# Formal Specification and Verification of UML Class Diagram Refactorings - *Based on FGT Paradigm*

**Emad Sa'adeh**

An-Najah National University, Computerized Information System Department
Email: esaadeh@najah.edu

## ABSTRACT

Refactoring UML class diagrams for evolution are usually carried out in an ad hoc way. These transformations can become an issue since it is hard to ensure that the semantics of models is preserved. Our work in this paper explores the use of the so-called Fine-Grained Transformations (FGTs) paradigm as a formal specification and verification of UML class diagram refactoring. More precisely, the paper expresses UML class diagram restructurings in terms of atomic FGTs, which are considered to be the core of a refactoring system. The paper presents the feasibility of building traditional class diagrams refactoring (primitive and composite) from sequences of FGTs in a way that improves the structure and preserves the original behavior (semantic) of the class diagram. Besides the obvious benefits of providing rigorous specifications for refactoring tool builders and rigorous correctness guarantees, the paper presents many additional advantages and features of the approach. For testing, a refactoring tool FGTRefClass is implemented. Our experience shows that the tool facilitates the process to restructure class diagram models.

**Keywords**

Class diagram refactorings, fine-grain transformations, conflicts, sequential dependencies, redundancies

*Article Received: 10 August 2020, Revised: 25 October 2020, Accepted: 18 November 2020*

## Introduction

Refactorings have gained wide attention in software evolution community. The idea was first formalized in the work of Opdyke [18] and described in depth by Fowler [6]. Refactorings are techniques to improve the internal structure of the software while preserving its external behaviour [6, 17-18, 20-21].

The current trend is to apply refactorings at levels of abstraction above the code level [1, 19 and 23]. This is because many people are visually oriented and prefer to visualize the relationships between classes rather than apprehend them textually. Furthermore, being able to directly manipulate code at a higher level of granularity (i.e. methods, variables, and classes rather than characters) can make refactoring more efficient [1]. In line with this trend, this paper concentrates on refactoring of UML class diagrams.

Several approaches have been used to formalize refactorings. For example, the graph transformation approach [3, 4 and 5] represents software as a graph, and refactorings are formalized as graph production rules [2, 7, 10-14]. As another approach, the logic based conditional transformation approach [8, 9] represents software as logic terms and refactorings are formalized as conditional transformation with pre- and post-conditions.

In general, these approaches represent a refactoring (primitive and composite) in a refactoring tool as a sequence of code to transform the target system, with a set of preconditions that must be satisfied in order to apply that refactoring, as illustrated in Figure 1. There is no transparency of the detailed steps internally required during refactoring.

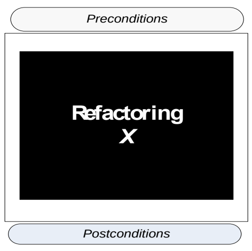

**Figure 1.** Refactorings as black box

Treating refactoring as a black box is the source of several problems and shortcomings in refactoring tools:

1. Where redundancy inside or between refactoring may exist, there is no possibility to remove it, which implies that effort spent on checking preconditions and on executing the transformation is wasted.
2. Where conflict occurs between two refactorings, there is no possibility to know

which part of the two refactorings caused the conflict. This makes the process of resolving the conflicts more difficult.

3. Where there is a sequential dependency between two refactorings, there is no possibility to know at what specific point one of the two refactorings is sequentially dependent on the other.
4. Because refactorings are considered as a piece of code, it is difficult to parallelize the resulting transformations that need to be applied on the model.
5. Because the list of possible refactorings is unbounded, no tool vender can provide end users with all their needs. Instead, refactoring tools providers need to give end users the ability to create their own refactorings. This is difficult in the current approaches, because it requires that the end user should write code *ab initio* to perform the refactoring.

In order to address problems such as these, our previous work, described in [22] uses so-called Fine-Grained Transformations (FGTs). It defines these FGTs and constructs/executes the different refactorings in reference to them. The main contribution of this paper is to extend our work in [22] by presents the feasibility of the FGT paradigm to formalize class diagram refactorings, and presents the features of using such an approach.

## FGTs-Based Approach

The refactoring approach described in [22] is based on a predefined set of fine-grain transformations (FGTs) which are the basis for the construction of model transformations in general. These FGTs are derived from the general actions that can be performed on elements of a model. From a formal point of view, these FGTs are sufficient to generate any kind of transformation to a given model. Therefore any refactoring can be constructed by using a sequence of these FGTs.

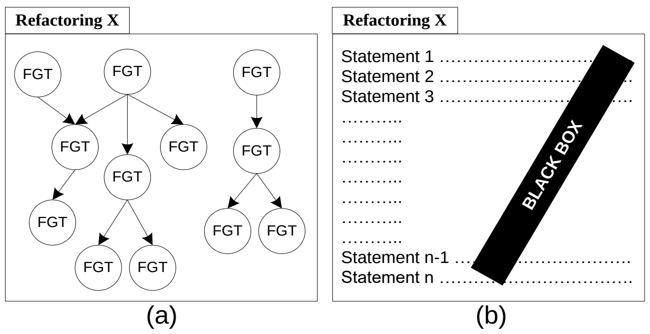
Refactoring X
FGT
FGT
FGT
FGT
FGT
FGT
FGT
FGT
FGT
FGT
FGT
FGT
FGT
(a)
Refactoring X
Statement 1 ............................
Statement 2 ............................
Statement 3 ............................
...........
...........
...........
...........
...........
...........
...........
Statement n-1 ............................
Statement n ............................
BLACK BOX
(b)

**Figure 2.** Refactoring different considerations

As shown in Figure 2.b, in previous approaches to building refactoring tools, a set of refactorings is mapped to a sequence of code that, when executed, translates the model to an equivalent model. This sequence of code may include any kind of statements. No meta-information about what each part of the code does is available to the refactoring tool, and consequently, the tool has no ability to control or manipulate any part of the code, other than to execute it. In this sense, the tools in these approaches treat refactoring as a black box with a set of preconditions that need to be satisfied before applying that refactoring. On the other hand, as shown in Figure 2.a, a set of refactorings in the FGT approach is set of directed acyclic graphs (FGT-DAGs), each of which specifies an ordering of FGTs to be used in the refactoring. The order, effect, preconditions and post-conditions of each FGT in each FGT-DAG is known to the tool, and can be controlled at the time of refactoring.

### *Fine-Grain Transformations (FGTs)*

An FGT is an abstract operation on the model—i.e. a model will always be one of the implicit operands of an FGT, and this model will always undergo an incremental atomic change as a result of applying the FGT to it. Indeed, the change can be regarded as atomic in the sense that the change specified by the FGT cannot be broken down into further smaller change steps from the modeling perspective. The operation is abstract in the sense that it could be specified in a wide variety of concrete syntactic representations.

As a proof of concept, a PROLOG prototype for class diagram refactoring tool has been implemented. Throughout the paper, a concrete syntax that resembles PROLOG rules (also called procedures) will be used to specify FGTs. This choice of concrete syntax was made to support the PROLOG prototype refactoring tool that has been built to illustrate the various ideas. The UML class diagram in the tool is itself stored as a set of facts in the PROLOG database. As will be seen below, the concrete syntax of each FGT has to uniquely identify the various components of the class diagram that are to change, and it also has to indicate the nature of the change. In general, the nature of the change is encapsulated in the name of the PROLOG rule, and the class diagram components that are affected are specified as arguments of the rule.

The set of FGTs that have been identified are closely related to the vocabulary and semantics of standard UML class diagram. The vocabulary of UML class diagram consists of a set of Objects (packages, classes, attributes, methods and parameters) to represent discrete concepts in software systems. The vocabulary also contains a set of Relations (extends, associations, reads, writes, calls, types) to relate the objects to one another.

The set of FGTs proposed here are accordingly classified into two groups, where each group corresponds to one of the two specific kinds of UML class diagram vocabularies. The first group is concerned with all the transformation operations whose characterizing operands are object elements of the UML class diagram. In the rest of the paper, these FGTs are called Object Element FGTs. FGTs of this group are:

- addObject FGT: used to add *object* elements to the class diagram
- renameObject FGT: used to change the name of an *object* element
- changeOAMode FGT: used to change the access mode of an *object* element
- changeODefType FGT: used to change the definition type of an *object* element
- deleteObject: used to delete *object* element from the class diagram

## *FGT Precondition Conjuncts*

Each FGT of the two groups has a set of precondition conjuncts (i.e. X and Y and Z and …) that need to be satisfied in order to consider it as a legal transformation operation. In some cases, one or more of these conjuncts is itself a number of disjuncts (i.e. (X or Y)). A procedure called FGTPrecondConj(*FGT*) is implemented in the refactoring tool for each one of the proposed FGTs. FGTs precondition conjuncts will play an important role in preserving the behaviour of the system at the time of refactoring. For example, in order to apply the FGT:

addObject(College,Student,getMark,_,_,int,1, public,[],method)

the underlying system must have a class with name *Student* inside the package *College*; and this class should not contain a method *getMark* with empty parameter list. The method *getMark* with empty parameter list should also not inherited from any of the ancestors of class *College.Student*. In addition, the return definition type of the method should be valid and accessible and the access mode of the created method should be valid. The precondition conjuncts for this FGT, as implemented in our tool, are specified as follows:

FGTPrecondConj(addObject(Pn,Cn,Methn,_,_,ODefT,ONum,OAMode,

PrmLT,method)):-

existsObject(*Pn, Cn, class*), not(existsObject(*Pn, Cn, Methn, PrmLT, method*),not(isInherited(*Pn, Cn, Methn, PrmLT, method*),validDefType(*ODefT*), canAccessType(ODefT),

validOAMode(OAMode,method).

Note that the comma (,) between two conjuncts retains the PROLOG semantics of a "logical and" between two rules. As another example, in order to apply the FGT:

addRelation(e1,College,Student,getMark,_,[], method,College, Student, Mark,_,_,attribute,read)

the method *College.Student.getMark* with empty parameter list and the attribute *College.Student.Mark* should be defined in the underlying system. The system may not already have a read access between the method *College.Student.getMark* and the attribute *College.Student.Mark*. In addition, the location of the source object *College.Student.getMark* and the destination object *College.Student.Mark* in the model together with the access mode of the destination object *College.Student.Mark* play an important role in determining the applicability of the previous addRelation FGT. The precondition conjuncts for this FGT, as implemented in our tool, are specified as follows:

FGTPrecondConj(addRelation(_,FPn,FCn,FMethn,_,FPrmLT,method,TPn,TCn,TAttn,_,_,attribute, RelT)):-

existsObject(FPn,FCn,FMethn,FPrmLT,method), existsObject(TPn,TCn,TAttn,attribute),not(existRelation(_,FPn,FCn,FMethn,FPrmLT,

method,TPn,TCn,TAttn,attribute,RelT)
,[(objectAMode(TPn,TCn,TAttn,attribute,private),
FPn.FCn=TPn.TCn)|
(objectAMode(TPn,TCn,TAttn,attribute,default ),

FPn=TPn)|
objectAMode(TPn,TCn,TAttn,attribute,protected),
(subClass(FPn,FCn, TPn, TCn)|FPn=TPn))|
objectAMode(TPn,TCn,TAttn,attribute,public)].

Note that in the above rule, the vertical bar (|) between two conjuncts retains the PROLOG semantics of "logical or" between two rules.

### *FGT Directed Acyclic Graphs (FGT-DAGs)*

Sequential dependency between two FGTs, FGTi and FGTj occurs when the FGTj is not applicable (its set of precondition conjuncts are not satisfied) and there is a consequent need to first apply FGTi on the system to modify the state of the system so that will FGTj will indeed be applicable (its set of precondition conjuncts will be satisfied). In this case we say that FGTj is sequentially dependent on FGTi. We represent the sequential dependency between the two FGTs as: FGTi □ FGTj

For example, the FGT

addObject('P','A', m1,_,_,void, 0,public,[],method)

that is used to add the method m1 inside the class P.A is sequentially dependent on the FGT

addObject('P','A',_,_,_,_,_,public,_, class)

that is used to add the class A inside the package P, because one first has to add the container (class P.A) and before adding members inside it. The sequential dependency between the two FGTs is represented as:

addObject('P','A',_,_,_,_,_,public,_,class) → addObject('P','A',m1,_,_,void,0,public,[], method)

In the proposed approach, the sequence of FGTs that represent a specific refactoring are inserted into one or more special data structures called an FGT directed acyclic graph (FGT-DAGs). Each node in the FGT-DAG represents one of the FGTs. These FGTs are ordered in the FGT-DAG according to their sequential dependencies. All the possible sequential dependencies between all the FGTs that are used in the approach have been catalogued. There is no dependency between the different FGT-DAGs of a refactoring. As a result they can be processed concurrently, thus increasing the scope for parallelizing the refactoring operations to be carried out on the class diagram.

## FGTs for Primitive and Composite Refactorings

Refactoring theory and tools assume that there exists a finite set of primitive refactorings [12, 18 and 20]. A primitive refactoring is an atomic refactoring that cannot be split into more refactorings. For each primitive refactoring, a set of preconditions exists that will guarantee behaviour preservation of the system under consideration. These preconditions are implemented inside the refactoring tool and need to be checked before applying the related refactoring.

A composite refactoring is a sequence of primitive refactorings that need to be applied on the model as one unit. This means that either all of the primitive refactorings that constitute the composite refactoring will be applied on the model; or, if there is one of the preconditions of one of the including primitive refactorings is not satisfied, then none of the primitive refactorings will be applied. A composite refactoring can be applied to the system if all its constituent primitive refactorings can be applied to the system. Because a composite refactoring consists of primitive ones that preserve system behaviour, the composite will also preserve the behaviour of the system.

Our work in this paper shifts the granularity of refactoring one level down: primitive refactorings are constructed from a sequence of FGTs. The relationship between primitive refactorings, composite refactorings and FGTs is intuitively reflected in Figure 3.b. Figure 3.a shows that a composite refactoring is a sequence of primitive ones, and each primitive refactoring can be defined as a sequence of FGTs. Thus, each composite refactoring can be carried out as a sequence of FGTs.

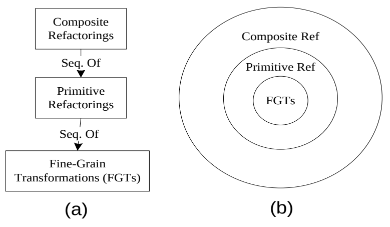

**Figure 3.** Primitive, composite refactorings and FGTs.

Note that in most of the cases the precondition conjuncts of the FGTs that the primitive consists of are sufficient to cover the preconditions of that primitive, so they are enough to ensure the behaviour preservation of the system. However, in some cases the precondition conjuncts of the FGTs are alone not sufficient to preserve the behaviour of the system. Behviour preservation is only guaranteed if all the preconditions of the primitive refactoring are satisfied. In recognition of this fact, and to keep our approach general and thus leave the door open to define new refactorings in the future we define -as shown in Figure 4.a- the set of preconditions for each primitive refactoring at two different levels:

a. FGT-Level Preconditions: The set of precondition conjuncts that are defined at the level of FGTs.

b. Refactoring-Level Preconditions: The set of precondition conjuncts that are defined at the level of the whole refactoring. This set contains preconditions that are not covered by (cannot be extracted from) the precondition conjuncts of the FGTs from which the refactoring is constructed.

In the present text, the focus is on preconditions. However, post-conditions can also be viewed as being at the refactoring-level as well as at the FGT-level. These notions are abstractly portrayed in Figures 4.a and 4.b with respect to our approach and previous approaches respectively.

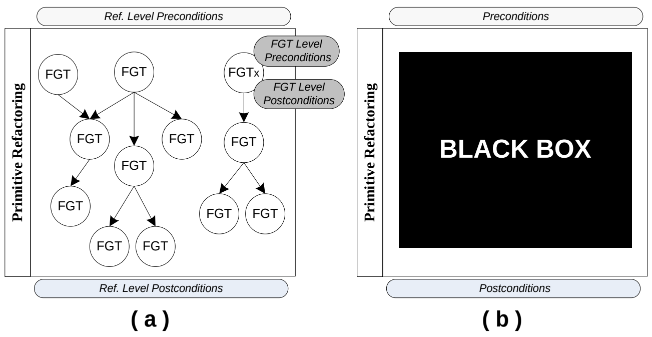

**Figure 4.** Primitive refactoring different considerations

## Motivated Example

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

Figure 5.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 5.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 we use the composite refactoring *encapsulateAttribute*. The composite *encapsulateAttribute* includes the following actions:

- Add getter and setter methods. This is done by using the primitive refactorings *addGetter* and *addSetter*.
- Replace accesses to the attribute by calls to the newly created methods. This is done by using the primitive refactorings *attributeReadToMethodCall* and *attributeWriteToMethodCall*.
- Make the attribute private. This is done by using the primitive refactoring *changeAttributeAccess*.

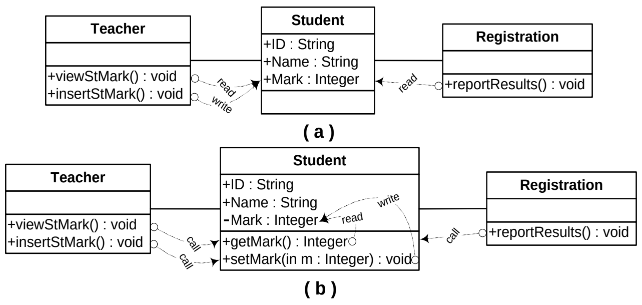

**Figure 5.** A simplified UML class diagram of a *college* system. (a) before refactoring and (b) after refactoring

The order of the primitive refactorings inside the composite is shown in Figure 6. Note that the order reflects the sequential dependency that exist between the different refactorings inside 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. The refactoring tool can therefore change the access mode of the attribute from public to private.

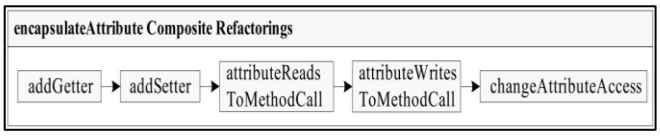
encapsulateAttribute Composite Refactorings
addGetter
addSetter
attributeReads ToMethodCall
attributeWrites ToMethodCall
changeAttributeAccess

**Figure 6.** encapsulateAttribute composite refactoring

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

encapsulateAttribute('College','Student', 'Mark')

where the three arguments in the procedure refer to the name of the attribute *College.Student.Mark* to be encapsulated. For each one of the primitive refactorings that are included in the composite *encapsulateAttribute* (shown in Figure 6, and also in the left column of Table 1) the *encapsulateAttribute* procedure will produce a sequence of FGTs which represents the transformation actions to be performed as part of the encapsulation process. These FGTs are shown in the right column of the table.

**Table 1.** encapsulateAttribute Refactoring

| **Sequence Of Primitive Refactorings** | **Sequence Of FGTs For Each Primitive Refactoring** |
|---|---|
| addGetter('College', 'Student', 'Mark') | FGT1:addObject(College,Student, getMark,_,_,int,1,public,[],method)<br>FGT2:addRelation(_,College,Student,getMark,_,[],method,College,Student,Mark,_,_, attribute, read) |
| addSetter('College', 'Student', 'Mark') | FGT3:addObject(College,Student,setMark,_,_,void,0,public,[(p,(int,1))],method)<br>FGT4:addRelation(_,College,Student,setMark,_,[int],method,College, Student, Mark,_,_, attribute,write) |
| attributeReadToMethodCall('College', 'Student', 'Mark', 'College', 'Student',getMark, []) | FGT5:deleteRelation(_,College,Teacher,viewStMark,_,[],method, College, Student, Mark,_,_, attribute,read)<br>FGT6:deleteRelation(_,College,Registration, reportResults,_,[],method, College,Student,Mark,_,_,attribute, read)<br>FGT7:addRelation(_,College,Teacher, viewStMark,_,[],method,College,Student,getMark,_,[],method,call)<br>FGT8:addRelation(_,College,Registration,reportResults,_,[],method,College,Student,getMark,_,[],method, call) |
| attributeWriteToMethodCall( 'College', 'Student', 'Mark', 'College', 'Student', setMark, [int]) | FGT9:deleteRelation(_,College,Teacher,insertStMark,_,[],method,College,Student, Mark,_,_,attribute,write)<br>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) |

For example, in the primitive refactoring *attributeReadToMethodCall* that has the following format:

attributeReadToMethodCall(Destx, Desty)

any read access from anywhere in the system to the destination *Dest*$_x$ will be redirected to a new destination *Dest*$_y$. This means that for each *read* access, two FGT operations will be produced, one to delete the original read access "*read relation*" from the source *S* to the destination *Dest*$_x$, this is done by FGT:

deleteRelation(_, S, Destx, read)

and the other to add a new *read* access from the source *S* to the new distention *Dest*$_y$, this is done by FGT:

addRelation(_, S, Desty, read)

In the *College* system, the attribute *Student.Mark* has two *read* accesses: one from the method *Teacher.viewStMark*; and the other from the method *Registration.reportResults*. This means that four FGTs will be produced by this refactoring: FGT5, FGT6, FGT7 and FGT8 as shown in right column of Table 1.

FGTs produced by each primitive refactoring in the composite are then allocated to one or more FGT-DAGs; each according to its specific sequential dependencies. Thus, sequential dependencies between the different FGTs in the different primitive refactorings have to be found. After that the reduction algorithm is executed on these FGT-DAGs to remove any redundancies between the different FGTs. At the end of these actions, the composite refactoring *encapsulateAttribute* will be represented as shown in Figure 7.

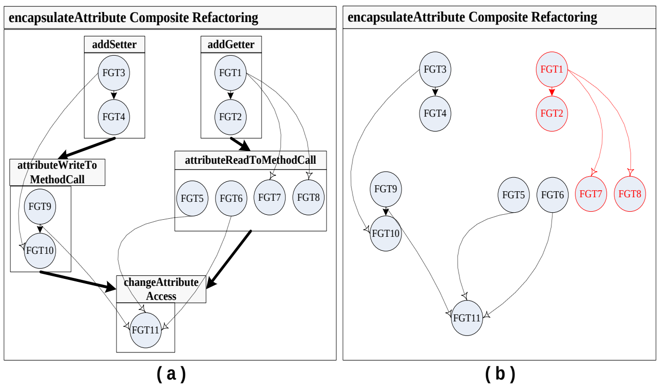

**Figure 7.** encapsulateAttribute Composite Refactoring as represented in our approach

Figure 7.a shows from what primitive refactorings each FGT comes from. Figure 7.b shows the sequential dependencies between these FGTs. Note that as shown in Figure 7.b the eleventh FGTs that are generated by the composite *encapsulateAttribute* are distributed among two sequentially independent FGT-DAGs which gives the possibility to apply the two FGT-DAGs concurrently to the system.

## IV. Features of the FGT approach

In this section we discuss the feature of using the FGT paradigm as a formalization of class diagram refactorings.

### *1. Remove Redundancies*

One of the advantages of dealing with refactoring as a sequence of FGTs is the ability to remove redundancies between sequences of FGTs. We call this process a reduction process. The final effect of the sequence of FGTs on the class diagram after the reduction process is the same as the effect of the sequence without any reduction. Two types of FGT reductions can be identified:

*Absorb Reduction:* This occurs when two FGTs are absorbed by one that has the same effect of the two. For example, suppose that the user wants to add new method *m*$_x$ inside class *P.A*. To do this he uses the FGT:

addObject('P','A',mx,_,_,void,0,public,[],method)

After that the same user or another one decides to rename method *m*$_x$ in class *P.A* to another name *m*$_y$. To do this he uses FGT:

renameObject('P','A', mx,_,[],method,my)

By the reduction process the two operations will be absorbed into the single FGT:

addObject('P','A',my,_,_,void,0,public,[],method )

*Cancel Reduction:* This occurs when two FGTs cancel each other. For example, suppose that the user wants to add new method *m*$_x$ inside class *P.A*. To do this he uses FGT:

addObject('P','A',mx,_,_,void,0,public,[],method)

After that the same user or another one for some reason decides to delete the method *m*$_x$ from class *P.A*. To do this he uses FGT:

deleteObject('P','A', mx,_,[],method)

By the reduction process, the two operations will be removed from the FGT sequence. In general, the reduction process increases the efficiency of the refactoring algorithm by reducing the number of FGTs that need to be applied on the model.

### *2. Detect & Resolve Conflicts*

Conflicts between multiple refactorings can be managed at the level of FGTs rather than at the level of the whole refactorings. Conflict between $FGT_i$ and $FGT_j$ occurs when it is the case that applying them in any order will make the later one inapplicable. For example,

addObject('P','A',mx,_,_,void,0,public,[],method)

and

renameObject('P','A', my,_,[],method, mx)

are in conflict (mutually exclusive), because applying any one will prohibit applying the other. The tool can discover the first occurrence of conflict between the two refactorings which gives the ability to resolve this conflict later.

### *3. Find Sequential dependencies*

Sequential dependencies between multiple refactorings can be managed at the level of FGTs rather than at the level of the whole refactoring. Sequential dependency between two FGTs, $FGT_i$ and $FGT_j$ ($FGT_j \rightarrow FGT_i$) occurs when $FGT_i$ is not applicable (its preconditions are not satisfied) but if $FGT_j$ is applied first, then $FGT_i$ will become applicable. In this case we say that $FGT_i$ is sequentially dependent on $FGT_j$. For example,

renameObject('P','A',mx,_,[],method,my)

sequentially depends on

addObject('P','A',mx,_,_,void,0,public,[],method)

because if the method $m_x$ is not in class *P.A*, then one first has to add it to the class *P.A*, before attempting to rename it. The tool can discover at what specific point or points the two refactorings are sequentially dependent.

### *4 Increasing Parallelizing opportunities*

Parallelizing opportunities in our approach are automatically manifested at the time of refactoring or during the process of detecting conflicts, removing redundancies and finding sequential dependencies between refactorings. This is basically because the FGTs for a refactoring may be assigned to one of multiple FGT-DAGs, depending on the sequentially dependency between these FGTs. These FGT-DAGs are independent and can be managed concurrently.

### *5. Build new Refactorings*

In a refactoring tool based on FGTs, end users will be able to build their own refactorings without a need to write code. This is a very important feature because the list of possible refactorings is undetermined, and no tool vendor can support the end users with all their needs. Our approach solves this shortage by giving the end users the ability to construct new refactorings by using the set of the low level FGTs. To create a new refactoring: the end user need just to select the sequence of FGTs needed to construct his refactoring. The new refactoring will be given a name, list of input parameters, and can be saved in the refactoring tool for a later use.

## FGTRefClass TOOL: Evaluation and Testing

Because of its overall suitability for prototyping, it was decided to build our refactoring tool FGTRefClass based on Prolog to experiment with FGT-based refactoring concepts. This decision was partially inspired by the JTRANSFORMER tool described in [13], which represents Java code as Prolog facts, and executes refactoring by manipulating these facts. The decision also means that many of the explanations relating to FGT-based refactoring can be given by referring to the Prolog facts (logic-terms) that have been used as data for the tool.

Fig 8 describes the architecture of our class diagram refactoring tool. The tool takes the XML document of the class diagram as input. Then it extracts the Prolog facts (or logic-terms) from the XML document. The vocabulary extracted is limited to a set of facts to represent commonly referenced objects (i.e. packages, classes, attributes, methods, and parameters) and relations (i.e. extensions, associations, reads, writes, calls, types) within or between the object elements in UML class diagrams.

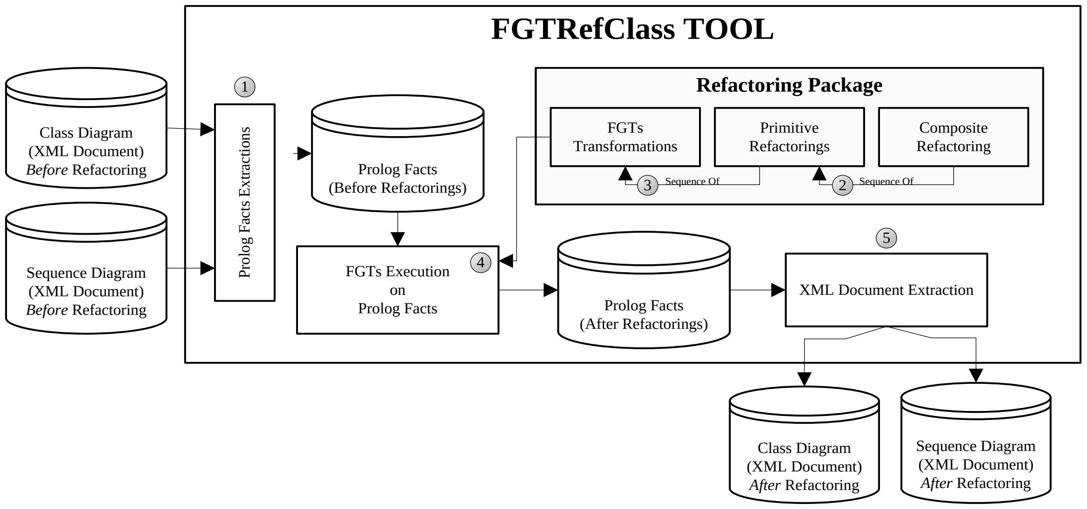
FGTRefClass TOOL
Class Diagram (XML Document) Before Refactoring
Sequence Diagram (XML Document) Before Refactoring
1
Prolog Facts Extractions
Prolog Facts (Before Refactorings)
FGTs Execution on Prolog Facts
4
Refactoring Package
FGTs Transformations
Primitive Refactorings
Composite Refactoring
3 Sequence Of
2 Sequence Of
Prolog Facts (After Refactorings)
5
XML Document Extraction
Class Diagram (XML Document) After Refactoring
Sequence Diagram (XML Document) After Refactoring

**Figure 8.** Architecture of our **FGTRefClass** TOOL

However, it turns out that the conventionally available UML class diagram information is inadequate for implementing the full range of refactorings mentioned in the literature. Some refactorings require, in addition, access information—i.e. information that shows call relationships between methods and read or write relationships between methods and attributes. This need for augmenting UML class diagram information with additional access information was also recognized in the graph-based approach to refactoring, pioneered by Mens [14]. In FGTRefClass tool, access information is gotten from the sequence diagrams.

When the developer asks the tool to apply a specific refactoring on the stored Prolog facts, then the ***Refactoring Package Module*** works here. The module will extract all the sequences of FGTs for that refactoring. These FGTs will be applied one after another on the system. The output of this process will be in the form of XML document of the restructured class diagram.

## Conclusions

This paper investigated the feasibility of using FGT paradigm as a formal specification of UML class diagram refactorings. The approach defines and executes class diagram refactorings as a set of FGTs and manages sequential dependencies, conflicts and redundancies at the level of these low level operations. In addition, it gives the end users the ability to build new refactorings without having to write a code.

## References

[1] Astels, D. (2002), Refactoring with UML, In: Proc. Int'l Conf. eXtreme Programming and Flexible Processes in Software Engineering (pp. 67-70).

[2] Bottoni, P., Parisi-Presicce, F., & Taentzer, G. (2002). Coordinated distributed diagram transformation for software evolution, Electronic Notes in Theoretical Computer Science 72(4).

[3] Cuny, J., Ehrig, H., Engels, G., & Rozenberg, G. (1996). editor "Graph Grammars and Their Application to Computer Science," Lecture Notes in Computer Science 1073, Springer-Verlag.

[4] Engels, G., Hartmut, E., & Rozenberg, G. (1996). editors, "Special Issue on Graph Transformations", Fundamenta Informaticae 26 (3,4), IOS Press.

[5] Ehrig, H., Engels, G., Kreowski, J., & Rozenberg, G. (2000). editors, "Theory and Application to Graph Transformations," Lecture Notes in Computer Science 1764,Springer-Verlag.

[6] Fowler, M. (1999), Refactoring: Improving the Design of Existing Code. Addison-Wesley.

[7] Jahnke, J., & Zundorf, A. (1997). Rewriting poor design patterns by good design patterns, in: S. Demeyer and H. Gall,editors, Proc. of ESEC/FSE '97 Workshop on Object-Oriented Reengineering, Technical University of Vienna,Technical Report TUV-1841-97-10.

[8] Kniesel, G. (2006), A logic foundation for conditional program transformations.

Technical report no IAI-TR-2006-01,ISSN 0944-8535,CS Dept III.

[9] Kniesel, G. & Koch, H. (2004). Static composition of refactorings. Science of Computer Programming, 52:9-51.

[10] Mens, T. (1999), A formal foundation for object-oriented software evolution. PhD thesis, *Vrije Universiteit Brussel*.

[11] Mens, T. (2005), "On the use of graph transformations for model refactoring," in Generative and transformational techniques in software engineering (J. V. Ralf Lämmel,Joao Saraivaed.), pp. 67–98, Departamento di Informatica,Universidade do Minho.

[12] Mens, T., Van Eetvelde, N., Demeyer, S., & Janssens, D. (2005). Formalizing refactorings with graph transformations. *Journal on Software Maintenance and Evolution* 17(4), 247–276, Wiley.

[13] Mens, T., Van Gorp, P., ,Varró, D., & Karsai, G. (2005). "Applying a model transformation taxonomy to graph transformation technology," in Proc. *Int'l Workshop on Graph and Model Transformation* (GraMoT2005).

[14] Mens, T., Demeyer, S., & Janssens, D. (2002). "Formalising behaviour preserving program transformations," in Graph Transformation, *Lecture Notes in Computer Science*, vol. 2505, pp. 286-301, Springer-Verlag.

[15] Mens, T. & Tourwe', T. (2004). A survey of software refactoring. *IEEE Transactions on Software Engineering*, vol. 30 n.2, p. 126-139.

[16] Object Management Group (2005), "Unified Modeling Language: *Infrastructure version 2.0." formal/2005-07-05.*

[17] Opdyke, W., & Johnson R. (1993). Creating abstract superclasses by refactoring. *Proceedings ACM Computer Science Conference. ACM Press*, pp. 66-73.

[18] Opdyke, W. (1992), *Refactoring object-oriented frameworks*. Ph.D. thesis. University of Illinois at Urbana-Champaign.

[19] Porres, I. (2003), Model refactorings as rule-based update transformations. *Proceedings of UML 2003 Conference*, pages 159-174.

[20] Roberts, D. (1999), *Practical Analysis for Refactoring*. PhD thesis, University of Illinois at Urbana-Champaign

[21] Roberts, D., Brant, J., & Johnson, R. (1997). A refactoring tool for smalltalk. *Theory and Practice of Object Systems*, vol. 3, no. 4, pp. 253-263.

[22] Saadeh, E., Derrick G. (2013). *Refactoring With Ordered Collections Of Fine-Grain Transformations.* International Journal of Software Engineering and Knowledge Engineering. Volume 23, Issue 03, pp. 309-339, April 2013. DOI: 10.1142/S0218194013500095.

[23] Sunyé, G., Pollet, D., Le Traon, Y., & Jézéquel, J.-M. (2001). Refactoring UML models, In: *Proc. Int'l Conf. Unified Modeling Language* (pp. 134-138), LNCS 2185, Springer.