Model Refactorings as Logic-Based Fine-Grain Transformations

Emmad Saadeh* — Derrick G Kourie —Andrew Boake

Department of Computer Science,
University of Pretoria,
Lynwood Road, Pretoria, SOUTH AFRICA
emmad_saadeh@aauj.edu, dkourie@cs.up.ac.za, Andrew.Boake@absa.co.za

RESUME. Le Refactoring est un processus permettant d'améliorer la structure interne d'un system logiciel tout en conservant ses caractéristiques et comportements externes. La tendance actuelle est d'appliquer le refactoring aux niveaux d'abstractions supérieures a celle de la codification (programmation). Dans cet article, nous proposons les transformations "fine-grain" qui représentent une nouvelle approche pour définir et exécuter les refactorings d'un model. L'approche est basée sur un ensemble prédéfinit des transformations "fine-grain" (FGTs) qui supporte l'évolution de model, et s'appui sur la logique de représentation du model UML de base. La méthode présente plusieurs avantages au delà des approches précédentes, et garantit l'homogénéité entre les différents diagrammes UML.

ABSTRACT. Refactoring is the process of improving the internal structure of a software system while preserving its external behavior. The current trend is to apply refactoring at levels of abstraction higher than the code level. In this paper, we propose a new approach to define and execute model refactorings as fine-grained transformations. It is based on predefined set of fine-grain transformations (FGTs) that support model evolution, and relies on logic-based representation of the underlying UML model. The approach has many advantages over the preceding ones and guarantees consistency between the different UML diagrams.

MOTS-CLES: transformation fine-grain, dépendance séquentielle, processus de tassement

KEYWORDS: fine-grain transformation, sequential dependency, compaction process

1. Introduction

In [1, 2], Roberts defines refactorings as program transformations containing particular preconditions that must be verified before the transformation can be applied. Base on this definition, the different model refactoring approaches look at refactoring as a model transformation with pre and post conditions that the model must satisfy in order to apply the refactoring.

Figure 1. *Refactoring as black box*

In the refactoring literature, there are two main approaches to represent refactorings. The first one is based on graph theory and uses graph-matching mechanisms for refactoring [3, 4]. The second one is based on a conditional transformation mechanism: before being applied, transformations are guarded by preconditions that need to be satisfied [5, 6]. In both of the approaches, refactoring is a black box that, if applicable, will translate the model from one state to another. There is no concern of the detailed steps internally require during the refactoring. Such a dealing with refactorings (as black box) causes many problems and shortcomings in refactoring tools:

1. Sequential dependencies are found between refactorings as a whole and not between the components of refactorings; this reduces the potential for parallelism when applying the refactorings (keeping in mind the special dealing with composite refactorings).
2. Where conflicts occur between two refactorings, it is difficult to know which part of the two refactorings caused the conflicts. This will make the process of resolving the conflicts more difficult.
3. The pre and post conditions of refactoring sometimes do not give the exact idea about what the refactoring did. As a simple example, suppose that refactoring X has as preconditions : (*Object A found, Object B not found*) and as postconditions (*Object A not found, Object B found*). Here X could be one of two possible refactorings: X is (**RenameObject**(*A, B, type*)); or X is (**DeleteObject**(*A, type*), **AddObject**(*B, type*)).
4. A “Compaction” process that is used to remove redundancy between the refactorings does not work when dealing with refactoring as a black box.
5. Each refactoring inside the refactoring tool is implemented, tested and saved, to be used by the end users. Such a procedure has to be repeated for every new refactoring. This is difficult because the list of possible refactorings is unbounded; every day a new refactoring is proposed.
6. It is difficult for the end user to build new refactorings because there is a need to write a code.

In order to address problems such as these, our refactoring approach uses a different paradigm; we called it FGT-Based approach. We predefine a set of fine-grain transformations (FGTs) that will be the basis for the construction of any refactoring.

The set is sufficiently complete to generate any modification on the model. As a result, it is easy and straightforward to control the effects, find the dependencies, detect and resolve the conflicts, remove the redundancies between the different FGTs and give the end user the possibility to build his own refactoring with out a need to write any code.

The remainder of the paper will be structured as follows. Section 2 presents the logic-based underlying representation of the UML model. Section 3 defines the concept of using fine-grain transformation to construct model transformations. Related work is discussed in Section 4. Section 5 concludes this paper.

2. Logic-Based underlying representation of the model

Our approach relies on representing all relevant elements in a UML model as logic terms called Model Element Terms (METs). Each MET expresses the semantics of standard UML [7] modeling vocabulary related to Objects and Relations. These METs are represented as Prolog facts so as to benefit from Prolog's powerful search engine and backtracking techniques. The algorithm produces two kinds of METs as shown in Table (1).

Category	METs	Description
Object METs	**PackageMET (** *PackageID, PackageName,* [*ClassIDs*] **)**	*ClassIDs* refers to the list of IDs of class METs inside the package
	ClassMET (*ClassID, PackageID, ClassName, Modifier* , [*MethodIDs*], [*AttributeIDs*]**)**	*PackageID* refers to the package where the class located
	MethodMET (*MethodID, ClassID, MethodName, ReturnType, Num, Modifier*, [*ParamIDs*] **)**	*ReturnType* refers to the return type of the method. *Num>1* return type is array *Num=1* return type is primitive type or class object *Num=0* return type is void
	AttributeMET (*AttributeID, ClassID, AttributeName, AttributeType, Num , Modifier* **)**	
	ParameterMET (*ParamID, MethodID, ParamName, ParamType, Num* **)**	
Link METs	**LinkMET (** *LinkID, label, FromID, ToID, LinkType* **)**	Represent the relation between two objects in the model (*FromID* and *ToID*).*LinkType* can be: generalization, association, call, update, access and definition type.

Table 1. *Underlying set of Model Element Terms (METs)*

In order to simplify the presentation of our approach, we restrict ourselves to the simplified UML meta-model shown in Figure (2). Thus, we do not consider here interfaces, abstract classes, abstract methods, aggregations and so on.

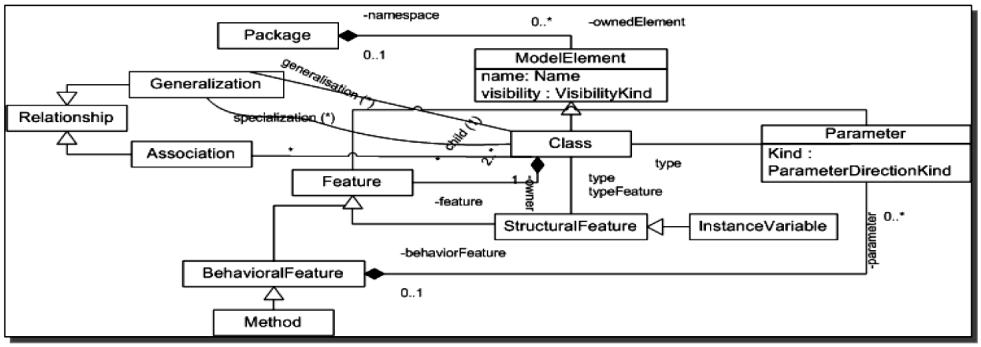

Figure 2: S*implified UML Meta-Model*

3. Fine-Grained Transformations

The main idea of our approach is to define a set of fine-grain transformations (FGTs) which will be the basis for the construction of any model transformations. A FGT is derived from the general actions that can be performed on an element of a UML model. We then build a refactoring using a sequence of these low-level "atomic" operations, rather than a piece of dummy code.

The set of FGTs to be used in our approach are shown in Table (2). In addition, our approach also assumes a list of auxiliary functions and constructs (such as **GETTYPE**(), **GETMODE**(), **CONCATENATE**(), **IF** statement, **LOOP**, etc). These are not transformation operations and are not included in the R-DAGs or in the compaction process, as we will explain later.

FGTs Operations	Description
AddObject (*objectname, type, num, modifier, objecttype*)	Add class/method/attribute/parameter object to the model
RenameObject (*oldobjectname, objecttype, newobjectname*)	Rename class/method/attribute/parameter object
ChangeObjectAccessMode (*objectname, oldmode, newmode*)	Change object access mode (public/private/protected)
ChangeObjectDefType(*objectname, oldtype, newtype*)	Change object type definition (class,int,float,long, String, …)
DeleteObject (*objectname, object type*)	Delete class/method/attribute/parameter object
AddLink(*name, fromobject, ftype, toobject ,totype, ltype*)	Add gen/assoc/call/update/access/type link to the model
RenameLink(*oldname, fromobject,ftype, toobject ,ttype, ltype,newname*)	Rename gen/assoc/call/update/access/type link
DeleteLink(*name, fromobject,ftype, toobject ,ttype, ltype*)	Delete gen/assoc/call/update/access/type link

Table 2. *Proposed set of predefined Fine-Grain Transformations (FGTs)*

Building a refactoring as a set of FGTs has great advantages over approaches that deal with refactorings as a black box:

1. End users are able build their own refactoring by using a sequence of FGTs. There is no need to write any code: the user merely selects the operations needed to construct a refactoring. The built refactoring will be given a name, list of input parameters, and can be saved in the refactoring tool for a later use. Figure (3) below shows how to build a refactoring called CreateClass that is used to insert an empty class between some superclass and a list of n subclasses.

```
CreateClass ( class, modifier, superclass, sublist [sub1, sub2, .., subn] )
1. AddObject (class, null, null, modifier, "class" )
2. AddLink ("isa", superclass,"class", class, "class", "generalization" )
3. Loop {   DeleteLink ( -, -,-, sublist[i],"class","generalization")
            AddLink ("isa", class,"class", sublist[i],"class","generalization") }
```

Figure 3. *CreateClass Refactoring*

As another example, Figure (4) shows how to build a refactoring called **EncapsulateAttribute** that is used to encapsulate an atrribute inside a class.

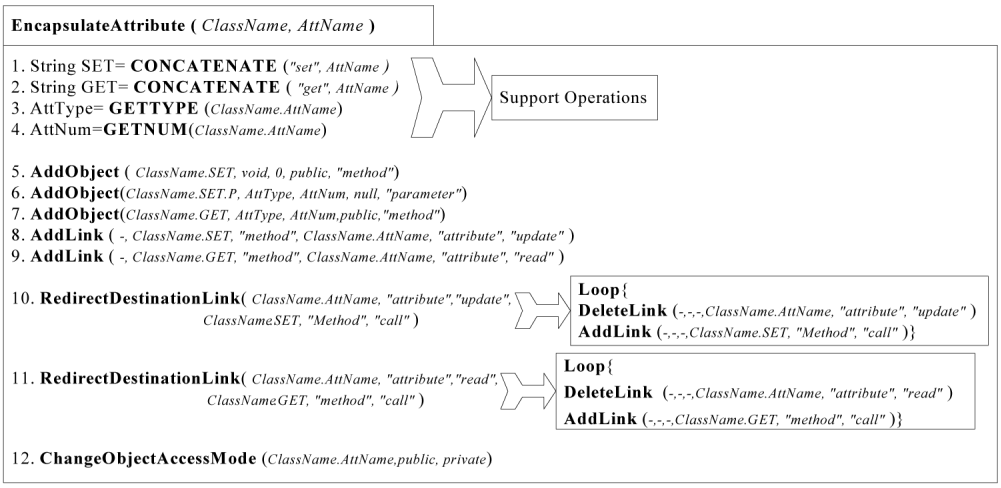

Figure 4. *EncapsulateAttribute Refactoring*

Giving the end user the ability to construct their own refactorings is a powerful feature of our approach because, as we mentioned before, the list of possible refactorings is infinite; and no tool vendor can support the end users with all their needs.

2. The approach manages sequential dependencies and conflicts at the level of FGTs rather than at the level of the whole refactoring. This has a great benefit in finding the optimal order, detecting and resolving conflicts, removing redundant operations between refactorings and merging two different sequences of refactorings. Although a detailed discussion of these algorithms is beyond the scope of this paper, relying on the semantics of FGTs, we have catalogued the various possibilities for sequential dependencies as in Figure (5) and conflicts as in Figure (6) between FGTs. Our approach provides a mechanism to resolve (automatically or interactively) all these type of conflicts.

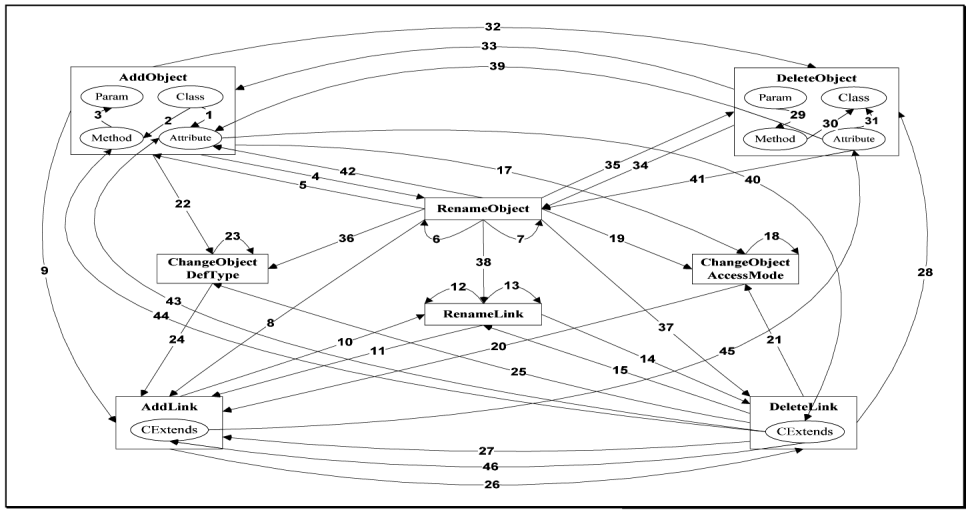

Figure 5. *Possible sequential dependencies between FGTs.*

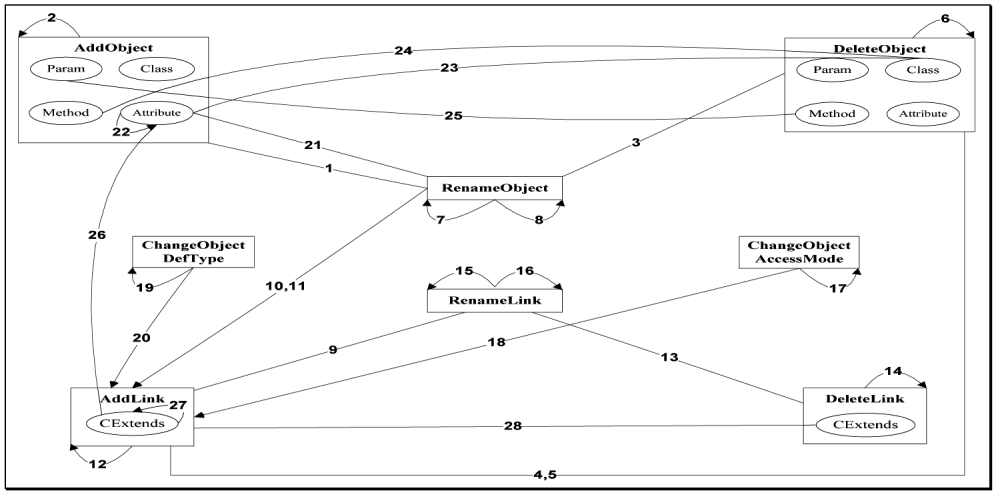

Figure 6. *Possible conflicts between FGTs.*

3. The FGT approach increases the opportunities for parallelising FGTs in one or more refactorings. It involves inserting sequentially dependent FGTs into the same data structure, called Refactoring Dependency Acyclic Graph (R-DAG) while maintaining a mapping of the original refactoring (primitive or composite) to which each FGT relates. FGTs of a refactoring (especially the composite ones) may be inserted in more than one R-DAG. However, since FGTs in different R-DAGS are guaranteed to be independent of one another, the R-DAGs can be applied concurrently on a model.
4. The approach can also remove redundancies between a sequence of FGTs. The final effect of the operations on the model after the compaction process is the same as the effect of the operations without any compaction. In general, the compaction process increases the efficiency of the refactoring algorithm by reducing the number of FGTs that need to be applied on the model. Mens [8] talks about the basis of the compaction process. He considered as future work the task of increasing the efficiency of his algorithm and dealing with composite refactorings. In our work, we have developed a new algorithm that increases the compaction process efficiency and deals with the problem of composite refactorings. Although a detailed discussion of our compaction algorithm is beyond the scope of this paper, it should be clear that it involves, inter alia, the systematic application of information in figures 5 and 6 to produce R-DAGs, which are then reduced in relation to certain FGT-based rules.

4. Related Works

The current research trend is to apply refactoring at levels of abstraction higher than the code level. Suny_e et al [9] have provided a fundamental paradigm for model refactoring to improve the design of object-oriented applications. They use OCL pre and post conditions. In [10], Porres implemented refactorings as a collection of transformation rules. In [11] Mens represented refactorings as graph transformations and used AGG as an experimental tool. In [12, 13] Kniesel uses CT-based refactorings and use Condor as an experimental tool.

Our approach of using a predefined set of FGTs to represent the refactorings is closely related to the work on Reuse Contracts [14]. In [8, 15], Mens provides a formalism for Reuse Contracts to express graph transformations.

5. Conclusion

The main contribution of this paper is to define and execute model refactorings as a set of FGTs. Sequential dependencies, conflicts and compaction opportunities between each pair of FGTs are specified. The approach relies on an R-DAG as a data structure to store the set of FGTs in one refactoring in a way that reflects the sequential dependencies between these FGTs. It relies on logic-based representations of the underlying UML model. The approach enjoys several advantages over the previous ones, as summarized in Table (3).

No.	Alternative approaches	Our approach
1.	Refactoring is a black box	Refactoring is a set of FGTs that is inserted in one or more R-DAGs
2.	Sequential dependencies are founded between refactorings as a whole	Sequential dependencies are founded between components (FGTs) of refactorings
3.	It is difficult to know which part of refactoring causes the conflict and difficult to resolve these conflicts	Conflicts are detected at the level of FGTs. The approach resolves these conflicts automatically or interactively
5.	Less parallelizing opportunities	More parallelizing opportunities
6.	Difficult for end users to build new refactoring that is not predefine before in the refactoring tool	Easy and straightforward by using the list of FGTs and the supported operations
7.	No ability to remove the redundant operations	Remove the redundant FGTs inside refactoring

Table 3. *A comparison between our refactoring approach and the other alternative ones*

All the above features result in an approach that is efficient, saves time and effort at when refactoring. As future work we plan to extend our simplified UML meta-model to deal with new constructs such as interfaces, aggregations, constructors, method overloadings and so on. We expect that new dependencies and conflicts between the different FGTs will be introduced, and ways will have to be found deal with these.

6. Bibliographie

[1] DB Roberts. *Practical Analysis for Refactoring*. PhD thesis, University of Illinois at Urbana-Champaign, 1999.

[2] DB Roberts, J Brant, and RE Johnson. A refactoring tool for smalltalk. *Theory and Practice of Object Systems 3*, pages 253-263, 1997.

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

[4] T. Mens, P. Van Gorp, D. Varr , and G. Karsai, "Applying a model transformation taxonomy to graph transformation technology," in Proc. Int'l Workshop on Graph and Model Transformation (GraMoT 2005), September 2005.

[5] G Kniesel. A logic foundation for conditional program transformations. *Technical report no IAI-TR-2006-01, ISSN 0944-8535, CS Dept III*, 2006.

[6] G Kniesel and H Koch. Static composition of refactorings. *Science of Computer Programming*, 52:9-51, 2004.

[7] Object Management Group, "Unified Modeling Language: Infrastructure version 2.0." formal/2005-07-05, August 2005.

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

[9] G Sunye, D Pollet, Y L Traon, and J-M Jezequel. Refactoring uml models. *Proceedings of UML 2001 Conference*, pages 138-148, 2001.

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

[11] G Taentzer. A graph transformation environment for modeling and validation of software. *Proc AGTIVE 2003*, 3062:446-453, 2004.

[12] G Kniesel and U Bardey. Static dependency analysis for conditional program transformations. *Technical report no IAI-TR-03, 2003*.

[13] U Bardey. *Abhangigkeitsanalyse fur Programm-Transformationenur Programm Transformationen*. PhD thesis, University of Bonn, 2003.

[14] P Steyaert, C Lucas, T Mens, and T DiHondt. Reuse contracts: Managing the evolution of reusable assets. *Proc OOPSLA 96*, pages 268-286, 1996.

[15] T Mens. Conditional graph rewriting as a domain independent formalism for software evolution. *Proc IntAgtive 99 Conference, LNCS 1779*, pages 127-143,2000.

[16] S Demeyer, F V Rysselberghe, T Girba, J Ratzinger, R Marinescu, T Mens, B D Bois, D Janssens, S Ducasse, M Lanza, M Riegerand H Gall, and M El-Ramly. The lan-simulation: A refactoring teaching example. *IWPSE05*, pages 123-134, 2005.