REFACTORING WITH ORDERED COLLECTIONS OF FINE-GRAIN TRANSFORMATIONS

EMMAD SAADEH

Computerized Information Systems
An-Najah National University, Nablus, Palestine
esaadeh@najah.edu

DERRICK G. KOURIE

Espresso Research Group
University of Pretoria, Pretoria, South Africa
dkourie@cs.up.ac.za

The objective of this paper is to explain the notion of fine-grain transformations (FGTs), showing how they can be used as prototypical building blocks for constructing refactorings of a design-level system description. FGT semantics are specified in terms of pre- and postconditions which, in turn, also determines the sequential dependency relationships between them. An algorithm is provided which uses sequential dependency relationships to convert an FGT-list to a set of so-called FGT-DAGs. It is shown how to compute the precondition of such ordered collections of FGTs. The paper introduces a new approach to deal with refactoring pre- and postconditions by defining them at two different levels. To give these concepts syntactical form, we rely on the Prolog formats used by an FGT-based refactoring prototype tool. An example is provided to illustrate the various concepts and to demonstrate that, because of their simplicity, well-defined pre-post semantics and their intuitive nature, FGTs provide a pragmatic basis for building refactorings.

Keywords: Refactoring; FGT-based refactoring; fine-grain transformations; refactoring-level precondition; FGT-enabling precondition; directed acyclic graph.

1. Introduction

Refactoring is the process of improving the internal structure of the software while preserving its external behaviour [1−3]. Behaviour is preserved if the refactored software produces exactly the same output as the original software for every possible set of inputs. The purpose of refactoring software is to render it easier to understand, to extend, to find bugs, etc. [4, 5].

A current trend is to apply refactorings at levels of abstraction above the code level [4, 6–8]. Typically such refactorings are applied at the design-level of the system, where the system is represented by a UML model. This is because many 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 (e.g. in terms of methods or classes instead of code) can make refactoring more efficient [4]. For the present purposes, it will be assumed that the UML model represents a Java-based system — i.e. the system conforms to Java's semantics in regard to inheritance, access modes, basic types, etc.

Refactoring theory generally starts by identifying a finite set of design-level primitive refactorings [2, 3, 5]. Each primitive refactoring is characterised by a precondition to which the system to be refactored (henceforth simply referred to as the system) should comply. Compliance guarantees two things: (a) that the refactoring will conform to the semantic rules that apply to the system (e.g. no overloading of class names); and (b) that the system's behaviour will not be changed by the refactoring. After applying the primitive refactoring, the system complies with its postcondition. Pre- or postconditions are predicates consisting of one or more conjuncts, each of which asserts the existence or non-existence in the system of a named object, a relation between objects, or some particular property of an object. Simple examples of primitive refactorings are: "add class C to package P"; "add a subclass relation between classes C1 and C2"; or "change the access mode of method M". The precondition of the "add class C to package P" refactoring is a predicate asserting that "class C does not exist in package P", and the postcondition is a predicate asserting that "class C exists in the package P". Subsequent primitive refactorings may add methods and attributes to the class, specify relationships between it and other objects, etc.

A refactoring tool which is based on these principles will require access to a design-level description of the system. The next section describes the format used in a Prolog-based prototype refactoring tool that has been built to test and illustrate ideas discussed in the rest of this paper.

Conventionally, a primitive refactoring is implemented in refactoring tools as a procedure. In Sec. 3, prototypical transformations, called fine-grain transformations (FGTs), that occur within such hard-coded procedures are identified and described in terms of their respective pre- and postconditions. Also described in this section is the notion of sequential dependency between FGTs.

Section 4 then discusses the notion of pre- and postconditions of ordered collections of FGTs. It is shown that primitive refactorings may be expressed in terms of one such ordered collection, a feasible FGT-list. An algorithm is provided to convert a feasible FGT-list into one or more so-called FGT-DAGs, where each FGT-DAG is also an ordered collection of FGTs. The resulting set of FGT-DAGs has characterising pre- and postconditions which are the same as the pre- and postconditions of the input FGT-list. Moreover, the structure of the set of FGT-DAGs reflects the sequential dependency relationships of FGTs in the input FGT-list. However, it turns

out that if an FGT-list representing a primitive refactoring is applied to a system that complies with the FGT-list's precondition, system behaviour will not necessarily be preserved. It is shown that in order to guarantee preservation of behaviour, the system should also comply with a so-called refactoring-level precondition which characterises some of the primitive refactorings. It is also briefly pointed out that these concepts carry over to composite refactorings, although details in this regard are provided in [9].

Before concluding, Sec. 5 walks the reader through an example to illustrate concretely the various concepts that have been described.

It should be pointed out that our early ideas on using FGTs for refactoring were published in [10]. Subsequently, two conference presentations were made [11, 12] which outline in broad overview the way in which FGTs can be used for refactoring. The main focus of our previous work was on introducing the idea of FGTs and its feasibility to construct refactoring. The dealing with pre- and postcondition conjuncts of the refactoring was not considered. Therefore, in the new submission, we extend our previous work with the contribution of proposing a generalized approach to address this problem. The pre and postcondition conjuncts of the refactoring are defined at two different levels instead of just one. One of these two levels can be inferred automatically from the constitute FGTs of that refactoring as shown in Secs. 4.2 and 4.4. Accordingly, the present paper augments the information provided in the previous work, providing algorithmic details, refining some of the earlier notions of pre- and postconditions of ordered FGT collections, and describing the stages that are needed to apply a refactoring on a system which differ from all the previous approaches including our previous ones.

In addition, in [9] details were provided on how a single set of FGT-DAGs can be created to represent a composite refactoring by merging the respective sets of FGT-DAGs associated with the constituent primitive refactorings.

2. Representing the System

Because of its overall suitability for prototyping, it was decided to build a prototype tool in 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 refactorings 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.

The first task is to represent a UML class diagram description of the system as Prolog facts (or logic-terms). The vocabulary 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.

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 indicates 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 recognised in the graph-based approach to refactoring, pioneered by Mens [14].

In principle, access information could be obtained from sequence- and/or state diagrams (provided that the diagrams are adequately annotated). It is also relatively easy to write software to extract access information directly from the code. However, these matters are not considered further in this paper. Neither is a subsequent concern addressed; namely, that of keeping various forms of system representation (class diagrams, sequence diagrams, code, etc) consistent with one another after a refactoring. For the present purposes, it is assumed that this access-related information is available to the system, and that the refactored system stores the new access-related information in the format described below.

The following two subsections show how the logic-terms relating to object elements and relational elements are constructed. It will be seen that, in general, the functor name (indicated in bold) denotes the object or relation type, and the first argument of each logic-term is a system-unique identifier (an ID) for the associated model element. The other functor arguments are properties of that element (name, definition type and access mode) or are IDs referring to other model elements. The IDs are necessary for disambiguation where names (e.g. method names) are overloaded.

2.1. *Object element logic-terms*

Object element facts represent the *object* elements of a UML class diagrams and can be extracted directly from the UML class diagram of the system. The information in the class diagrams is sufficient for their representation — it is not necessary to reference additional access-related information in, for example, the code. The format of the relevant terms is as follows:

```
package(PID,OwnerID,PName,CsList)
class(CID,PID,CName,AccMode,MethsList,AttrsList)
attribute(AttrID,CID,AttrName,DefType,AccMode)
method(MethID,CID,MethName,RetType,AccMode,PrmsList)
parameter(PrmID,MethID,PrmName,DefType)
```

where:

— *AttrID* is a ID of the attribute. *CID* is a ID of a class. *OwnerID* is a ID of a container in which packages are located. *MethID* is a ID of a method. *PID* is a ID of a package. *PrmID* is a ID of a parameter.

— *AttrName* is the name of the attribute. *CName* is the name of a class. *MethName* is the name of a method. *PName* is the name of a package. *PrmName* is the name of a parameter.
— *AttrsList* is a list of IDs of attributes (*AttrIDs*) in a class. CsList is a list of IDs of classes (*CIDs*) in a package. *MethsList* is a list that contains the IDs of methods (*MethIDs*) defined in a class. PrmsList is a list of IDs of method parameters (*PrmIDs*).
— *AccMode* is the access mode (public, private, protected) of a class, method or attribute. *DefType* is the definition type of an attribute or parameter. *RetType* is the definition type of the return value of the method.

The definition types referred to above (*DefType* and *RetType*) are instantiated to a logic-term of the form: `type(Comp,Name,Num)`. Here, *Comp* indicates the type's "complexity", either as *basic* or as *complex*; *Num* is an integer that indicates the dimension of the array if the type is an array, or is set to 0 if it is not an array. If *Comp* is set to *basic*, then *Name* indicates the basic type as *int* or *float*, etc. If Comp is set to *complex*, the *Name* is assigned the ID (e.g. some instance of *CID*) associated with the relevant class (or interface, etc.).

Also note the order of the parameter identifiers in *PrmsList* corresponds to the order of parameters in the signature of the associated method, and can thus be used in method disambiguation at the signature level.

2.2. *Relation elements logic-terms*

This group of logic-terms represents *relation* elements of the UML class diagrams. Each *relation* logic-term denotes a specific *relation* between one source- and one destination object element in the UML class diagram. All logic-terms have the same format and arguments, namely:

RelationType(*RID,Label,SourceID,DestinationID*) where:

— *RID* is a ID of the *relation*. *Label* is a label of the *relation* (if it exists). *SourceID* is the ID of the source *object* element of the *relation*. *DestinationID* is the ID of the destination *object* element of the *relation*.

Logic-terms of this group include **extends**(*RID, Label, SourceID, DestinationID*) and **association**(*RID, Label, SourceID, DestinationID*) which denote an *extends* (generalization, specialization) or an association relation, respectively, between the *object* elements *SourceID* and *DestinationID*. The *Label* corresponds to whatever label has been used in the UML diagram. Additionally, logic-terms with functors **read**, **write**, and **call** indicate the corresponding actions between the relevant *SourceID* and *DestinationID* objects in the UML diagram. Typically they express the fact that a given source method reads from- or writes to a given destination attribute; or that a given source method calls another destination method. In practice, information for these logic-terms has to be inferred from the code or from

UML sequence- or state diagrams. Finally, a logic-term with functor **type** is available to indicate that a given source attribute has a given destination type. Note that the second argument, *Label*, in the logic-terms **read**, **write**, **call** and **type** is given as the Prolog "don't care" (i.e. "_") variable. This reflects the fact that labels are not used in these cases. Also note that if a method **reads** an attribute more than once from the same destination object, then just one **read** relation will be recorded. The same applies for the **write** and **call** relations.

3. Fine-Grain Transformations

When applying a refactoring to a system, the prototype Prolog tool accesses and manipulates information that conforms to the above schema. Applying a refactoring entails (a) checking that the precondition conjuncts of the relevant refactoring are satisfied and (b) retracting existing logic-terms and/or asserting new logic-terms as required by that particular refactoring. In the present work, applying a refactoring involves applying a number of FGTs and these too, each have specific preconditions which have to be checked before application. This section will elaborate further on FGTs, their associated preconditions, and their Prolog representation.

An FGT is an *abstract operation* on a UML model. Thus, operands of an FGT are elements of a UML model, and the model will undergo an incremental *atomic* change as a result of applying an FGT to it. The change is atomic in the sense that it cannot be broken down into further smaller change steps from the modelling perspective. The operation is abstract in the sense that it could be specified in a wide variety of concrete syntactic representations.

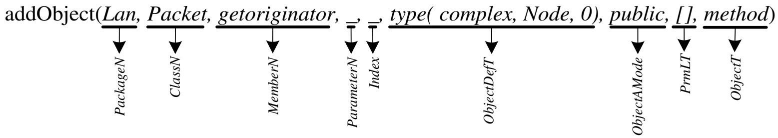
In our prototype, FGT operations are specified in terms of Prolog rules. In general, rules are provided of the form: `F:-A1,A2,`... An where the rule's head, F, describes the specific FGT to be implemented, and A1,... An are Prolog terms to be instantiated in reference to the Prolog database, in order to carry out the FGT-specified operation. Section 3.1 describes the form that F takes. After that, Sec. 3.2 describes the form that the first term, A1, takes — i.e. the term that activates relevant rules to check the FGT's preconditions. Other terms (A2,... An) manipulate the Prolog database by asserting and retracting logic-terms from it to modify the system description in line with the requirements of the associated FGT.

3.1. *Implementing FGTs*

The set of FGTs is partitioned into two groups, corresponding to the previously mentioned categories of UML class diagram entities: objects and relations.

Object Element FGTs are concerned with all the transformation operations whose characterising operands are *object* elements of the UML class diagram. They include addObject, deleteObject, renameObject, changeOAMode, changeODefType. They are used to modify a class diagram by respectively adding, deleting or renaming *object* elements; or by changing an object's access mode or definition type.

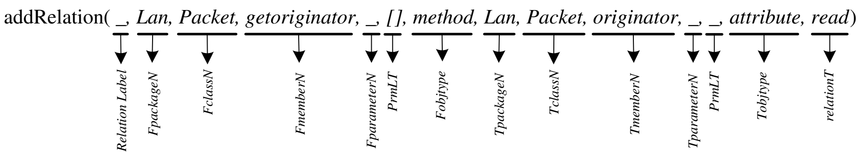
As an example of the Prolog implementation of FGTs in this group, the following represents the head of Prolog rule that is used to add to the class diagram an *object* element with name *getoriginator* and access mode *public*. It is to be added to the class *Packet* that is in the package *Lan*. The *object* is to return a value (that is not an array) of complex type *Node* class. The last argument of the FGT tells the tool that the added *object* in this FGT is of type *method*. The empty list *PrmLT* indicates that the added method has no parameters.



After applying this FGT, a fact will be asserted into the underlying database to represent this method. Its form could be, for example: `method(46,2,getoriginator,type(complex,1,0),public,[])` where 46 is a system-assigned ID for the new method; 2 is the ID of the class called *Packet* in which the new method is defined; 1 in the term *type* is the ID of the definition type of the return value of the method, which is in this case the class *Node*. Note that the system has been designed to infer the packet and class IDs from their corresponding names given in the FGT.

The second group of FGTs is concerned with transformation operations on *relational* elements of a UML class diagram. These FGTs will be called *Relational Element FGTs*. FGTs of this group are used to add, rename or delete relations between two object elements.

As an example of the Prolog representation of FGTs in this group, a *read* relation whose source is the method *Lan.Packet.getoriginator* and whose destination is the attribute *Lan.Packet.originator*, may be asserted into the database using the following:



After applying this FGT, the following fact is asserted into the underlying database: **read**(47, _, 46, 2002) where 47 is the system-assigned ID of the new *read* relation; 46 is the ID of the source *object*, namely the method *Lan.Packet.getoriginator*; and 2002 is the ID of the destination object, namely the attribute *Lan. Packet.originator*. The _ indicates that a *read* relation does not have a label.

3.2. *FGT preconditions*

Each FGT has a precondition (a set of conjuncts) which must be true in a system before the FGT may be applied legitimately to that system. In some cases, a conjunct may itself consist of a number of disjuncts (e.g. (X or Y)). A procedure called

FGTPrecondConj is implemented in the refactoring tool for each one of the proposed FGTs. Assuming that $C_1, C_2, \ldots C_N$ are precondition conjuncts for FGT, the Prolog rules for this procedure take the form: **FGTPrecondConj**(FGT):- $C1, C2, \ldots CN$. To illustrate these precondition rules, the preconditions are briefly discussed below of two FGTs, namely: **addObject** and **addRelation**.

The FGT **addObject**(*Pn,Cn,Methn,,,ODefT,OAMode,PrmLT,method*) adds to the class *Pn.Cn* a new method called *Methn* with access mode *OAMode*, return type *ODefT* and parameter list *PrmLT*. This FGT's precondition consists of the following six conjuncts:

(1) The class *Pn.Cn* should be already declared in the system.
(2) Methn should be unique in *Pn.Cn*.
(3) Methn should not be in an ancestor class of *Pn.Cn*.
(4) The access mode *OAMode* should be valid (e.g. no public method in a private class).
(5) The return value's definition type, *ODefT*, should be valid (e.g. cannot return a non-existent class type).
(6) The type of the return value *ODefT* should be accessible.

Clearly, the FGT should not be applied unless all conjuncts hold. For example, the third conjunct prevents redefinition of an inherited method in a class, since such a redefinition risks changing the behaviour of the system. On the other hand, adding a method into a class will not risk behaviour change if there are methods of the same signature in descendant classes, and so no conjunct considers methods of the same signature in descendent classes. As another example, consider the last conjunct. It ensures that if an object of type class is to be returned, then the access mode of that class should be accessible. Accordingly, the prototype has a rule of the following form:

```
FGTPrecondConj(addObject(Pn,Cn,Methn,,,ODefT,OAMode,PrmLT,method)):-
  existsObject(Pn,Cn,class),
  not(existsObject(Pn,Cn,Methn,PrmLT,method)),
  not(isInherited(Pn,Cn,Methn,PrmLT,method)),
  validDefType(ODefT),
  validOAMode(OAMode,method),
  canAccessType(ODefT).
```

Note that the comma (,) between the two conjuncts indicates a "*logical and*".

Next, consider the FGT: **addRelation**(*,FPn,FCn,FMethn,,FPrmLT,method, TPn,TCn,TAttn,,,attribute,RelT*) where *RelT* may be either a *read* or a *write* relation. This FGT is used to add a *read/write* relation from the source, namely method *FPn.FCn.FMethn* which has *FPrmLT* as its parameter list, to the destination,

namely attribute *TPn.TCn.TAttn*. This indicates that at the code level there will be one or more statements in the method *FMethn* that will *read/write* the attribute *TAttn*. To apply this FGT on the system the following should hold:

(1) Method FPn.FCn.FMethn with parameters FPrmLT should be declared in the system.
(2) Attribute *TPn.TCn.TAttn* should be declared in the system.
(3) Relation *RelT* between the two objects does not exist.
(4) If the access mode of *TAttn* is private then *FMethn* should be in the same class as *TAttn*.
(5) If the access mode of *TAttn* is default then *FMethn* should be in the same package as *TAttn*.
(6) If the access mode of *TAttn* is protected then *FMethn* should be in the same package as *TAttn* or in one of the subclasses of the class *TPn.TCn*.
(7) If the access mode of *TAttn* is public then *FMethn* can be anywhere.

```
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 the vertical bar (|) between the two conjuncts indicates a "*logical or*".

In [15] the Prolog representation of all object- and relational FGTs has been catalogued in detail, together with appropriate rules such as those above for checking their preconditions.

3.3. *FGT sequential dependency*

It is evident that whenever an FGT is applied to a system, one or more of its precondition conjuncts (but not necessarily all) will be negated. For example:

- When adding an object, the precondition that the object does not exist is negated, since the object now exists in the system.
- In renaming an object, the precondition requires that an object of the old name exists, and an object of the new name may not exist. After application of the relevant FGT, the negation of these requirements is true.

The conjunction of predicates that become true after applying an FGT constitutes an FGT's *postcondition.* The interplay between the pre- and postconditions of FGTs determines whether or not they are sequentially dependent.

FGT_j is said to be (*inherently*) *sequentially dependent* on FGT_i if and only if the postcondition conjuncts of FGT_i satisfies one or more precondition conjuncts of FGT_j. The sequential dependency between the two FGTs is represented by: $\mathrm{FGT}_i \rightarrow \mathrm{FGT}_j$. For example, suppose

`addObject`(*P,A,m1,,,type(basic,void,0),public,[],method*)
represents the FGT that adds the method *m1* into the class *P.A* and
`addObject`(*P,A,,,,,public,,class*)
represents the FGT that adds the class A in the package *P*. Then:
`addObject`(*P,A,,,,,public,,class*)→ `addObject`
(*P,A,m1,,,type(basic,void,0),public,[],method*).

The sequential dependence is evident, since one obviously first has to add the class *P.A* before adding members in it.

Note that, as defined above, the sequential dependency between two FGTs is *inherent*, in the sense that it does not depend on the description of the system. (This is in contrast with Roberts [3], who defines sequential dependency between two refactorings relative to a program or system.) This means that if $\mathrm{FGT}_j \rightarrow \mathrm{FGT}_i$ and there is a need to apply FGT_i to some given system, S, one of the following scenarios may occur:

- S already satisfies all precondition conjuncts of FGT_i. In this case, FGT_i may be directly applied to S. In this case, it would not be possible to first apply FGT_j to S, since the satisfaction of all FGT_i's precondition conjuncts indicates that at least one of FGT_j's precondition conjuncts is not satisfied by S.
- S does not satisfy all precondition conjuncts of FGT_i. In this case it may be necessary to select one or more FGTs upon which FGT_i sequentially depends, to apply it (or them) to S, and then to apply FGT_i. Whether or not FGT_j is to be included in this selection depends on the description of S.

For a full catalogue of all the possible sequential dependencies that may exist between the different FGTs defined in the approach, the interested reader is referred to [15].

4. Ordered Collections of FGTs

This section considers various ordered collections of FGTs. It defines the notion of a *feasible* FGT-*list* and that of an FGT *directed acyclic graph* (FGT-DAG). It shows how the former can be mapped to a *set* of the latter by relying on the FGT sequential dependency relationships just discussed. It also shows how the precondition for such

an ordered collection of FGTs may be derived by considering the ordering, as well as the pre- and postconditions of the FGTs in the ordered collection. It then relates these ordered collections of FGTs to primitive and composite refactorings, introducing the notion refactoring-level preconditions.

4.1. *Feasible FGT-lists*

A feasible FGT-list is a list of FGTs for which at least one system exists, such that the FGTs in the list can feasibly be applied to the system, starting at the head of the list and applying each successive FGT until the tail of the list has been applied. By the phrase "an FGT can feasibly be applied to the system" we mean that the FGT's precondition is satisfied just prior to its application.

A consequence of applying a feasible FGT-list to an appropriate system is that a set of objects and a set of relations (each possibly empty) will be guaranteed to exist in the system; and a set of objects and a set of relations (each possibly empty) will be guaranteed *not* to exist in the system. The conjunction of the assertions about the existence and non-existence of these entities can be regarded as the feasible FGT-list's postcondition.

Of course, not every list of FGTs is feasible. For example, any FGT-list that specifies two successive deletions of the same object cannot be feasible. Such FGTs are said to conflict. For more information about FGT conflict pairs, and the detection of conflicts, the interested reader is referred to [15]. In the present context, it will be assumed that all FGT-lists are conflict-free, unless otherwise stated. How such a feasible list of FGTs is derived is currently not of concern. It may, for example, be proposed by a developer who wishes transform a given system design in some particular way. Whether or not the transformation retains the original system behaviour (i.e. whether or not it constitutes a refactoring) is also not of immediate concern.

Clearly, if an FGT-list is to be applied to a particular system, that system should comply with certain requirements that ensure that the FGTs in the list can indeed be applied in the given order — i.e. the FGT-list has a *precondition* and this precondition is composed of conjuncts to which the system should conform.

Note that an FGT-list's precondition is not simply the conjunction of all precondition conjuncts of its constituent FGTs. Instead, it consists of the conjunction of only those FGT precondition conjuncts that are *not* negated as a result of applying the FGTs. For example, consider the feasible FGT-list [FGT_1, FGT_2]. Suppose the precondition of FGT_1 is P_1^P_2 and the precondition of FGT_2 is P_3^P_4. Suppose, also, that the postcondition of FGT_1 is P_3 (or, more generally, that the postcondition of FGT_1 logically entails P_3, but not P_4). Then the precondition of the FGT-list is

P_1^P_2^P_4. Since a similar notion applies to other ordered collections of FGTs (as discussed in the next subsection), a formal definition of an FGT-list's precondition is deferred until then.

4.2. *FGT-DAGs*

An FGT-DAG is a directed acyclic graph in which each node represents an FGT, and there is an arc between two nodes, say from node FGT_j to FGT_i, if and only if:

(1) FGT_i is sequentially dependent on FGT_j;
(2) FGT_i is not sequentially dependent on any successor of FGT_j; and
(3) no ancestor of FGT_j has an arc to FGT_i (even if FGT_i is sequentially dependent on that ancestor).

An FGT-DAG is applied to a system by applying the FGTs in any order that respects the sequential dependency relationships represented by the arcs. This means that a given FGT may only be applied after all its ancestors have been applied.

As in the case of a FGT-list, an FGT-DAG is characterised by a precondition and a postcondition. Again, the postcondition is simply the conjunction of predicates asserting what objects and relations exist and/or do not exist as a result of applying the FGT-DAG.

To formalize the notion of an FGT-DAG's precondition (and, simultaneously, formalise the definition of an FGT-list's precondition), consider an arbitrary FGT in the FGT-DAG (or FGT-list). A subset of its precondition conjuncts can be considered "enabling" in the context of a specific FGT-DAG (or FGT-list). The enabling conjuncts of a given FGT in a given context is defined as follows:

> If $\text{Pre} = \{P_i \mid i = 1,\ldots n\}$ is the set of the FGT's precondition conjuncts, and $\text{Post} = \{Q_j \mid j = 1,\ldots m\}$ is the set of postconditions of all of its parents (whether in the FGT-DAG or the FGT-list), then $\text{En} = \{P_i \in \text{Pre} \mid \forall\, Q_j \in \text{Post} : \sim(Q_j \Rightarrow P_i)\}$ defines the set of precondition conjuncts of the FGT that are not entailed by its parents' postconditions. This will be called the *FGT's set of enabling precondition conjuncts* in the context of the FGT-DAG (or FGT-list).

An FGT-DAG's (or FGT-list's) precondition is thus the conjunction of the enabling precondition conjuncts of FGTs in the FGT-DAG (or FGT-list).

From the definition of an FGT-DAG, it is evident that each FGT has the property that the postcondition of each of its parents logically entails of one or more of that FGT's precondition conjuncts. As a result, the precondition of an FGT-DAG (consisting of more than one FGT) will not simply the conjunction of all FGT preconditions.

Clearly, if a system complies with an FGT-DAG's preconditions, then the FGTs in the FGT-DAG can systematically be applied to the system in the order determined by the FGT-DAG, with the assurance that all FGT preconditions will be fulfilled by the system when they are to be applied to the system. As a consequence, the system will conform to the FGT-DAG's postcondition.

These notions can be extended directly to *sets* of FGT-DAGS. The conjunction of the preconditions (postconditions) of all FGT-DAGs in a set of FGT-DAGS constitutes the precondition (postcondition) of the set of FGT-DAGs.

Note that it is possible that the precondition of an arbitrary set of FGT-DAGs may evaluate to *false*. This means that no system exists to which the set of FGT-DAGs may be applied. In such a case, the set of FGT-DAGs may be considered to be *infeasible*; otherwise the set of FGT-DAGs is *feasible*. Note also that the precondition of a set of FGT-DAGs may also be thought of as the conjunction of all enabling precondition conjuncts of all FGTs in the set.

4.3. *From FGT-lists to sets of FGT-DAGs*

The question arises: How can a (feasible) FGT-list be mapped to a (feasible) set of FGT-DAGs that has the same postcondition as the FGT-list? An algorithm, called **build-FGT-DAG** has been implemented in the prototype tool that achieves this objective. While obviously implemented in our prototype as logic program, its pseudo-code is given in Algorithm 1 below in imperative form. The algorithm derives from a feasible FGT-list, *FGTList*, a set of FGT-DAGs, *DSET*, that has the same postcondition as *FGTList*.

It does this by setting *DSET* to the empty set, and then processing the FGTs in *FGTList* from first to last. Each next FGT, FGT_i, to be processed begins as a new singleton FGT-DAG in *DSET*. All paths in the other FGT-DAGs in *DSET* are then traversed in a bottom-up fashion, searching for the first FGT upon which FGT_i sequentially depends. If such a node, FGT_j, is found in a path, it is connected to FGT_i and all ancestors of FGT_j are eliminated as candidates for further consideration.

When the algorithm completes, each FGT in *FGTList* will have been inserted into one and only one FGT-DAG in *DSET*. Each FGT node will have inbound arcs from the closest FGTs that precede it in *FGTList* upon which it sequentially depends. It will have outbound arcs to the closest FGTs following it in *FGTList* and which are sequentially dependent on it. Note that the algorithm has been designed to ensure that whenever a candidate bi-directional sequential dependency relationship between two elements in *FGTList* is found, the direction reflected in the FGT-DAG corresponds to the intended execution order dictated by *FGTList*.

Algorithm 1 (Building FGT-DAGs algorithm)

```
build-FGT-DAG (FGTList)
Input: FGTList: A feasible list of FGTs
Output: DSET: A set of FGT-DAGs whose postcondition is the same as
that of FGTList
Set DSET to the empty set
For each FGT_i in FGTList do { //FGTs should be selected in order from
first to last
       Mark each FGT in each FGT-DAG of DSET as unchecked
       Insert FGT_i into DSET as a single node of a new FGT-DAG and
         mark it as checked
       While there are unchecked FGTs in DSET do {
    Select an unchecked FGT with no unchecked children, say FGT_j
    Mark FGT_j as checked
    If FGT_j → FGT_i (as determined from oneDirSD and twoDirSD) then {
                 Mark all ancestors of FGT_j as checked
                 Insert an arc from FGT_j to FGT_i
            } //end If
      } //end While
} //end For
Return DSET
```

Note that the structure (FGT-DAGs) produced by the algorithm are indeed acyclic, and not cyclic. This can be verified by considering the following two points:

(1) The resulting set of DAGs represents a feasible FGT-list. As mentioned above, the FGTs of a feasible FGT-list are ordered according to their sequential dependencies in such a way that their overall precondition does not evaluate to false.

(2) Logically, the set of DAGs have been set up in such a way that they encapsulate the sequential dependency between the FGTs. The sequential dependency conveys the nature of the pre/post conditions of the FGTs. Suppose that one FGT-DAG contained a cycle of sequential dependencies, for example: A→B→C and C→A. This would mean that part of the post conditions of C is needed to satisfy the preconditions of A and at the same time part of the postcondition of A is needed to satisfy the preconditions of C, which leads to a contradiction. Such a contradiction could only arise if the input FGT-list was not feasible.

There are several advantages to being able to convert an FGT-list to a set of FGT-DAGs. For example, it allows for the detection and possible elimination of conflicts and redundancies. It also exposes groups of FGTs which are mutually

independent and which can therefore be applied independently (i.e. in parallel) to a system. For further discussion of these matters, the reader is referred to [15].

4.4. *FGTs and refactorings*

Refactorings may either be primitive or composite. A *primitive refactoring* is a refactoring that is atomic — it cannot be split into more refactorings. The list of primitive refactorings that are commonly agreed upon in the literature [2, 3, 5] is shown in Table 1, grouped according to whether they add, delete or change model elements. The literature also notes a precondition (not shown in the table) associated with each primitive refactoring, to which a system should conform in order to preserve behaviour when the refactoring is applied to the system.

In seventeen cases (marked by an asterisk in the table) the primitive refactoring maps directly onto an FGT. In [15] it is shown how each of the remaining twelve primitive refactorings in Table 1 may be mapped to a feasible FGT-list. In general, therefore, any primitive refactoring may be regarded as an ordered collection of FGTs, either in the form of an FGT-list, or its equivalent representation as a set of FGT-DAGs.

A *composite refactoring* is a collection of primitive refactorings that are applied on the model as one unit. The order in which the primitive refactorings are executed is important. On the one hand, the net effect of the changes wrought by each primitive refactoring has to comply with the intentions of the designer of the composite refactoring. On the other hand, a given application ordering may be discovered to be infeasible only after several primitive refactorings have been applied, demanding a roll-back to the original system and the possible re-application of the refactorings in a different order that nevertheless meets the original objectives. In [16] it is shown how a precondition can be derived for the composite refactoring that consists of a given sequence of primitive refactorings. This precondition is based on

Table 1. Primitive refactorings.

Add element refactorings	Delete element refactorings	Change element refactorings	
		Change characteristics	Change structure
addClass* addMethod* addAttribute* addParameter*	deleteClass deleteMethod* deleteAttribute* deleteParameter*	renameClass* renameMethod* renameAttribute* renameParameter*	changeSuper moveMethod moveAttribute
addGetter addSetter		changeClassAccess* changeMethodAccess* changeAttributeAccess*	attributeReadsToMethodCall+ attributeWritesToMethodCall+
		changeMethodType* changeAttributeType* changeParameterType*	pullUpMethod+ pullUpAttribute+ pushDownMethod+ pushDownAttribute+

the pre- and postconditions of the constituent primitive refactorings. In [9] it is shown that a composite refactoring can also be regarded as a collection of FGTs. We report on how to map a composite refactoring to a *single* set of FGT-DAGs, (i.e. the individual sets of FGT-DAGs corresponding to each primitive refactoring are merged) and use the set's precondition to avoid roll-back.

It will be convenient to use the collective term "ordered collection of FGTs" to refer either to an FGT-list, a sequence of FGT-lists, an FGT-DAG, or a set of FGT-DAGs. Summarising the foregoing discussion, it is clear that: (a) each primitive refactoring maps to an FGT-list which, in turn, maps to a set of FGT-DAGs; (b) each composite refactoring can be mapped to a sequence of FGT-lists which, in turn, can be merged into a single set of FGT-DAGs; (c) FGTs, ordered collections of FGTs, primitive refactorings and composite refactorings all have pre- and post-conditions; and (d) the precondition of an ordered collection of FGTs is inferred from the ordering of the constituent FGTs alone.

However, it is important to observe that for a given mapping of a refactoring to an ordered collection of FGTs, the behaviour of a system will *not* necessarily be preserved, simply because the precondition of the ordered collection of FGTs is satisfied before the individual FGTs are applied (in the specified order) to the system. The preconditions of certain refactorings include conjuncts that cannot be inferred from the respective preconditions of the individual FGTs in the ordered collection of FGTs.

A refactoring's precondition conjuncts that are *not* logically entailed by the precondition of the equivalent ordered collection of FGTs will be referred to as *refactoring-level precondition conjuncts* of the ordered collection of FGTs. The refactoring-level precondition conjuncts for all FGT-lists mapping to primitive refactorings have been identified in [15]. The primitive refactorings concerned are marked with a plus (+) in Table 1. Typically, these refactoring-level preconditions demand specific visibility and access mode levels of the entities involved in the refactoring, even though these requirements are not necessary for carrying out the constituent FGTs. These notions are abstractly portrayed in Figs. 1(a) and 1(b) with respect to an FGT approach and previous approaches respectively.

In the FGT-based prototype refactoring tool, refactoring-level preconditions are explicitly stored for the relevant primitive refactorings, and for any stored composite refactorings. The tool uses them as a first-pass screening test to check whether or not a refactoring may be applied to a given system. Only after a system has passed this test is it checked against the precondition of the relevant ordered collection of FGTs. If this second level of tests also succeeds then the tool proceeds to the final phase in which the refactoring itself is applied to the system by applying, in an appropriate order, each FGT in the ordered collection of FGTs.

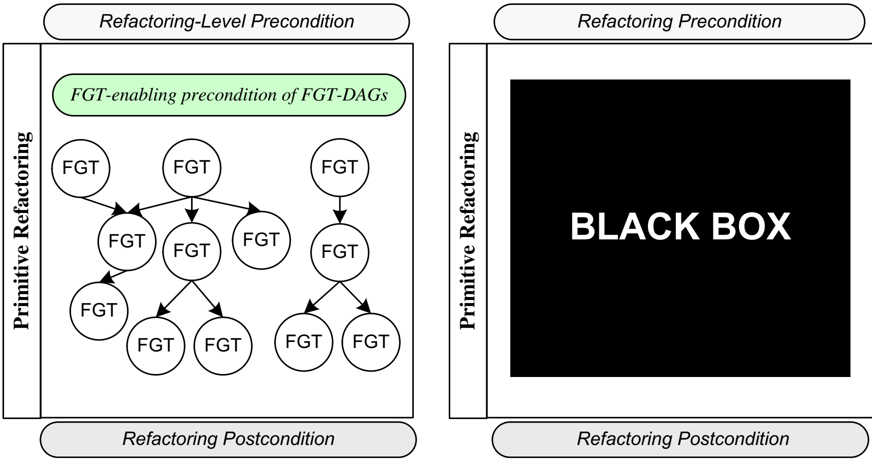


Fig. 1. Primitive refactoring different considerations.

5. An Example

To illustrate the approach outlined above, an example is presented that is frequently used for teaching refactoring: the simulation of a Local Area Network (Lan) [17]. Initially there are five classes: *Packet, Node* and the three subclasses: *Workstation, PrintServer and FileServer*. The idea is that all *Node* objects are linked to each other in a token ring network (via the *NextNode* variable) and that they can send or accept a *Packet object*. *PrintServer, FileServer* and *Workstation* refine the behaviour of *Node* objects. A *Packet* object can only originate from a *Workstation* object, and sequentially visits every *Node* object in the network until it reaches its receiver that accepts the *Packet*, or until it returns to its originator Workstation (indicating that the *Packet* cannot be delivered).

The UML class diagram for the LAN example is shown in Fig. 2. The dashed arrows represent the extra information extracted from the code level of the LAN system. Before doing any refactoring, information in Fig. 2 has to be represented as collection of logic-terms as discussed in Sec. 2. Figure 3 shows the collection of logic-terms for the LAN simulation example. All refactorings will be done on this underlying representation of the model.

Suppose that it is required to enhance the structure of the LAN model by:

- encapsulating the *Packet* class' *originator* attribute;
- creating a *Server* superclass of *PrintServer* and *FileServer* which is also a subclass of *Node*; and then
- pulling up the *accept* method from *FileServer* and *PrintServer* to their new *Server* superclass.

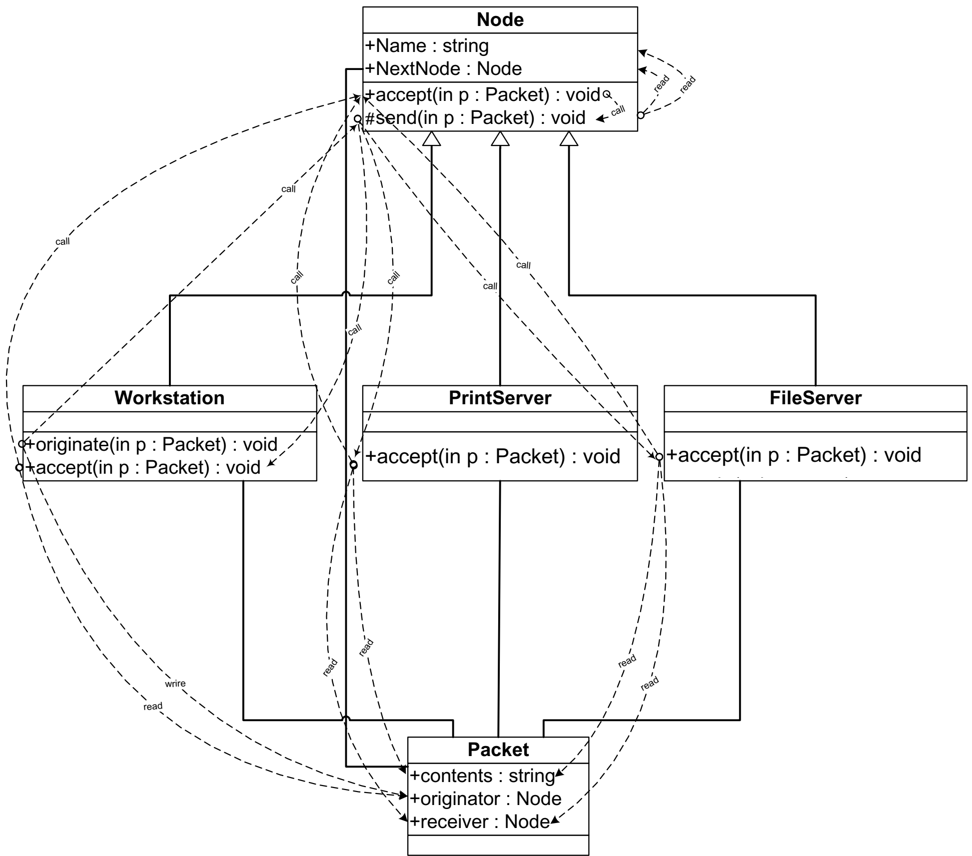


Fig. 2. A UML class diagram of the LAN simulation before refactoring.

For this restructuring two composite refactorings called **encapsulateAttribute** and **createClass** respectively, can be applied to the system, followed by the **pullUpMethod** primitive refactoring.

Each of the composite refactorings is specified below as a sequence of primitive refactorings, each of which is represented, in turn as an FGT-list. In the next subsection, the various FGTs are considered in turn and a number of subtleties relating to their system representation are pointed. This is followed by a subsection which discusses issues relating to preconditions and FGT execution order in greater detail.

5.1. *FGTs in the refactorings*

As shown in the first column of Table 2, **encapsulateAttribute** (*'Lan'*,*'Packet'*, *originator*) invokes the **addGetter** and **addSetter** primitive refactorings to install *getoriginator* and *setoriginator* methods in the *Lan.Packet* class. It then invokes the

```
package(0,00,Lan,[1,2,3,4,5]).
class(1,0,Node,public,[1001,1002],[10001,10002]).
method(1001,1,send,type(basic,void,0),protected,[100001]).
method(1002, 1, accept,type(basic,void,0),public,[100002]).
attribute(10001,1,Name,type(basic,string,0),public).
attribute(10002,1,NextNode,type(complex,1,0),public).
parameter(100001,1001,p,type(complex,2,0)).
parameter(100002,1002,p,type(complex,2,0)).
call(1000001,_,1002,1001).
call(1000002,_,1001,3002).
call(1000003,_,1001,4002).
call(1000004,_,1001,5002).
read(1000008,_,1001,10001).
read(1000009,_,1001,10002).
extends(10000010,isa,1,3).
extends(10000011,isa,1,4).
extends(10000012,isa,1,5).
class(2,0,Packet,public,[],[20001,20002,20003]).
attribute(20001,2,contents,type(basic,string,0),public).
attribute(20002,2,originator,type(complex,1,0),public).
attribute(20003,2,receiver,type(complex,1,0),public).
```

```
class(3,0,FileServer,public,[3002],[]).
method(3002,3,accept,type(basic,void,0),public,[300002]).
parameter(300002,3002,p,type(complex,2,0)).
read(3000001,_,3002,20001).
read(3000003,_,3002,20003).
call(3000005,_,3002,1002).
class(4,0,PrintServer,public,[4002],[]).
method(4002,4,accept,type(basic,void,0),public,[400002]).
parameter( 400002,4002,p,type(complex,2,0)).
read(4000001,_,4002,20001).
read(4000003,_,4002,20003).
call(4000005,_,4002,1002).
class(5,0,Workstation, public,[5001,5002],[]).
method(5001,5,originate,type(basic,void,0),public,[500001]).
method(5002,5,accept,type(basic,void,0),public,[500002]).
parameter(500001,5001,p,type(complex,2,0)).
parameter(500002,5002,p,type(complex,2,0)).
write(5000001,_,5001,20002).
call(5000002,_,5001,1001).
read(5000004,_,5002,20002).
call(5000005,_,5002,1002).
```

Fig. 3. Underlying logic representations of the LAN simulation before refactoring.

Table 2. encapsulateAttribute(*'Lan','Packet',originator*).

Primitive refactorings	FGT-lists
`addGetter ('Lan','Packet',originator)`	`FGT1: addObject(Lan,Packet,getoriginator,_,_,type (complex,Node,0), public,[], method)` `FGT2: addRelation(_,Lan,Packet,getoriginator,_,[], method, Lan, Packet, originator,_, _, attribute,read)`
`addSetter ('Lan','Packet',originator)`	`FGT3: addObject(Lan,Packet,setoriginator,_,_,type (basic, void,0), public,[(p, type(basic,Node,0))], method)` `FGT4: addRelation(_,Lan,Packet,setoriginator,_,[Node], method,Lan, Packet,originator,_,_,attribute,write)`
`attributeReadsToMethodCall ('Lan','Packet',originator, 'Lan','Packet',getoriginator, [])`	`FGT5: deleteRelation(_,Lan,Workstation,accept,_, [Packet],method, Lan,Packet,originator,_,_, attribute,read)` `FGT6: addRelation(_,Lan,Workstation,accept,_,[Packet], method,Lan, Packet,getoriginator,_,[],method,call)`
`attributeWritesToMethodCall ('Lan','Packet',originator, 'Lan', 'Packet',setoriginator, ['Node'])`	`FGT7: deleteRelation(_,Lan,Workstation,originate,_, [Packet],method, Lan,Packet,originator,_,_, attribute,write)` `FGT8: addRelation(_,Lan,Workstation,originate,_, [Packet],method, Lan,Packet,setoriginator,_, [Node],method,call)`
`changeAttributeAccess ('Lan', 'Packet',originator, private)`	`FGT9: changeOAMode(Lan,Packet,originator,_,_, attribute,public, private)`

primitive refactorings **attributeReadsToMethodCall** and **attributeWrites-ToMethodCall** to re-direct all existing accesses to the *originator* attribute via calls to these methods. Finally, it uses the **changeAttributeAccess** primitive refactoring to make the access mode of *orginator* private.

The second column of Table 2 shows the FGT-lists that are associated with each of these respective primitive refactorings. Thus, from an FGT perspective, adding a getter for the *originator* attribute in class *Lan.Packet* entails the execution of FGT1 and FGT2.

Note that the procedure (Prolog rule) for converting the **addGetter** primitive refactoring into an FGT-list, has to infer dynamically from the current context two parameters of the FGT-list.

- The name of the getter method has to be inferred from the attribute name. To do this, the procedure **concat**(*'get', Attn, Methn*) is invoked to concatenate the word '*get*' with the attribute *Attn* (in this case, *originator*) and return *Methn* (in this case *getoriginator*).
- In addition, a procedure has to be invoked to infer from the system description that *originator*'s type (which is **type**(*complex, Node,* 0) in this case).

Other FGT parameters can be hard-coded into the rule for converting the refactoring into an FGT-list.

- It is known *a priori* that the object to be added is a *method* with an empty list [] of parameters, and a *public* access mode. These are the last three parameters of FGT1.
- It is also known *a priori* that a *read* relation should be added from *Lan.Packet*'s newly installed *getorignator* object *method* with parameter list [], to *Lan.Packet's attribute* called *originator*. These parameters can be seen in FGT2.

Dual remarks apply to FGT3 and FGT4 on the FGT-list of the **addSetter** primitive refactoring. The principle difference between the FGTs for the setter and getter methods is that the former require [*Node*] for the parameter list and void (expressed as **type**(*basic, void,* 0)) for the return type, whereas the latter have [] as parameter list and **type**(*complex, Node,* 0) as return type.

The primitive refactoring **attributeReadToMethodcall** specifics that *every* read relations from a source object to the *originator* attribute in the *Lan.Packet* class must be deleted. Instead, a *call* relationship should be installed between each such source object and the *getoriginator* method in the *Lan.Packet* class — a method whose parameter list is [].

In general, building an FGT-list for this primitive refactoring involves a search through the entire system description for *read* relations terminating in *originator*, and detecting the relevant source object. For each such relation found, two FGTs must be generated, one to delete the *read* relation, and another to add the *call* relation. In the present context, only one *read* relation to *originator* is to be found.

This is from the *accept* method with parameter list [*Packet*] in the class *Lan.Workstation* to *originator*. The consequence of this mapping is the FGT-list consisting of FGT5 and FGT6.

Again, dual remarks apply to FGT7 and FGT8 on the FGT-list of the **attributeWriteToMethodcall** primitive refactoring.

Finally, the access mode of the *orginator* attribute in *Lan.Packet* class has to be made *private* using **changeAttributeAccess**. This primitive refactoring maps to the single FGT9. In instantiating this FGT, the only item directly inferred by the Prolog prototype from the context is that the current access mode is *public*.

The next composite refactoring to be invoked is **createClass**. The parameters used in this invocation are shown in the caption of Table 3. They indicate that a *public* class, *Lan.Server*, is to be installed as a subclass of *Lan.Node* and as a superclass of *Lan.FileServer* and of *Lan.PrintServer*. Again, the first column of Table 3 shows the constituent primitive refactorings required to do this, and the second column indicates their associated FGT-lists.

The **addClass** primitive refactorings maps directly to FGT1 without any complications. Then the primitive refactoring **changeSuper** is executed three times, mapping first to FGT2, then to FGT3 and FGT4, then to FGT5 and FGT6. In each case, the parameters of **changeSuper** indicate the subclass concerned (*Lan.Server*, *Lan.FileServer* and *Lan.PrintServer* respectively) and its new superclass (*Lan.Node*, *Lan.Server* and *Lan.Server* respectively).

The general rule for mapping this primitive refactoring to FGTs is that any existing *extends* relation from the named subclass to its superclass should be removed (using a **deleteRelation** FGT); and that a new *extends* relation, having an *is a* label, should be installed between the subclass and its designated new superclass (using an **addRelation** FGT). Since the newly installed *Lan.Server* class initially

Table 3. createClass(*'Lan','Server',public,'Lan','Node',*[*'Lan','FileServer','Lan','PrintServer'*]).

Primitive refactorings	FGT-lists
`addClass('Lan','Server',public)`	`FGT1: addObject(Lan,Server,_,_,_,public,_, class)`
`changeSuper('Lan','Server','Lan','Node')`	`FGT2: addRelation(isa,Lan,Node,_,_,_,class, Lan, Server,_,_,_,class,extends)`
`changeSuper('Lan','FileServer', 'Lan', 'Server')`	`FGT3: deleteRelation(_,Lan,Node,_,_,_,class, Lan, FileServer,_,_,_,class,extends)` `FGT4: addRelation(isa,Lan,Server,_,_,_, class, Lan, FileServer,_,_,_,class, extends)`
`changeSuper('Lan','PrintServer','Lan', 'Server')`	`FGT5: deleteRelation(_,Lan,Node,_,_,_,class, Lan,PrintServer,_,_,_,class,extends)` `FGT6: addRelation(isa,Lan,Server,_,_,_, class, Lan, PrintServer,_,_,_,class, extends)`

has no superclass, only FGT2 is required in its case, where as **deleteRelation** (FGT3 and FGT5) is required for the latter two **changeSuper** refactorings.

The final step involves the execution of a single primitive refactoring, **pullUpMethod**. The parameters of this procedure, as seen in the caption of Table 4, indicate that its task is to pull up the *accept* method (which has a single parameter of type *Packet*) from the classes *Lan.FileServer* and *Lan.Printer* to their common superclass. Although the superclass is not named in the **pullUpMethod** parameters, it is easily inferred from the current system description to be the newly created *Lan.Server* class. The three FGTs that ensure this pulling up are shown in Table 4. The parameters of FGT1, **addObject**, indicate that the object to be added to the class *Lan.Server* is a *method* called *accept* that returns a void type (***type***(*basic*, *void*, ()) and that has a single parameter called *p* whose type is *Packet* (***type***(*complex*, *Packet*, 0)). The next two FGTs indicate that this *accept* method is to be deleted from the classes *Lan.FilesServer* and *Lan.PrintServer* respectively.

5.2. *Preconditions and FGT execution order*

There are various ways in which the FGTs listed in Tables 2, 3 and 4 can be applied to the underlying logic-terms representing the LAN system shown in Fig. 3.

(1) *One FGT at a time*: The most naive approach is to handle each FGT separately, starting from the first in the first table, and going through to the last; and then doing the same for the FGTs listed in the next two tables. In each case, handling an FGT entails retrieving its precondition and checking each precondition conjunct against the data in Fig. 3. In addition, if the FGT is the first in an FGT-list, then any stored refactoring-level precondition conjuncts (referred to in Sec. 4.4) also have to be retrieved and checked against the data in Fig. 3. If all these precondition conjuncts are satisfied, the data in Fig. 3 is changed according to the FGTs specifications.

(2) *An FGT-list at a time*: Alternatively, the precondition of the FGT-list corresponding to each primitive refactoring can be pre-computed without reference to the data in Fig. 3. For each FGT-*list*, these conjuncts, as well as any stored primitive refactoring-level precondition, are then checked against the data in Fig. 3. If the system complies with these preconditions, then each FGT in the list can be applied without further checking their individual preconditions.

Table 4. pullUpMethod refactoring ([*'Lan'*,*'FileServer'*, *'Lan'*,*'PrintServer'*],*accept*,[*'Packet'*]).

FGT-list
`FGT1:addObject(Lan,Server,accept,_,_,type(basic,void,0),public,[(p, type( complex, Packet,0))],method)`
`FGT2: deleteObject(Lan,FileServer,accept,_,[Packet],method)`
`FGT3: deleteObject(Lan,PrintServer,accept,_,[Packet],method)`

(3) *An FGT-DAG set at a time*: The **build-FGT-DAG** algorithm outlined in Sec. 4.3 can be applied to the various FGT-lists. Again the precondition of this set can be pre-computed without reference to the data in Fig. 3. Indeed the precondition of the set of FGT-DAGs will be the same as the precondition for the corresponding FGT-list. Again, any stored refactoring-level precondition should also be checked.

(4) *An FGT-DAG set per composite refactoring*: In [9] it is shown how to merge the sequence of FGT-DAG sets corresponding to the list of primitive refactorings that make up a composite refactoring. The result is a new set of FGT-DAGS, which also has its own precondition. Again this precondition can be pre-computed without reference to Fig. 3 and stored for future application of the composite in other contexts. Again, too, refactoring-level preconditions should be stored checked before applying the refactoring in the order dictated by the FGT-DAGs.

To illustrate the third approach, consider the first primitive refactoring in Table 2, namely **addGetter**(*Pn,Cn,Attn*). Applying the algorithm **build-FGT-DAG** in Sec. 4.3 to its constituent FGTs (namely FGT1 and FGT2 in Table 2) would yield the FGT-DAG set that consists of only one FGT-DAG. Its root would be FGT1, with one child FGT2. The preconditions of FGT1 and FGT2, stated in generic terms in Sec. 3.2, are given again below for convenience. However, variables used in Sec. 3.2 that refer to specific characteristics adding a getter method, have been suitably instantiated. Also, to emphasise that the *Methn* variable used above is mapped to the concatenation of the string "get" and the instantiated value of *Attn*, the variable *GetAttn* is used below:

(1) FGT1 precondition conjuncts:

   (a) The class *Pn.Cn* should be already declared in the system.
   (b) *GetAttn* should be unique in *Pn.Cn*
   (c) *GetAttn* should not be in an ancestor class of *Pn.Cn*.
   (d) The access mode *public* of *GetAttn* should be valid (e.g. no public method in a private class).
   (e) The return value's definition type, *ODefT*, should be valid
   (f) The type of the return value *ODefT* should be accessible.

(2) FGT2 precondition conjuncts:

   (a) Method *Pn.Cn.GetAttn* with parameters *[]* should be declared in the system
   (b) Attribute *Pn.Cn.Attn* should be declared in the system
   (c) Relation *read* between the two objects does not exist
   (d) If the access mode of *Attn* is private then *GetAttn* should be in the same class as *Attn*.
   (e) If the access mode of *Attn* is default then *GetAttn* should be in the same package as *Attn*.

(f) If the access mode of *Attn* is protected then *GetAttn* should be in the same package as *Attn* or in one of the subclasses of the class *Pn.Cn*

(g) If the access mode of *Attn* is public then *GetAttn* can be anywhere.

It is immediately evident that conjuncts of (2.a) and (2.c) of FGT2 are satisfied by the postcondition of FGT1: FGT1's task is specifically to declare *Pn.Cn.GetAttn* with parameters *[]* in the system; and since it does not add any relation information "Relation *read* between the two objects does not exist". Similarly since *GetAttn* is specifically installed in class *Pn.Cn*, conjuncts (2.d) to (2.g) of FGT2 are satisfied, irrespective of *Attn*'s access mode. Thus, the precondition of the set of FGT-DAGs corresponding to the **addGetter** (*Pn,Cn,Attn*) primitive refactoring consists of the conjuncts (1.a) to (1.f) as well as 2.2. These can be permanently stored, and appropriately tested against the data for a particular given system, before refactoring that system.

In the current example, *Pn.Cn* corresponds to *Lan.Packet* and *Attn* corresponds to *orginator*. All the conjuncts are fulfilled by the data in Fig. 3, and so the primitive refactoring can be legitimately applied by applying FGT1 and the FGT2. This is the application sequence dictated by the underlying FGT-DAG set.

Several matters regarding the precondition of **addGetter** (*Pn,Cn,Attn*) deserve a brief mention. The precondition has been derived here in a "bottom-up" fashion from the associated set of FGT preconditions. The literature generally does not enumerate these conjuncts as fully and formally as given above as the precondition for **addGetter**. Nevertheless, of necessity they will have to be tested in any refactoring system that legitimately implements the primitive refactoring. This remark carries over to all primitive refactorings mentioned in the literature whose precondition could be derived along similar lines to those described above.

However, this observation needs to be qualified by noting that **addGetter** happened to be a primitive refactoring that did *not* incorporate what we have called a refactoring-level precondition. For each primitive refactoring's FGT-DAG set, the conjuncts of this precondition have to be inferred by asking: "Apart from the FGT-DAG set's precondition, derived in this bottom-up fashion, what else has to hold in order to preserve the system's behaviour?" This question can be answered from first principles, or by referring to published information about a primitive refactoring's preconditions.

The primitive refactoring **pullUpMethod**(*SubclassNames, Methn, MethTList*) that was used in the current example as described in Table 4 is an example of where refactoring-level preconditions do come into play. The FGTs associated with this refactoring merely add a method called *Methn* with parameter list *MethTList* to the common superclass of the classes mentioned in the list *SubclassNames*. The preconditions of these FGTs (and the resulting refactoring of the associated FGT-DAG set) have to do with local requirements for adding and deleting methods. They do not take into account specific requirements that apply

to pulling up a method. In particular, they do not specify the following two behaviour-preserving requirements:

(1) The access mode of the method *Methn* in the subclasses is may not be *private*.
(2) References made by *Methn* to the other *object* elements must be visible from the superclass.

These two conjuncts are part of the **pullUpMethod** prerequisites, and should be installed as refactoring-level prerequisites when using the FGT approach to implement this primitive refactoring.

Conversion of each FGT-list in Table 2 to sets of FGT-DAGs and subsequent checking of the associated refactoring-level- and FGT-DAG set preconditions against the system description in Fig. 3 would confirm that all precondition conjuncts hold. The application of FGTs in the order indicated by each of the specific FGT-DAGs would lead to the new system description given in Fig. 4. Note that numbered FGTs alongside logical terms indicate by which FGTs they have been changed.

Following this same procedure for FGT-lists in Table 3, and thereafter in Table 4 will change system's logic-based description as shown in Fig. 4. Figure 5 shows how this description would map to revised UML class diagram description (that includes call information as discussed above).

```
package(0,00,Lan,[53, 1, 2, 3, 4, 5]).
class(1,0,Node,public,[1001, 1002],[10001, 10002]).
method(1001,1,send,type(basic,void,0),protected,[100001]).
method(1002,1,accept,type(basic,void,0),public,[100002]).
attribute(10001,1,Name,type(basic,string,0),public).
attribute(10002,1,NextNode,type(complex,1,0),public).
parameter(100001,1001,p,type(complex,2,0)).
parameter(100002,1002,p,type(complex,2,0)).
call(1000001,_,1002,1001).
call(1000002,_,1001,53_90).
call(1000003,_,1001,53_91).
call(1000004,_,1001,5002).
read(1000008,_,1001,10001).
read(1000009,_,1001,10002).
extends(10000012,isa,1,5).
extends(54,isa,1,53).
class(2,0,Packet,public,[48, 46],[20001,20002,20003]).
method(46,2,getoriginator,type(complex,1,0),public,[]).
method(48,2,setoriginator,type(basic,void,0),public,[49]).
attribute(20001,2,contents,string,1,public).
attribute(20003,2,receiver,type(complex,1,0),public).
attribute(20002,2,originator,type(complex,1,0),private).
parameter(49,48,p,type(complex,1,0)).
read(47,_,46,20002).
write(50,_,48,20002).
```

```
class(3,0,FileServer,public,[53_90],[]).
method(53_90,3,accept,type(basic,void,0),public,[61]).
read(3000001,_,53_90,20001).
call(3000003,_,53_90,1002).
read(3000004,_,53_90,20003).
class(4,0,PrintServer,public,[53_91],[]).
method(53_91,4,accept,type(basic,void,0),public,[61]).
read(4000001,_,53_91,20001).
call(4000003,_,53_91,1002).
read(4000004,_,53_91,20003).
class(53,0,Server,public,[57],[]).
method(57,53,accept,type(basic,void,0),public,[61]).
parameter(61,57,p,type(complex,2,0)).
extends(55,isa,53,3).
extends(56,isa,53,4).
class(5,0,Workstation,public,[5001, 5002],[]).
method(5001,5,originate,type(basic,void,0),public,[500001]).
method(5002,5,accept,type(basic,void,0),public,[500002]).
parameter(500001,5001,p,type(complex,2,0)).
parameter(500002,5002,p,type(complex,2,0)).
call(5000002,_,5001,1001).
call(5000005,_,5002,1002).
call(51,_,5001,48).
call(52,_,5002,46).
```

Fig. 4. Underlying logic representations of the LAN simulation after refactorings.

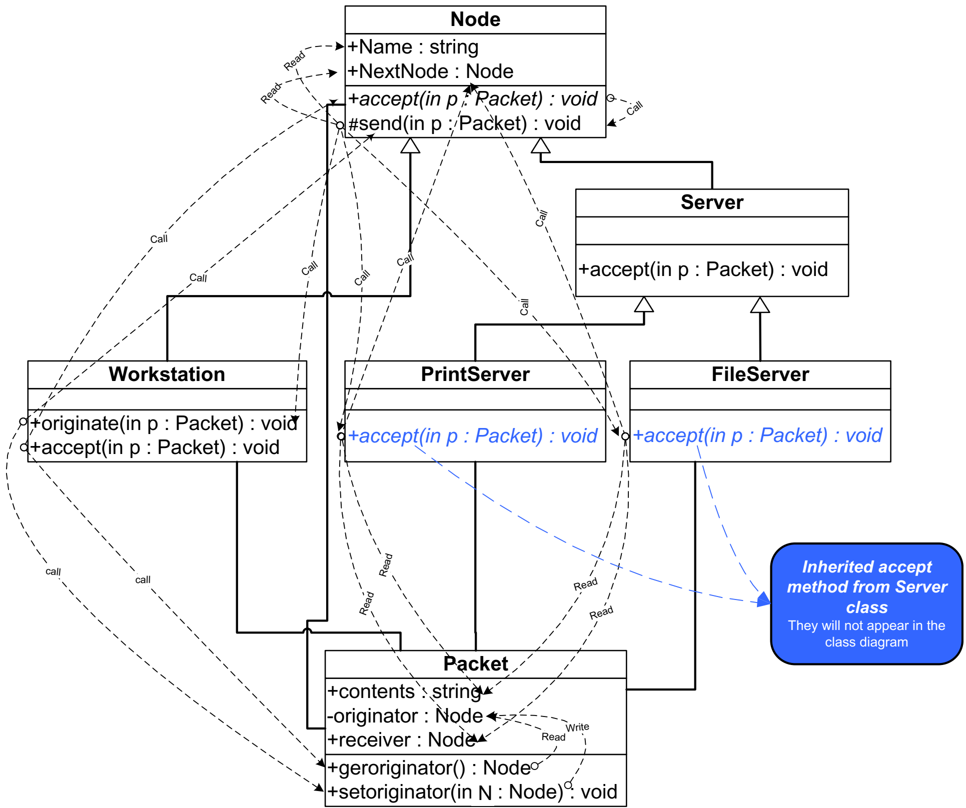


Fig. 5. A UML class diagram of the LAN simulation after refactoring.

6. Related Work

There appear to be two alternative approaches other than an FGT-based approach that used to build primitive refactorings. The first is the graph transformation approach [18−20] which relies on graph production rules from which primitive refactorings are constructed: successive application of appropriate rules result in the required refactoring of the system's graph representation.

Mens *et al.* use the graph rewriting formalism to prove that refactorings preserve certain kinds of relationships (updates, accesses and invocations) that can be inferred statically from the source code [21, 22]. Bottoni *et al.* describe refactorings as coordinated graph transformation schemes in order to maintain consistency between a program and its design when any of them evolves by means of a refactoring [23]. Qayum *et al.* [24] propose a local formulation of refactoring based on graph transformation. They use graphs to represent software architectures and graph transformation to formally describe their refactoring operations which makes it possible to use concepts and techniques from the theory of graph transformation such as unfolding and critical pair analysis to identify dependencies between refactoring steps.

Heckel [25] uses graph transformations to formally prove the claim (and corresponding algorithm) of Roberts [3] that any set of refactoring postcondition conjuncts can be translated into an equivalent set of precondition conjuncts. Van Eetvelde and Janssens [26] propose a hierarchical graph transformation approach to be able to view and manipulate the software and its refactorings at different levels of detail. Folli [27] discusses eight primitive model refactorings for UML Class diagrams and UML State Machine diagrams. He shows that it is possible to formalize the specification and execution of model refactoring using graph transformation rules.

Mens *et al.* [5] show how the **encapsulateAttribute** refactoring results from applying two graph productions rules successively. Another two productions rules (different from those used for the **encapsulateAttribute** refactoring) are needed to implement the **pullUpMethod** refactoring. As shown in Table 2, the former is a composite refactoring that is built up of five primitive refactorings and implemented by nine FGTs. Table 4 shows that the latter is a single primitive refactoring that is built up out of three FGTs. Clearly, therefore, FGTs are more fine-grained than the graph production rules. Indeed, this raises the interesting question of whether graph production rules can be devised to implement FGT transformations on program graphs. We conjecture that this would indeed be possible, but have not focused on this possibility for the present study.

The second approach to building primitive refactorings is to write one "monolithic" procedure to implement each primitive refactoring. Each procedure is formalized as conditional transformations with pre- and postconditions. A composite refactoring is then executed by successively invoking these procedures. The use of pre- and postcondition has been suggested repeatedly in research literature as a way to address the problem of behaviour preservation when restructuring or refactoring software artifacts. To automate the refactoring process, integrated development environments such as Eclipse [28, 29] and refactoring engines such as Huiqing [30] provides machine support for refactoring. These systems implement refactoring as atomic transformations guarded by preconditions. The underlying assumption is that if the precondition holds, the transformation will preserve behaviour. If the precondition does not hold, then the transformation are disallowed.

In the context of object-oriented database schemas (which are similar to UML class diagrams), Banerjee and Kim identify a set of invariants that preserve the behaviour of these schemas [31]. Opdyke adopts this approach to object-oriented programs, and additionally provides precondition conjuncts or enabling conditions for each refactoring [2]. He argues that this precondition preserves the invariants. Roberts uses first order predicate calculus to specify these precondition conjuncts in a formal way [3]. Kniesel [32] represents software as logic-terms and refactorings are formalized as conditional transformations with pre- and postconditions.

It is clear that reasoning in the previous approaches take place at the level of refactorings themselves, and attention is not paid to the detailed transformational steps that must be applied to the model to achieve the refactoring as in the FGT approach. Whenever a refactoring is applied, the hard coded sequence of statements

in the refactoring procedure is executed atomically. The inter-relationship between the different code statements both within and between refactoring procedures cannot be determined as in the FGT approach. On the other hand, the pre- and post-conditions in the previous approaches are defined manually -by the developer of the refactoring- at one level (refactoring level). While in the FGT approach, they are defined at two different levels as explained in Sec. 4.4 One of these two levels can be inferred automatically.

7. Conclusions, Limitations and Future Work

7.1. *Conclusions*

This paper shows how FGTs can be used to construct primitive refactorings (and thus also composite refactorings), principally at the UML class diagram design level. It defines and executes model refactorings as a set of FGTs ordered in one or more FGT-DAGs. The semantics of each FGT is specified in terms of its pre- and post-condition conjuncts. The paper introduces refactoring pre- and postcondition conjuncts at two different levels (FGT-level and refactoring-level). It shows that the preconditions for the primitive refactorings can be partially inferred from an analysis of the FGT preconditions, but that in some cases, so-called refactoring-level preconditions have to be added to these in order to guarantee behaviour preservation. Accordingly, the paper describes the three stages needed to apply a refactoring on a system: check refactoring-level precondition conjuncts, then check the enabled FGTs conjuncts and finally apply the FGTs according to their order in the DFG-DAGs.

The principle container for FGTs is an FGT-list in which the ordering of FGTs respects the sequential relationships between them. Such a list is characterised by the set of FGT precondition conjuncts (which a system should satisfy if the FGTs are to be sequentially applied to the system) as well as the resulting postcondition conjuncts (that describe the effect of applying the list).

An alternative container for FGTs is defined, called an FGT-DAG. It is a directed acyclic graph with FGTs as nodes, and with arcs that reflect the sequential dependency relationships between constituent FGTs. An algorithm named **build-FGT-DAG** is provided to convert a list of FGTs into a corresponding set of FGT-DAGs. Thus design level refactorings specified as FGT-lists can be also be converted to corresponding sets of FGT-DAGs.

Many advantages offered by the using of FGTs for constructing refactorings. These advantages, which are fully discussed in [15], may be briefly stated as follows:

(1) Where redundancy occurs between transformation operations that are carried out by the refactorings, the redundancy can be discovered and removed at the FGT level.
(2) In the case of conflict between two refactorings, the FGTs that cause the conflict can be discovered, and in some cases the conflict can be resolved without withdrawing one of the refactorings.

(3) Sequential dependency between two refactorings can be discovered at the FGT level.
(4) Composite refactorings of more than one refactoring can be composed in a way that will avoid roll-back problems. However, this is done by manipulating the ordering of FGT execution, rather than of refactoring execution.
(5) Parallel execution can be exploited at the FGT-DAG level. Thus, all FGT-DAGs in one refactoring can be executed concurrently because there is no sequential dependency between the FGT-DAGs.
(6) FGTs afford the user to define a wider range of refactorings than is possible when relying on primitive refactorings as building blocks. The semantics of the new refactorings are constrained, not by the selection of existing refactorings that have been implemented in the tool, but rather by the semantics of the FGTs that have been pre-defined in the tool.

The graph production rule approach can potentially share some of these features, provided that rules are devised to express transformations that are equivalent to the FGTs that have been identified. The extent to which this is possible is a matter for further study. However, the second approach to building primitive refactorings that we mentioned about — writing a "monolithic" procedure for implementing each primitive refactoring — does not allow for any of these features. In this case, the only building blocks available for reasoning about and constructing composite refactorings are the primitive refactorings themselves.

In order to test the applicability of the FGTs approach, a prototypical software tool is built. The different FGTs with their pre- and postconditions are fully defined and implemented in the tool. The tool also implements the different algorithms used in the approach. The FGTs defined inside the tool are then used to construct the twenty-nine commonly used primitive refactorings mentioned in Table 1.

7.2. *Limitations*

In general, there will be more FGTs than there are refactorings, and therefore more computational operations. However, this additional computational cost buys more flexibility — including, and especially, the flexibility afforded to the end user to define a wider range of refactorings than is possible when relying on primitive refactorings as building blocks.

In the contemporary world of high-speed processors, and the relatively small scale of entities to be processed in a design level refactoring applications (typically in the order of thousands rather than millions or billions) the additional processing cost does not seem to be a significant factor. Moreover, it should also be borne in mind that the additional computational cost can be offset against the enhanced scope for parallelizing operations afforded by the FGT paradigm, where one can rely on the ever-increasing number of multi-core processors available on contemporary chips.

7.3. *Future work*

While our work in this paper has shown that FGTs are sufficiently versatile to express any of the primitive refactorings mentioned in Table 1, it is a matter of further study to define FGTs that rely on the full UML class diagram vocabulary — abstract classes, interfaces, etc. In addition, the work in this paper was based on applying refactorings to UML class diagrams. It may be possible to extend the approach to a wider range of UML modeling notations such as state and sequence diagrams. It may also be possible to extend the approach to the code-level, to database schemas, to software architectures or to the software requirements' levels. A more thorough investigation into these possibilities is needed, both in terms of feasibility and in terms of desirability.

Throughout this work, the refactorings are reflected at the UML class diagram level. Ideally, the modifications should also be reflected on the other UML models affected by the refactoring, as well as on the code-level implementation of the system. This is because it is important to keep the different system models and code consistent with one another. Clearly, further research in this direction would be beneficial.

References

1. M. Fowler, *Refactoring: Improving the Design of Existing Code* (Addison-Wesley, 1999).
2. W. Opdyke, Refactoring Object-Oriented Frameworks, Ph.D. thesis, University of Illinois at Urbana-Champaign, 1992.
3. D. Roberts, Practical Analysis for Refactoring, Ph.D. thesis, University of Illinois at Urbana-Champaign, 1999.
4. D. Astels, Refactoring with UML, in *Proc. Int'l Conf. eXtreme Programming and Flexible Processes in Software Engineering*, 2002, pp. 67–70.
5. T. Mens, N. Eetvelde, S. Demeyer and D. Janssens, Formalizing refactorings with graph transformations, *Journal on Software Maintenance and Evolution* **17**(4) (2005) 247–276.
6. R. France and M. James, Multi-view software evolution: A UML based framework for evolving object-oriented software, in *Proceedings of the IEEE International Conference on Software Maintenance*, 2001.
7. I. Porres, Model refactorings as rule-based update transformations, in *Proceedings of UML 2003 Conference*, 2003, pp. 159–174.
8. G. Sunyé, D. Pollet, Y. Traon and M. Jézéquel, Refactoring UML models, in *Proceedings of International Conference of Unified Modeling Language*, 2001, pp. 134–138.
9. E. Saadeh and D. Kourie, Composite refactoring using fine-grained transformations, in *Proceedings of the South African Institute of Computer Scientists and Information Technologists (SAICSIT 2009)*, pp. 22–29.
10. E. Saadeh, D. Kourie and A. Boake, An algorithm for ordering refactorings based on fine-grained model transformations, in *Proceedings of 7th International Conference on Software Methodologies, Tools and Techniques*, 2008, pp. 225–243.
11. E. Saadeh, D. Kourie and A. Boake, Fine-grain transformations to refactor UML models, in *Proceedings of the Warm Up Workshop for ACM/IEEE ICSE 2010*, pp. 45–51.
12. E. Saadeh, D. Kourie and A. Boake, Model refactorings as logic-based fine-grained transformations, in *Proceedings of the 9th African Conference on Research in Computer Science and Applied Mathematics*, 2008, pp. 703–710.
13. JTransformer homepage, http://roots.iai.uni-bonn.de/research/jtransformer/.

14. T. Mens, G. Taentzer and O. Runge, Analyzing refactoring dependencies using graph transformation, *Software and Systems Modeling*, 2006.
15. E. Saadeh, Fine-Grain Transformations for Refactorings. Ph.D. thesis, University of Pretoria, South Africa, 2009.
16. G. Kniesel and H. Koch, Static composition of refactorings, *Science of Computer Programming*, 2004, pp. 9–51.
17. S. Demeyer, The LAN-simulation: A refactoring teaching example, *Int. Workshop on Principles of Software Evolution (IWPSE)*, 2005, pp. 123–134.
18. J. Cuny, H. Ehrig, G. Engels and G. Rozenberg, Graph grammars and their application to computer science, Lecture Notes in Computer Science, Vol. 1073, 1996.
19. H. Ehrig, G. Engels, J. Kreowski and G. Rozenberg, Theory and application to graph transformations, Lecture Notes in Computer Science, Vol. 1764, 2000.
20. G. Engels, E. Hartmut and G. Rozenberg, Special issue on graph transformations, *Fundamenta Informaticae* **26**(3, 4) (1996).
21. T. Mens, S. Demeyer and D. Janssens, Formalising behaviour preserving program transformations, in Graph Transformation, Lecture Notes in Computer Science, Vol. 2505, 2003, pp. 286–301.
22. T. Mens, On the use of graph transformations for model refactoring, in *Generative and Transformational Techniques in Software Engineering*, 2005, pp. 67–98.
23. P. Bottoni, F. Parisie and G. Taentzer, Coordinated distributed diagram transformation for software evolution, *Electronic Notes in Theoretical Computer Science* **72**(4) (2002).
24. F. Qayum and R. Heckel, Local search-based refactoring as graph transformation, in *Proceedings of the 2009 1st International Symposium on Search Based Software Engineering (SSBSE '09)*, pp. 43–46.
25. R. Heckel, Algebraic Graph Transformations with Application Conditions, M.S. thesis, TU Berlin, 1995.
26. N. Eetvelde and D. Janssens, A hierarchical program representation for refactoring, in *Proceedings of the UniGra'03 Workshop*, 2003.
27. A. Folli, UML Model Refactoring Using Graph Transformation, Master's Thesis, Institut d'Informatiqu, Université de Mons-Hainaut, 2007.
28. A. Kiezun, Advanced refactoring in Eclipse: Past, present and future, in *First Workshop on Refactoring Tools*, Berlin, 2007. https://netfiles.uiuc.edu/dig/RefactoringWorkshop/Presentations/AdvancedRefactoringInEclipse.pdf.
29. J. Shavor, *The Java Developers Guide to Eclipse* (Addison-Wesley, 2003).
30. L. Huiqing, Tool support for refactoring functional programs, in *ACM Sigplan Haskell Workshop* (2003), pp. 27–38.
31. J. Banerjee and W. Kim, Semantics and implementation of schema evolution in object-oriented databases, in *Proceedings of the SIGMOD Conference*, ACM, 1987.
32. G. Kniesel, A logic foundation for conditional program transformations, Technical report no. IAI-TR-2006-01, 2006.