can be achieved in more than one place:

A. In **reduction algorithm**: As described in section 7.5 that the reduction algorithm works separately on each FGT-DAG. In a parallel version of the reduction algorithm, each FGT-DAG can be assigned to a separate processor.

B. In **conflict detection algorithm**: As described in section 8.4, the detection algorithm checks if there is a conflict between a given FGT in each FGT-DAG of refactoring X and the FGTs in all the other FGT-DAGs of refactoring Y. If a conflict is detected, then the algorithm has to resolve the conflict, either by withdrawing a refactoring, or by modifying an FGT-DAG in a refactoring. Various parallel versions of this algorithm can be developed. One is to assign separate processors to the FGT-DAGs in refactoring Y. Then each processor will run the detection algorithm described in section 8.4 to search conflicts between FGTs in its FGT-DAG and FGTs in FGT-DAGs of the other refactorings. Note that this parallel version assumes that any detected conflict will be resolved in the FGT-DAG of refactoring Y.

C. In **sequential dependency algorithm**: As described in section 9.4, the sequential dependency algorithm takes each FGT from refactoring X and checks if it has a sequential dependency relation with at least one FGT in refactoring Y. Here parallelizing can be done at the level of each FGT. One of the most fine-grained parallel versions of this algorithm would be to have one processor per FGT-pair to be tested. A processor first checks if there is a match between its pair of FGTs and one of the **uniDirSD** facts. If so, it terminates and declares a sequential dependency. Otherwise it continues to check for a match with **biDirSD** facts, determines the direction if one is found, declares a sequential dependency and terminates. A centralised scheduler should receive results and direct all running processes to abort when the first sequential dependency is reported.

D. At **refactoring time**: As explained in section 4.4.4, applying refactoring (primitive or composite) on the system can be done in two phases:

- In the first phase, the tool checks refactoring's precondition conjuncts against the system. To do that it checks first the refactoring-level precondition conjuncts, and if they are satisfied it checks the FGT-enabling preconditions of the various FGT-DAGs in the refactoring. Checking the FGT-enabling preconditions can be executed in parallel by assigning a processor to each one of the various FGT-DAGs of that refactoring. Each

processor then will be responsible for checking the FGT-enabling precondition conjuncts of its FGT-DAG. If all the precondition conjuncts are satisfied in all processors then the tool goes to the second phase.

- In the second phase, the tool applies the FGTs of the refactoring under consideration. This stage also can be done in parallel by letting each processor apply the FGTs in its FGT-DAG to the system.

The foregoing describes in overview the potential for parallelizing at the FGT-DAG level. However, it should be noted that more fine-grained parallelization would also be possible within an FGT-DAG. In this case, FGTs on separate branches of the FGT-DAG could run in parallel pipelines with one another, but would then have to synchronize appropriately on FGT nodes at which there is more than one inbound arc.

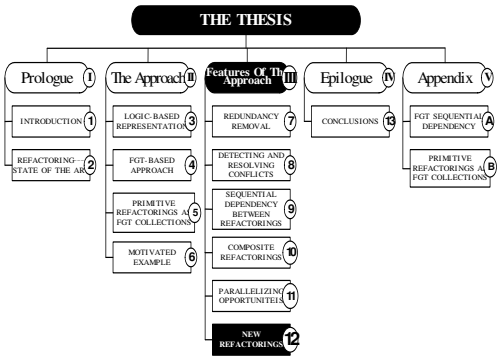
11.3 Reflection on Parallelization

Although some may question the relevance of parallelizing the refactoring task and associated algorithms in the contemporary world of refactoring, this would seem to be a rather short-sighted view. On the one hand, the matter of potential for parallelizing computational tasks has always been of theoretical interest in computer science. On the other hand, it is now widely acknowledged that current trends in chip design are in the direction of increasing the number of cores per chip. Indeed, in recent years, Intel and others have emphasized the importance of adapting computer science curricula to prepare students for a future in which parallel/concurrent programming will become ever-more dominant. This chapter has suggested that an FGT-based approach to refactoring seems well-adapted to such a future.

Chapter 12

NEW REFACTORINGS

12.1 Introduction



Kniesel and Koch [38] point to a dilema that confronts the developer of a refactoring tool. On the one hand, user needs are not limited to a core of custom refactorings that can be embedded into a tool. In fact, the type and complexity of refactorings needed varies according to areas of application and needs evolve over time. In this regard, they mention applications like "refactoring to patterns" [8, 37] and "refactoring to aspects" [30]. On the other hand, they note that tools lack user-definable refactorings. They point out that:

> *"[This] lack of user-definable refactorings is equally unsatisfactory for tool providers and for their users. For tool providers, because they must continuously invest time and money in the never-ending evolution of refactorings. For users, because they are forced either to wait for some future release, hoping that it will provide the missing functionality, or to implement their own custom refactorings. However, the latter is not a real option for most users. After all, most developers are interested in refactoring as a means of speeding their own development activities, not as an additional development task within an anyway much too tight schedule."*

One of the solutions for the above problem is for a refactoring tool to provide the end user with a facility for composing larger refactorings from primitive ones. This possibility was described in detail in chapter 10. Unfortunately, the provision of such a facility in a tool is not be sufficient. Providing a set of primitive refactorings that can be executed in sequence in order to achieve a complex effect, is not the same as providing users with the ability to define their own refactorings. Some cases may exist where the end user needs to build a new refactoring that cannot be constructed by just using the primitive refactorings that have been implemented in the refactoring tool.

12.2 Example

Return to the LAN motivated example presented in chapter 6. As before, suppose that one of the proposed enhancements to make to the class diagram shown in Figure 12.1, is to pull up the *accept* method from the subclasses *FileServer* and *PrintServer* to their superclass *Server*. The motivation for this refactoring is that the *accept* method in the two subclasses are identical, and it is preferred to pull it up to the common superclass for the reasons described in section 5.3.2.6.

Assume that, in contrast to chapter 6, the *accept* method accesses the *public* method, *process*, in the two subclasses. If the *accept* method is moved from the subclasses to the superclass, this access (to the *process* method) will not be visible from the superclass. In fact, at the code-level such a move would result in a "*process method is undefined*" compiler error[2].

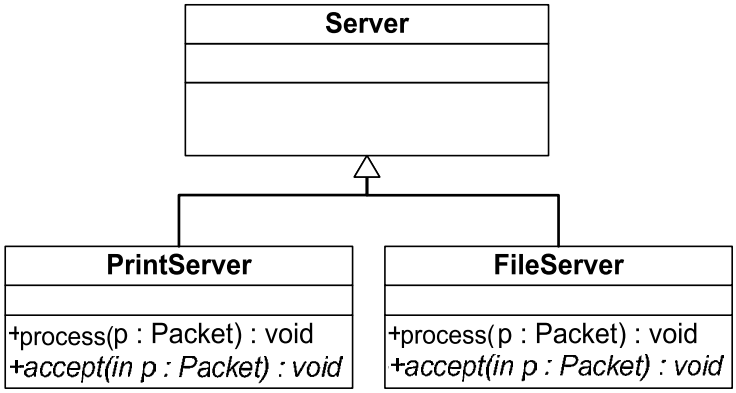


Figure 12.1: Part of the LAN system's class diagram

To avoid such problems, one of the precondition conjuncts of the refactoring **pullUpMethod** requires that all the references made by the pulled up method to the other object elements in the system must be visible from the superclass. According to this precondition, pulling up the method *accept* in the example will be rejected.

Suppose that the developer considers that this precondition is very restrictive. There are ways in which such a refactoring can be applied without affecting the behaviour of the system. With reference to the *accept* method in the present case, one solution is to define an empty method with name *process* in the superclass, which has the same signature as the *process* method in

[2] Note that the fact that the *process* method is *public* is incidental to the argument here. The argument would be the same if its access mode had been *protected* or *private*.

the subclasses. The result will be that the referenced made by the *accept* method in the superclass to the *process* method will be valid now.

Therefore, the idea is to define in the superclass all the methods in the subclasses that the pulled up method references in the subclasses. (It is recognised that a cleaner solution would be to define an abstract method in the superclass, but these have not been considered in the present work.)

Defining these methods in the superclass will not affect the behaviour of the system because the newly defined methods are not referenced by any other object elements. Also, they are empty and will be overridden by the original members defined in the subclasses.

Note that the above enhancement to the refactoring **pullUpMethod** is valid only if the access mode of the *process* method in the subclasses is *public* or *protected*—i.e. it may not be *private*.

Suppose that the end user wants to create a new refactoring that takes into consideration the above enhancement to the **pullUpMethod** refactoring. The new refactoring should work as follows:

a. The set of methods in the subclasses referenced by at least one subclass version of the method to be pulled up should be noted. Each element of this set should have the same signature in the subclasses in which it occurs. No element of this set should have a *private* access mode in any of the subclasses. For each of these methods, an empty method with the same signature as defined in the subclasses should be added into the superclass. The access mode of each method should not be more general than the access modes of the corresponding versions of the method in the various subclasses—i.e. it should be *protected* if it is *protected* in one or more subclasses, and otherwise (if it is *public* in all subclasses) it should be *public*.

b. The set of attributes in the subclasses referenced by at least one subclass version of the method to be pulled up should be noted. No element of this set should have a *private* access mode in any of the subclasses. The access mode of each attribute should not be more general than the access modes of the corresponding versions of the attribute in the various subclasses—i.e. it should be *protected* if it is *protected* in one or more subclasses, and otherwise (if it is *public* in all subclasses) it should be *public*.

It is clear that it is impossible to accomplish this by trying to compose a sequence (collection) of the primitive refactorings presented in Table 4.1, which means that the approach of building

a composite refactoring presented in chapter 10 will not work here. A new approach is required.

12.3 New Refactorings in the FGT-Based Approach

A refactoring tool based on FGTs can, in principle, enabled users to build their own refactorings without needing to write code. Instead, they would rely on the set of the low-level FGTs proposed in the thesis, as well as various algorithms discussed in earlier chapters. In outline, what is needed is a small domain specific language (DSL) whose semantics allows for (a) selection from the tool's set of FGTs; (b) simple conditional and looping statements; (c) specification of a new refactoring's refactoring-level preconditions and (d) simple storage and retrieval of named and parameterised procedures. The body of the procedure would then consist of a sequence of FGTs, some of which may need to be conditionally included, depending on the system eventually to be refactored. The new refactoring can be saved as a named procedure with a list of input parameters. "Compilation" of the procedure would involve instantiating the refactoring into an FGT-list for a given system, decomposing the list into a set of FGT-DAGs, reducing FGTs where required, identifying and possibly resolving conflicts, and computing the refactorings FGT-enabling precondition for that system. Although it is beyond the scope of this thesis to develop such a DSL, its successful implementation would go a long way to resolving the refactoring dilemma referred to by Kniesel and Koch and cited above.

In the absence of such a DSL, the in which a new FGT-based can be implemented will now be illustrated. The refactoring

enh-pullUpMethod(*SubClassesNames, Methn, MethTList* **)**

takes into consideration the enhancement proposed in section 12.2, and consists of the following sequence of FGTs:

1. addObject(*SupPn, SupCn, Methn, _, _, MethRType, OAMode, MethTList, method*)

2. **For** each relational element between *Methn* as a source and *Methx* as destination where the access mode of *Methx* is *public* or *protected* and *Methx* is defined in the same class as *Methn* **do** {
   addObject(*SupPn, SupCn, Methx, _, _, MethxRType, MOAMode, MethxTList, method*) }

3. **For** each relational element between *Methn* as a source and *Attx* as destination where the access mode of *Attx* is *public* or *protected* and *Attx* is defined in the same class of *Methn* **do** { addObject(*SupPn, SupCn, Attx, _, _,AttxDType, AAMode, _,attribute*) }

4. **For** each subclass in the *SubClassesNames* list **do** {

   deleteObject($SubPn_i$,$SubCn_i$, *Methn,_, MethTList, method*) }

The differences between the new refactoring **enh-pullUpMethod** and the refactoring **pullUpMethod** presented in section 5.3.2.6 are in steps 2 and 3. These are not found in the **pullUpMethod**. In step 2, the method members added to superclass are all those methods defined in the subclasses *SubClassesNames*, referenced by the *Methn*, and their access mode is *public* or *protected*. Similar, checks are done in step 3 with respect to attribute members. Note that the value of the arguments *MOAMode* in step 2 and *AAMode* in step 3 are calculated according to the rule described in section 12.2.

After applying the new refactoring

enh-pullUpMethod(*['FileServer', 'PrintServer'], accept, ['Packet']* **)**

to the class diagram shown in Figure 12.1, the class diagram will be restructured as shown in Figure 12.2.

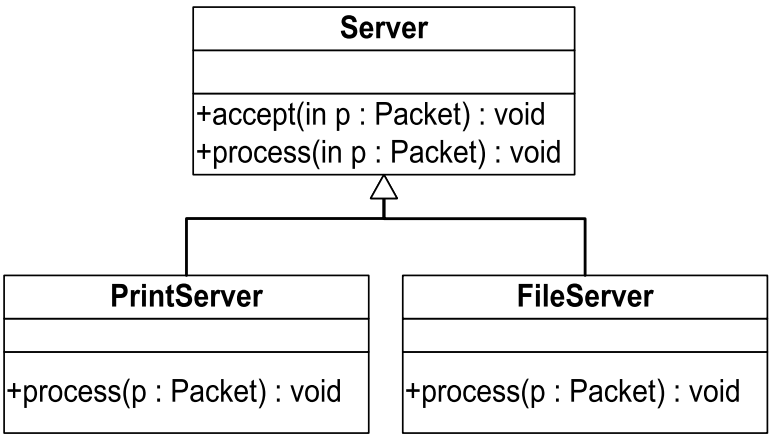


Figure 12.2: Part of the LAN system's class diagram after enh-pullUpMethod

12.4 Reflection on this Chapter

Giving end users the ability to use FGTs to construct their own refactorings has the following advantages:

1. The user is not restricted to use the list of primitive refactorings implemented in the tool for building other refactorings. Instead, a much wider variety of refactorings can be built, due to the more comprehensive semantics of FGTs.

2. Because pre- and postconditions of FGTs can be stored in the refactoring tool, the user is absolved from articulating them again when creating new refactorings. In fact, after the desired sequence of FGTs has been found, FGT-enabling preconditions for the various FGT-DAGs of the new refactoring can be automatically computed. Note, however, that articulating refactoring-level preconditions of the refactoring remains the responsibility of the user.

3. As mentioned above, there is no need from the user to write a pure code.

4. Because the new refactoring will be built as a collection of FGTs, all features presented in the thesis for such representation can be applied to the new refactorings.