Ingénierie des Systèmes d'Information
Vol. 28, No. 2, April, 2023, pp. 263-274
Journal homepage: http://iieta.org/journals/isi

A Novel Method for Refactoring UML Metamodel

Berraouna Abdelkader

Department of Computer Science, University of Mohamed-Cherif Messaadia-Souk Ahras, Souk Ahras 41000, Algeria

Corresponding Author Email: a.braouna@univ-soukahras.dz

https://doi.org/10.18280/isi.280201

Received: 18 December 2022
Accepted: 10 February 2023

Keywords:
UML metamodel, refactoring, MDA, software artifacts, adaptation

ABSTRACT

UML metamodel, like other metamodel change through time as a result of changing needs and technical improvements during their life cycle. Adding new update or bug fixing can change UML metamodel, so potential inconsistencies with existing models that correspond to the previous version of the UML metamodel and may become non-compliant with the new version. In this approach, the refactoring facilitates a UML metamodel refactoring in well-defined steps from the basic features. The use of this refactoring allows extending the functionality of the existing UML metamodel. This research focuses on the methods and processes involved in adapting the UML metamodel to changing needs and technical improvements over time. The study highlights the potential for inconsistencies to arise from updates and bug fixing in the UML metamodel. The research methodology used is the refactoring of the UML metamodel through a well-defined process in well-defined steps. The study found that the refactoring process allows for the extension of the basic features of the UML metamodel and the introduction of new functionalities. The research concludes that the use of well-defined refactoring processes is essential in maintaining the evolution of the UML metamodel and ensuring its compliance with changing needs and technical improvements.

1. INTRODUCTION

In a realistic environment, metamodel must be adapted to their environment or to new requirements. As the UML metamodel is improved, it grows increasingly sophisticated and deviates from its original design as it is changed and adapted to new requirements, which decreases the UML metamodel quality. The consequence is that the majority of the costs of a UML metamodel are induced by maintenance.

Refactoring addresses this problem of increasing complexity by enhancing structural properties of the metamodel. The concept of refactoring was introduced by Opdyke [1].

Refactoring is the process of making changes to a software system that do not change the code's external behavior while enhancing its internal structure.

This paper presents a new method for UML metamodel refactoring, based on MDA techniques. UML metamodel refactoring uses rules that refine it according to the designer's intention.

UML metamodel refactoring can be handled in a variety of methods. A refactoring technique only executes one of several feasible adjustments [2, 3].

Designers can refine the UML metamodel if it does not meet their requirements. Therefore, refinement mechanisms are needed to refine existing UML metamodel when they do not reflect the exact intentions.

In other words, UML metamodel refactoring creates varying adjustments based on the kind of UML metamodel to be adapted, each adaptation being formalized in a library. Default libraries specify recurring adaptations, which can be changed to match specific needs.

A UML metamodel refactoring involves changing the attributes of an existing UML metamodel concept through assignments to its characteristics. The body of a rule can include various modifications to the metamodel.

The research on refactoring UML metamodel is significant because it aims to improve the quality and efficiency of software development processes by enhancing the Unified Modeling Language (UML), which is a widely used language for modeling software systems. Refactoring the UML metamodel involves restructuring its underlying structure to make it more flexible, maintainable, and aligned with current software development practices. The ultimate goal is to support developers and designer in creating better software designs and reducing the time and effort required to make changes to those designs. The value of this research lies in the potential improvement of software development processes, which can lead to better software products, faster time-to-market, and reduced costs. Additionally, a better-designed UML metamodel can increase the adoption and use of UML as a modeling language, leading to improved collaboration and communication among software development teams.

The rest of this paper is organized as follows: Section 2 presents some of objectives of refactoring method and motivation of this work. Sections 3, 4, 5, 6, 7 outline the related work and discussion of existing approaches and classification of these approaches. A section 8, 9 describes UML metamodel refactoring rules in this approach. Section 10 presents the implementation of refactoring rules and some examples of refactoring and will end with a conclusion and some perspectives [2].

2. OBJECTIVES OF THE PROPOSED REFACTORING METHOD

The goals and objectives to be reached for the refactoring method (with respect to the limits of the existing system) are the following:

- A level of abstraction for refactoring this level of abstraction will allow refactoring to be reusable and generic.
- The refactoring model must be open to the addition of new refactoring methods.
- The refactoring model must be able to take advantage of the features of object-oriented refactoring.
- A new refactoring can be defined by combining already defined refactoring (via inheritance or composition relationships).
- Refactoring usage must be flexible and easy to manage.

In this method the refactoring facilitates a UML metamodel refactoring in well-defined steps. From the basic features, new features are introduced to the UML metamodel. The use of this refactoring allows extending the functionality of the existing UML metamodel.

This refactoring facilitates a UML metamodel refactoring in well-defined steps. From the basic elements, new elements are introduced by the construct. The use of refactoring allows extending the functionality of the UML metamodel. In this way, the similar concepts are often explicit. The construction allows reusing these concepts by specialization. Generalization and specialization allow refactoring of the UML metamodel. By changing particular steps, UML metamodel designers can alternate designs.

This method is based on the principles of both object-oriented refactoring and model refactoring. By combining these two approaches a foundation for autonomous adaption of UML metamodel is established.

The method defines several rules to ensure the refactoring of the UML metamodel. They have been used to derive semantic conservation and instance properties of the UML metamodel refactoring. The concepts described here can also be applied to other structural descriptions. In addition, a set of rules has been developed to facilitate the automatic refactoring of the UML metamodel through progressive adaptation. Implementation and preservation rules have been developed for each refactoring.

3. RELATED WORK

This section gives a summary of related work on refactoring and automated detection of refactoring, the focus is on the most closely related approaches.

Refactoring is an essential process in software reengineering [1], which aims to reorganize existing software. Refactoring is merely the final stage in this process and the technical challenge of (semi-)automatically altering the software to incorporate a new solution. The most essential difficulties, however, are selecting which elements of the old software should be converted and how to convert them precisely, taking into consideration the limits encountered by the reengineers as well as the possible impact of the proposed modifications.

Refactoring also seems to fit well into a reengineering process driven by the MDA model. One purpose of model-driven architectures is to make UML metamodel refactoring easier.

Refactoring can be used to convert current model designs into a format that an MDA tool's reverse engineering capabilities can understand.

The work on refactoring has been directed from the beginning towards the transformation of object-oriented programs [4]; Opdyke [1] gives two reasons for this refactoring:

-Compared to more traditional development approaches, object-oriented programming facilitates refactoring because it makes the necessary structural information explicit.

-Refactoring is especially important in object-oriented programming.

In some cases, the best way to improve the design of a program is to rewrite it; in other cases, redesigning it may be easier.

Refactoring is described as a critical tool for controlling the refactoring of software by Brant and Roberts [5]. They claim that because typical waterfall development methodologies position maintenance at the end of the software life cycle, they fail to account for software evolution. They also point out that other spiral life cycle-inspired methods, such as Joint Application Development and, more recently, Extreme Programming, provide better support for software evolution.

These methods promote the use of fourth generation languages such as UML and integrated development environments, making them better suited for refactoring. Since UML appears to be more in line with the spirit of the first type of methods than the more agile methods.

Recent methodologies, such as Catalysis [6], which uses UML as a notation, consider the evolution of software, and consequently the evolution of design. Furthermore, because certain tools now allow the creation of design meta-models from source code, refactoring might be used to edit this code and improve the design of current programs. However, in fact, it is difficult to assess the true impact of changes on the many aspects of the design as well as the execution.

This is especially true for uml: Iits numerous structural and dynamic views might share many meta-model features; for example, when a method of a concept is destroyed, it is impossible to discern at first glance, without the assistance of a tool, what the method was.

Tokuda and Batory [7] define large architectural changes in the different settings as a long series of small redesigns. They assume that automated refactorings are ten times faster than manual refactorings. Recent refactoring research extends the analysis for automated refactorings with more successful methods.

Tip et al. [8] use type constraints to aid in the analysis of refactoring that introduce generalized statement.

Garces et al. [9] provide a set of heuristics to process automatically the equivalences and differences between two metamodel variants to adapt the models to their evolved metamodel and thus follow a correspondence co-evolution approach. The computed equivalences and differences are saved in a so-called adaptation model, which serves as input to a higher-order transformation HOT [10], creating an implementable transformation adaptation.

The method presented by Cicchetti et al. [11] is comparable to that of Garces et al. [9] in that it is once more focused on a metamodel representation of the difference that serves as an input for a higher order transformation. Additionally, the computed differences are divided into: (i) unbreakable changes, (ii) brittle and resolvable changes, (iii) brittle and insurmountable changes.

Wachsmuth [12] suggests fusing concepts from object-oriented refactoring and grammatical adaptation.

In this way, the definitions of instance preservation and semantic preservation are built upon the definitions of metamodel relations.

A group of transformations that are based on QVT relations are also suggested, and they are categorized as refactoring, construction, and reduction transformations.

In Models employing the Model Change Language (MCL) [13] are presented with a co-evolution approach. The relationships between the components of the various metamodel versions are defined by the evolver. Relationships can take many different forms, from straightforward one-to-one mappings between classes to more intricate mappings for adding items to new subclasses or modifying the confining hierarchy.

Herrmannsdoerfer et al. [14] in order to minimize the migration effort, COPE proposes an integrated way to define the linked development of métamodèles and models. In this regard, a collection of so-called linked transactions, which together make up a larger co-evolution issue of modular transformations, achieve the co-evolution of métamodèles and related models. In order to reduce the work required for migration, coupled transactions are further classified into custom coupled transactions and reusable coupled transactions. Reusable coupled transactions are those that are preset and do not require user input [15, 16].

EMF Compare [17] is a tool which can match, combine (two and three ways), and compare EMF/Ecore models. Instead of using distinctive identifiers, it employs a distance connection to match similar parts. UMLDiff [18] employs a distance relationship that takes structure and names into consideration and is also non-ID based. Unlike EMF Compare, which covers EMF/Ecore-based models like Ecore, UML, etc., UMLDiff focuses exclusively on UML models. The UMLDiff technique is expanded by DSMDiff [19] to handle domain-specific modeling languages.

Williams et al. [20] use a search-based methodology to compute a (near) optimum history model. As a result, the historical model will include a variety of possible modifications that might lead to a model changing from one edition to another. A fitness function that chooses the much more likely evolution was defined by the authors. Be aware that this method depends on an operator-based tool, the adapt tool, rather than directly detecting complicated changes. It provides a list of operators that may undergo atomic alterations as well as complicated ones. As a result, they concentrate more on identifying the ideal arrangement of operators to characterize the change.

Di Ruscio et al. [21] description of a language enables users to actively define the evolution's alterations. A different option is to evaluate electromagnetic fields. May be used to determine their list of modifications.

The Model Change Language (MCL) was developed by Levendovszky et al. [22] to allow for the specification of co-evolution and metamodel evolution techniques.

A matching rule known as an idiom looks for a right-hand side (RHS) in the evolved metamodel and a left-hand side (LHS) in the original metamodel. A modification is recognized if both are discovered. In order to identify both atomic and complicated changes.

Garcés et al. [23] suggest computing the difference using a number of heuristics represented as transformations in the Atlas Transformation Language (ATL). Garcés et al. [23] still don't fully understand the nature of the complicated alterations or how to spot them.

By comparing MFEs, Langer et al. [4] suggested that complicated alterations may be detected. They specify a complicated change using the left side (LHS) and the right side (RHS) of a graphical transformation (RHS). When LHS is present in the original version and RHS is present in the evolved version, a complicated change is discernible. Two snapshots of an original metamodel and its evolving form can be used to represent DHS and RHS. So, a complicated change's variability is not addressed.

Precision and recall were tested by Langer et al. [4] and averaged 98% and 70%, respectively. They want to make sure their detection is accurate despite the lack of some necessary adjustments.

In order to identify atomic changes, Garcia et al. [24] compare EMFs as a preliminary step. They then use predicates that look for instances of the atomic change class to detect complicated changes. For each complicated modification, the predicates are implemented as ATL transformation scripts. They suggest that the overlap issue can be somewhat solved by identifying the most difficult changes without the enclosed updates, which may result in fewer calls. They do not, however, take variable complicated modifications into consideration.

4. LIMITATIONS OF EXISTING WORK AND CONTRIBUTION

It has been discovered that further research is necessary to establish which refactoring techniques can be implemented, where and when in a meta-model driven reengineering process, and which other techniques are complementary to reach meta-model refactoring. This was the result of a thorough analysis of the existing work in meta-model refactoring. Narayanan's approach defines the MCL "Model change language" [12] using an MOF compliant metamodel. MCL is a high-level visual language to describe the evolution of the metamodel. MCL defines a set of idioms and a compositional approach for specifying migration.

Using a metamodel that complies with MOF, Narayanan's method constructs the MCL "Model change language" [12]. MCL is a high-level visual language for describing the métamodèles development. For describing migration, MCL specifies a collection of idioms and a compositional method.

The most typical metamodel evolution scenarios, such as introducing a new concept, modifying an element, removing an element, adding new subtypes, updating local models, and automating the migration of instance models, may all be specified using rules. For common migrating scenarios, MCL employed a basic model that consisted of a "Maps To" link between an LHS element from the old metamodel and an RHS element from the new metamodel. The model uses a different unique connection named "WasMappedTo" to identify a node; it already underwent migration due to a prior migration regulation. MCL is more effective since it gives a DSML domain-specific modeling language as a specification language, as opposed to the sprinkling approach's [13] generic program for the migratory. The MCL is expressive, modular, and enables for reuse of knowledge migration. It also offers a straightforward graphical syntax. MCL can also define intricate connections between meta entities. But in MCL, some rules must be manually resolved, and in other situations, the creator of the transformation's purpose must be taken into

consideration.

In contrast to the previously mentioned approaches, the proposed approach addresses metamodel evolution using already-in-use transformation languages rather than domain-specific or M2M transformation languages.

It's possible to avoid copying elements that are resilient to metamodel changes and are supported by COPE by using specialized metamodel merging algorithm. Additionally, unlike other systems that need manual development, this approach enables for the automated removal of obsolete model components that are no longer covered by the updated metamodel.

In contrast, utilizing the unified metamodel together with in place refactoring, the method just requires one metamodel to define evolution rules.

Last but not least, automated development of refactoring rules for disruptive and reversible modifications is favored.

5. UML METAMODEL REFACTORING

In this method, a UML metamodel refactoring has the objective to improve its design described. Refactoring may be viewed as a restructuring of the information included in the UML metamodel as a whole. The original UML metamodel gets turned into a portion of it. Since the input UML metamodel is modified in place, it is more effective to implement a refactoring as an update transformation.

5.1 Description and characteristics of UML metamodel concept

The Unified Modeling Language (UML) metamodel is a standardized representation of the concepts and relationships used in UML, a widely used modeling language for software and systems design. The UML metamodel defines the structure and behaviour of UML models and provides a way to automatically describe and manipulate them [22, 25].

Some of the characteristics of the UML metamodel are:

- Abstraction: The UML metamodel provides abstractions for UML model elements such as classes, interfaces, associations and state machines.
- Extensibility: The UML metamodel is designed to be extensible so that new elements can be added to it on demand.
- Consistency: The UML metamodel provides a consistent and well-defined structure for UML models, ensuring that they can be easily understood and used by different stakeholders.
- Semantics: The UML metamodel provides a clear and concise definition of the semantics of UML models, making it easier for them to be automatically interpreted and manipulated.
- Object orientation: The UML metamodel is well suited for modeling complex systems and software applications because it is based on object-oriented principles.
- Platform independence: The UML metamodel is platform independent, meaning that UML models can be created, manipulated and transformed on any platform that supports the UML metamodel.

Overall, the UML metamodel is a powerful tool for software and systems design, enabling users to create, manipulate and transform UML models in a standardized and automated way.

5.2 Example of UML metamodel refactoring

- Adding a concept: A refactoring rule's body might generate new UML metamodel items. It is mandatory to link a recently established UML metamodel concept to the rest of the UML metamodel and use a structure property for such activity.
- Duplicating a concept in certain cases, it's like to see a same notion in several portions of a UML metamodel.
- Finally, the last fundamental operation involving a UML metamodel is concept deletion.
- Refactoring, when used frequently, is a potent strategy. Only the designer's selected subset has to be refactored and modified.
- Manual refactoring of UML metamodel can cause errors and may result in inconsistencies. Moreover, it is very difficult to perform all parts of the refactoring potentially in a manual way.

Also, manual refactoring of UML metamodel according to such changes is very time consuming and is a source of errors. Such an issue becomes very relevant when dealing with the refactoring of complex UML metamodel with a considerable number of rules.

Short macro commands are utilized to streamline repetitive tasks during UML metamodel refactoring. Refactoring is applied based on UML metamodel changes; the default refactoring can be expanded or even modified by designers, who can define new refactoring rules to modify or replace the refactoring adaption.

This allows, for example, a designer to use two different refactoring. The assumption is that refactoring should only change concepts that the designer has explicitly chosen. By extension, refactoring are concrete transformations that preserve the behavior of an application. Such transformations only affect the appearance without adding functionality, but allow a better understanding of the system or facilitate later functional modifications.

The development cycle of large projects can be long for various reasons, such as the complexity of the project, the number of team members involved, and the need for thorough testing and quality assurance.

As for coordinating changes in the reconstruction of the UML underlying building meta-model, it requires a well-defined development process, clear communication among team members, and a robust version control system. The development team should establish clear guidelines for making changes and ensure that everyone is on the same page regarding the expected outcome. Additionally, regular meetings and progress updates can help to keep everyone informed and prevent any misunderstandings. The use of a version control system, such as Git, can also help keep track of changes made to the UML meta-model and ensure that everyone is working with the most up-to-date version.

6. THE REFACTORING PROPOSED IN THIS METHOD

The refactoring proposed in this method applies essentially to three concepts: concepts, methods, and variables. These refactoring can be classified into five basic types of operations:

- Add.
- Modify.
- Deletion.
- Generalization of UML metamodel elements.
- Specialization of UML metamodel elements.

The last two types move elements through the inheritance hierarchy, along with the generalization relationships. Most of the elements that make up the meta-model can have a direct connection to other elements of the same UML metamodel.

Adding and removing elements

It is possible to add member's (attributes or methods) to a concept if the new member or association does not have the same signature as any other member or association of the considered concept, of a super-concept or a sub-concept of it.

The deletion of associations and members is only possible if the deleted element is not referenced in the UML metamodel.

When the inheritance structure is taken into consideration, adding and removing concepts becomes especially interesting.

One can insert a generalization instance in the middle of a generalization relation, between two parent elements; the inserted element must not introduce any behavior, and especially be of the same type as the two other concepts.

Deleting a generalization instance has the opposite effect: One removes a useless element to link its sub-concepts directly to its super-concepts; the element must then not be referenced in other concepts, neither directly or indirectly - through instances, members etc.

Generalization refactoring may be applied to concepts' constituents such as attributes, relationships, methods, and operations.

Private members cannot be relocated in this manner since they are not available from the sub-concepts. This refactoring suggests that all of the super-immediate concept's sub-concepts have an identical element, for attributes, associations or operations this equivalence can be checked structurally, but the problem is more difficult for methods.

Specialization refactoring is the reverse of the previous one: it sends an element of a concept to all its sub-concepts. Informally, it preserves the behavior if the original concept is not the reference context of the element, i.e., if the element is only used via instances of sub-concepts of the original concept.

Other problems may arise if the existence of multiple inheritances is neglected. It is necessary to check that the concepts that will receive the transferred element do not have a common sub concept, i.e., that the traditionally problematic inheritance pattern in diamond does not occur.

6.1 List of refactoring rules

The proposed method offers a list of refactoring rules to the designer and he should be able to specify rules tailored to his needs.

The modifications made to the UML metamodel concern the concept itself.

-Addition and deletion of a concept, modification of the concept name.

-The definition of a concept (addition or deletion of an attribute, modification of the name of an attribute, addition or deletion of a parameter).

-Methods (e.g., modification of a method name or signature). As well as a dynamic management of their modification.

The proposed method has a tool that allows to select an element of the UML metamodel to be modified, to proceed to the modification and to propagate the modification through the UML metamodel by creating if necessary new versions of the concepts. The tool will also give the possibility to modify refactoring strategies and their corresponding rules and reuse them. Refactoring operations supported in this method; Table 1 shows a table with all supported operations.

Table 1. Some example of a UML metamodel refactoring operations

Refactoring	Rename a concept Move a property of a concept Extract concept Association to concept Concept to association Generalise/restriction of proprieties
Construction	Add a concept Add a property Add a relationship Edit hierarchy
Destruction	Delete a concept Delete a property Delete the legacy

6.2 Refactoring algorithm

The refactoring is performed in two phases:

1- Firstly, all modules and types (concepts, enumerations, and data types) are constructed in the step. The algorithm examines each package and type before adding them to the new UML metamodel.

2-The second step is concerned with the accurate design of concept internal structure. This involves adding characteristics, references, and operations. This second step also includes the assignment of super concepts.

A case of UML metamodel refactoring can be described by a problem section and a solution section:

1. The problem section contains.

a- a semantic specification of the refactoring.

b- The previous UML metamodel prior to the refactoring.

2. The solution section contains.

a- the description of the refactoring steps (UML metamodel components that have been added, updated, or deleted).

b- The refactored UML metamodel.

A refactoring rule can edit any component of the model; its intended scope is not restricted to the components supplied as real inputs. The cause for this is that the number of elements that may be altered in a rule is limitless. It is often impractical to define the components to be updated through collections or requests.

Most of the time, a refactoring is described by a group of rules rather than a single rule. Each rule can have a unique signature, and a rule's guard (condition) can apply to the guards of other rules in the same iteration. It is possible, for example, to declare a rule that can only be performed when some other rule is disabled.

A refactoring to change a public attribute to a private attribute is an example of a more sophisticated refactoring.

- Refactoring process

The refactoring process is separated into many actions:

1. Determine which components of the model must be refactored.

2. Choose the refactoring(s) to use at these situations.

3. Ensure that the refactoring, once done, maintains the system's behavior.

4. Execute the refactoring.

5. Consider the impact of refactoring on UML metamodel quality level.

6. Keep the refactored UML metamodel and its model consistent.

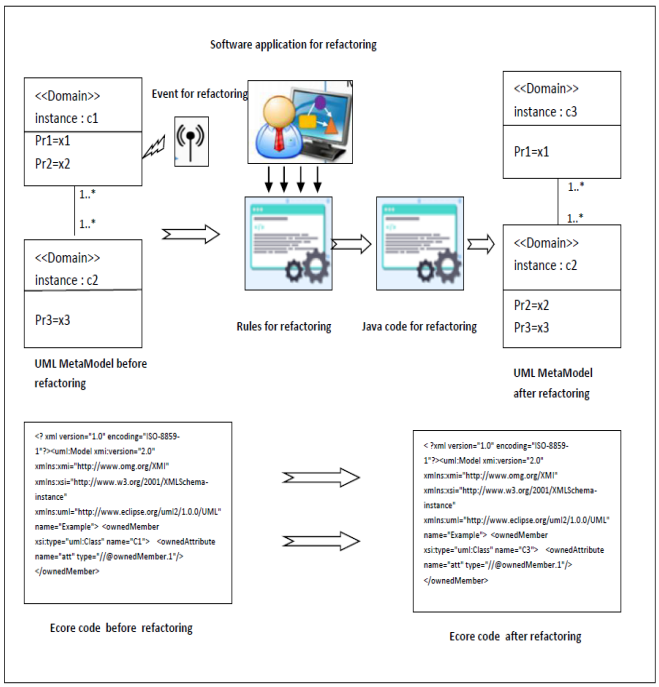

Figure 1. The proposed method for UML metamodel refactoring

The proposed method allows high-level refactoring of UML meta-models.

It provides a visual notation for defining meta-model refactoring chains it is supported by a GUI and a runtime engine that loads the appropriate models and executes the refactoring in the predefined path.

To ensure the refactoring the engine starts by reading the UML metamodel then it looks for the refactoring that are activated. Once these refactoring are identified, they are loaded and executed, and produce the corresponding output metamodel. The process continues until that no activated refactoring remains unexecuted. Figure 1 shows the process for refactoring of UML meta-models.

Then the next refactoring is activated and executed. The current version of the runtime executes the refactoring sequentially.

The proposed approach supposes that the designer who validates and verifies the refactoring.

After UML metamodel refactoring, designers can perform verification to ensure that the changes made to the UML metamodel have the intended effects on the code operation. The following are some ways to verify the UML metamodel validity after refactoring:

Simulation: The designer can simulate the behavior of the UML metamodel to ensure that it meets the requirements of the system and that the changes made during refactoring have not introduced any errors or unintended behavior.

Model Generation: The designer can generate model from the updated UML metamodel and compare it from the original UML metamodel to verify that the changes made during refactoring is valid.

Model Review: The designer can perform a model review to verify that the model generated from the updated UML metamodel meets the design and implementation standards of the organization.

Performance Testing: The designer can perform performance testing to verify that the changes made during refactoring have not impacted the efficiency of the model.

These are others verification methods that can be used after UML refactoring. The exact methods used will depend on the specific requirements and constraints of the project. These methods will be studied and explored in future work.

6.3 Refactoring the UML metamodel in complex system

There are several steps involved in refactoring the UML metamodel for a complex system:

1. Analysis: Before making any changes, it is important to thoroughly analyze the existing metamodel to understand its strengths and weaknesses and to identify areas for improvement. The various aspects and domains involved in the system, including hardware and software components, as well as the static structure and dynamic behaviour, should be considered in this analysis.

2. Planning: A plan for the refactoring process, including goals, scope and schedule, should be created based on the analysis.

3. Model decomposition: Decompose the existing UML metamodel into smaller, more manageable parts. This makes it easier to modify and understand. This may involve breaking the UML metamodel into separate models for hardware and software components, for example.

4. Model transformation: Use model transformation techniques to modify the UML metamodel, such as updating class and component diagrams, adding or removing classes and relationships, and changing model structure.

5. Validation: Validate the updated UML metamodel to make sure that it is an accurate reflection of the desired behaviour of the system, both statically and dynamically. This may involve testing the model with simulation and validation tools.

6. Documentation: Document the changes made to the UML metamodel and any associated updates to the development process to ensure that the updated UML metamodel is well understood by all team members.

It is important to continue to communicate and collaborate with team members throughout the refactoring process, and to make sure that everyone is aware of the goals and progress of the refactoring effort.

7. REFACTORING IMPLEMENTATION

The described method was implemented using EMFs, which may be thought of as an implementation of the Essential Meta Object Facility (EMOF) (Applied to the UML class diagram metamodel) [26, 27]. A UML metamodel is used to Express MM2MM refactoring, a script is used to conduct MM2MM refactoring based on distinct UML metamodel, and a library is used to provide automated data copying in endogenous refactoring.

A number of examples of UML metamodel refactoring operations are illustrated in a series of from Figure 2 to Figure 11.

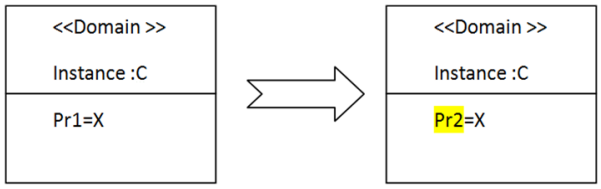

Figure 2. Rename property

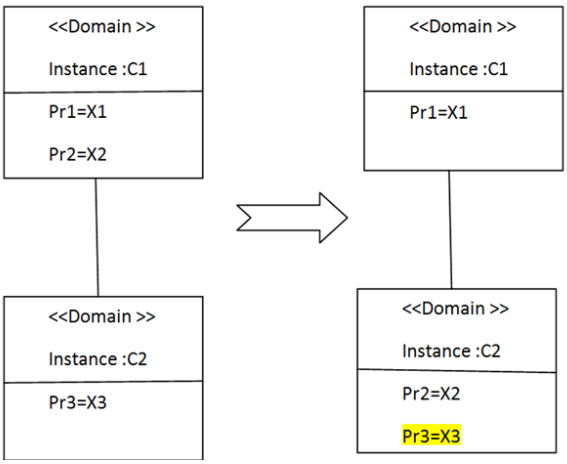

Figure 3. Move a new property

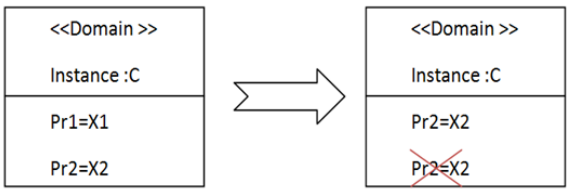

Figure 4. Delete a proporty

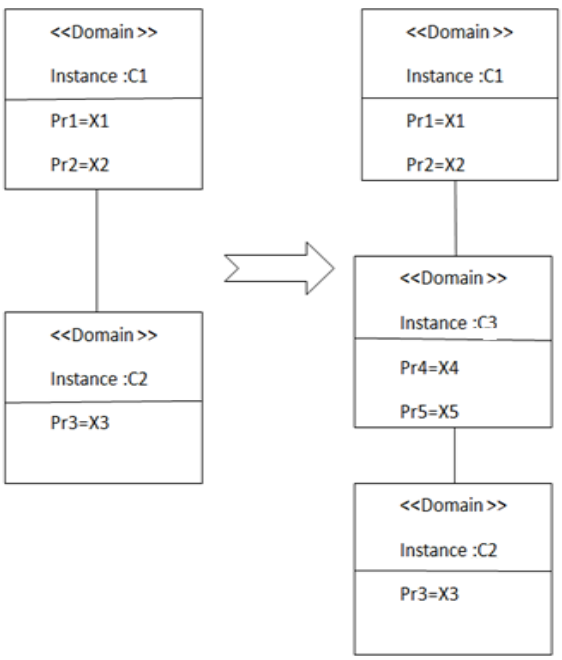

Figure 5. Add a new class

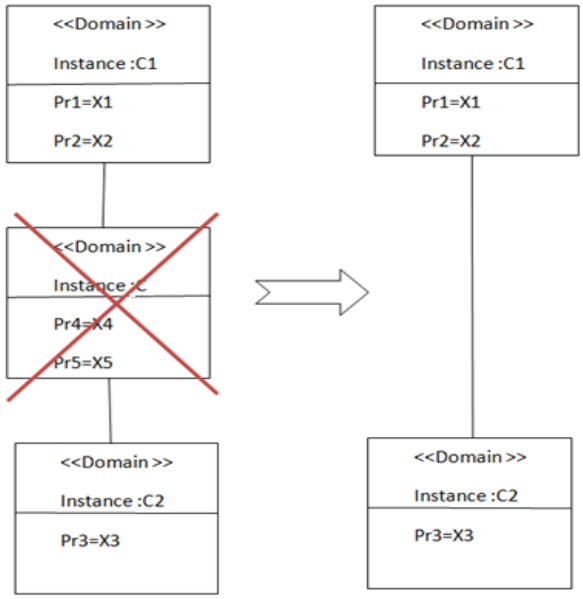

Figure 6. Delete a new concept and reference

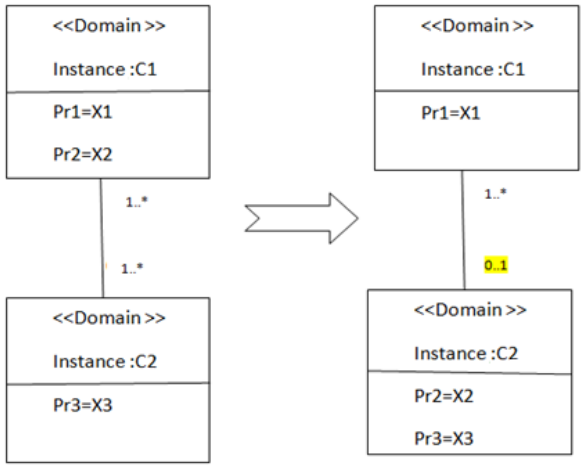

Figure 7. Modification of a concept and refrence

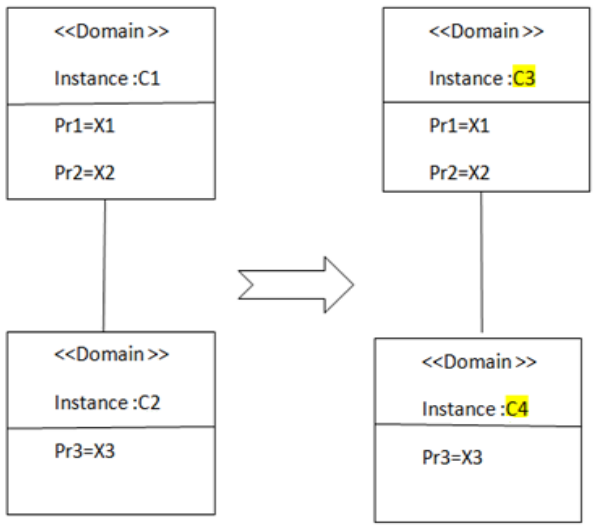

Figure 8. Change a concept name

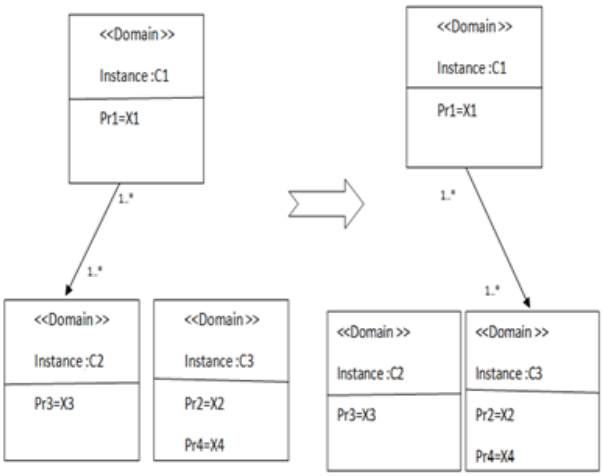

Figure 9. Modification of a reference

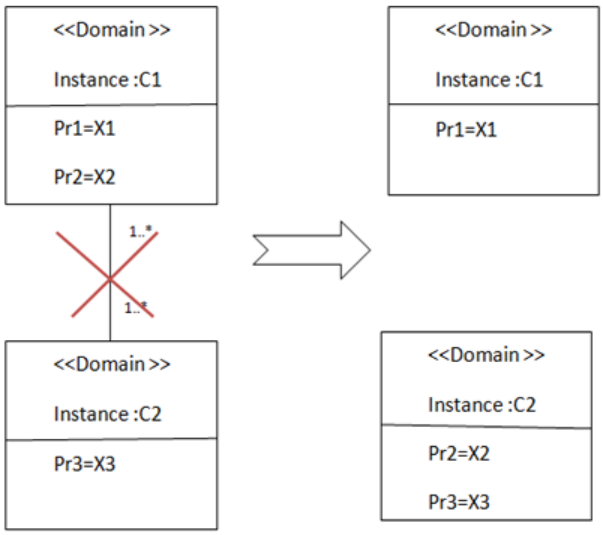

Figure 10. Delete a reference

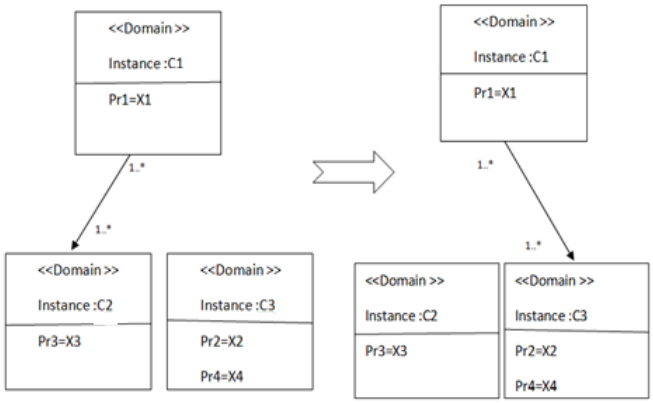

Figure 11. Modification of reference type

7.1 Implementation of refactoring rules

The ATL language will be used to implement several refactoring rules in the following sections.

7.1.1 Rule of Redundant Inheritance

According to this rule if a class inherits another through two or more separate inheritance pathways, delete all but one path if possible. A duplicate inheritance adds no information to the model but complicates it. The first inheritance is superfluous and should be eliminated (see Listing 1).

Listing 1 Rule of Redundant Inheritance

```
module RedundantInheritance;
create OUT : UML2target from IN : UML2;
 -- helper getSuperTypes
-- OUT : Sequence(UML2!Class)
helper context UML2!Class def :
getSuperTypes : Sequence(UML2!Class) =
 self.supertypes->iterate(a;
acc:Sequence(UML2!Class)=Sequence{}|
       if elf.getAllSuperTypes->count(a)>1
s
       then acc->union(Sequence{})
       else acc->append(a)
       endif
 );
 -- helper getAllSuperTypes
helper context UML2!Class def:
getAllSuperTypes : Sequence(UML2!Class) =
  if self.supertypes->isEmpty()
   then Sequence{}
  else   self.supertypes->select(c |
c.supertypes->notEmpty())
     ->iterate(a; acc :
Sequence(UML2!Class)=Sequence{} |(acc-
>including(a.getSuperTypes)))
     ->union(
     self.supertypes->iterate(a; acc :
Sequence(UML2!Class)=Sequence{} | acc-
>including(a))
 ).flatten()
  endif;
```

7.1.2 Rule of removing an association class (see Listing 2)

Listing 2 Rule of removing an association class

```
module RemovingAnAssociationClass;
create OUT : UML2 from IN : UML2;

rule Model {
  from
 inputM : UML2!Model
  to
  outputM : UML2!Model (
   name <- inputM.name,
   ownedMember <- inputM.ownedMember,
   ownedMember <- inputM.ownedMember ->
   select(a |
a.oclIsTypeOf(UML2!AssociationClass))->
   collect(c|Sequence
{thisModule.resolveTemp(c, 'outputAsso1')})-
>flatten(),
     ownedMember <- inputM.ownedMember ->
     select(a |
a.oclIsTypeOf(UML2!AssociationClass))->
     collect(c|Sequence
{thisModule.resolveTemp(c, 'outputAsso2')})-
>flatten()
  )
}
rule DataType {
  from
 inputC : UML2!DataType
  to
  outputC : UML2!DataType (
  name <- inputC.name
  )
}
rule LiteralNull {
  from
  inputLN : UML2!LiteralNull
  to
  outputLN : UML2!LiteralNull
}
rule teralInteger {   Li
  from
 inputLI : UML2!LiteralInteger
  to
  outputLI : UML2!LiteralInteger (
   value <- inputLI.value
  )
```

7.1.3 In the modification case

The objective of this refactoring is to transform a RootA element into a RootB, and transform an ElementA element into an ElementB. There are some additional constraints to be respected (see Listing 3):

- The order of the elements in the list must be preserved.
- An ElementB must be created from the name of a RootA. This element is added to the first position of the list.
- The name of each ElementB must start with 'B_'.

In summary, the created list will contain one more element than the original list. This extra element is created from of the name of the root of the list. It will be placed in the first position on the list.

Listing 3 code ATL in modification case

```
module A2B;
create OUT : B from IN : A;

-- Cette règle transforme un RootA en RootB.
-- Un ElementB est créé à partir du nom de RootA et il est placé en première position de la liste.
rule Root {
    from
        s : A!RootA
    to
        t : B!RootB(
            elms <- OrderedSet {first_element, s.elms}
        ),
        first_element : B!ElementB(
            name <- 'B_' + s.name
        )
}

-- Cette règle transforme un ElementA en un Element.
-- 'B' est préfixé au nom de chaque élément
rule Element {
    from
        s : A!ElementA
    to
        t : B!ElementB(
            name <- 'B_' + s.name
        )
}
```

7.1.4 In the case of addition

Listing 4 ATL code for addition case

```
module UML2Transformations; -- Module Template
create OUT : UML2target from IN : UML2;

helper context UML2!Class def : getProperties : Sequence(UML2!Properties) =
            UML2!Association.allInstances()->select(a|a.endType->includes(self))
                        ->iterate(a;acc : Sequence(UML2!Property) = Sequence {}|
                                                acc->including(a.ownedEnd-
>select(p|p.type <> self)
                                                                        ->first()))
      ;
rule model {
      from
            inputModel : UML2!Model
      to
            outputModel : UML2target!Model (
                  name <- inputModel.name,
                  visibility <- inputModel.visibility,
                  packageableElement_visibility <-
inputModel.packageableElement_visibility,
                  ownedMember <-inputModel.ownedMember
            )
}
rule class {
      from
            inputClass : UML2!Class
      to
            outputClass : UML2target!Class (
                  name <- inputClass.name,
                  visibility <- inputClass.visibility,
                  packageableElement_visibility <-
inputClass.packageableElement_visibility,
                  isAbstract <- inputClass.isAbstract,
                  isLeaf <- inputClass.isLeaf,
                  isActive <- inputClass.isActive,
                  ownedAttribute <- inputClass.getProperties-
>union(inputClass.ownedAttribute)
            )
}
rule property {
      from
            inputProperty : UML2!Property
      to
            outputProperty : UML2target!Property (
                  isDerived <- inputProperty.isDerived,
                  isDerivedUnion <- inputProperty.isDerivedUnion,
                  isLeaf <- inputProperty.isLeaf,
                  isOrdered <- inputProperty.isOrdered,
                  isReadOnly <- inputProperty.isReadOnly,
                  isStatic <- inputProperty.isStatic,
                  isUnique <- inputProperty.isUnique,
                  name <- inputProperty.name,
                  visibility <- inputProperty.visibility,
                  lowerValue <- inputProperty.lowerValue,
                  upperValue <- inputProperty.upperValue
            )
rule literalNull {
      from
            inputLiteral : UML2!LiteralNull
      to
            outputLiteral : UML2target!LiteralNull (
                  name <- inputLiteral.name,
                  value <- inputLiteral.value
```

For a UML metamodel element, new UML metamodel element is produced: Visibility and packageableElement_ visibility have the same name and are connected to the same OwnMember.

Another Class element is created for a Class element that has the same name, visibility, and packageableElement _visibility, as well as the same features, isAbstract, isLeaf, and isActive, and is linked to the same OwnAttribute. Additional Property element is produced for a property element.
With the same name, visibility and packageableElement_ visibility (see Listing 4).

7.1.5 In the case of deletion

The source and target of the refactoring have the same metamodel: UML2 (see Listing 5).

Rule model: For each model element, another model element containing the following elements is created:

The attribute name is also the same, and the same with the other UML metamodel elements.

Listing 5 ATL code in the case of deletion

```
module deletion;
create OUT : UML2 from IN : UML2;
rule Model {
from
inputM : UML2!Model
to
outputM : UML2!Model (
name <- inputM.name,
ownedMember <- inputM.ownedMember,
ownedMember <- inputM.ownedMember ->
select(a |
a.oclIsTypeOf(UML2!AssociationClass))->
collect(c|Sequence {thisModule.resolveTemp(c,
'outputAsso1')})->flatten(),
ownedMember <- inputM.ownedMember ->
select(a |
a.oclIsTypeOf(UML2!AssociationClass))->
collect(c|Sequence {thisModule.resolveTemp(c,
'outputAsso2')})->flatten())}
rule DataType {
from
inputC : UML2!DataType
to
outputC : UML2!DataType (
name <- inputC.name
)
}
rule LiteralNull {
from
inputLN : UML2!LiteralNull
to
outputLN : UML2!LiteralNull
}
rule LiteralInteger {
from
inputLI : UML2!LiteralInteger
to
outputLI : UML2!LiteralInteger (
value <- inputLI.value)}
rule LiteralUnlimitedNatural {
from
inputLUN : UML2!LiteralUnlimitedNatural
to
outputLUN : UML2!LiteralUnlimitedNatural (
value <- inputLUN.value
)
```

The UML metamodel contains also three important relations that can be implemented using ATL:

1. Equivalence: This relation is used to indicate that two or more concepts are equivalent (see Listing 6).
2. Implementation Inheritance: This relation is used to indicate that the implementation of a class is derived from another class (see Listing 7).
3. Interface Inheritance: This relation is used to indicate that the interface of a class is derived from another interface (see Listing 8).

Listing 6 ATL code for equivalence relation

```
module EquivalenceRelation;

create OUT : UML2Target from IN : UML2;

rule EquivalenceRelation
{
        from
                s : UML2!Class
        to
                t : UML2Target !Class (
                        name <- s.name
                )
}
```

This code defines a module named "Equivalence Relation". It creates an output UML metamodel of type "Target" from an input UML metamodel of type "Source". In this example, the name of each class in the "Target" UML metamodel is set to the name of the corresponding class in the "Source" UML metamodel .

Listing 7 ATL code for implementation of inheritance

```
module ImplementationInheritance;

create OUT : UML2Target from IN : UML2;


rule ParentClass
{
        from
                s : UML2!Class
        to
                t : UML2Target!Class (
                        name <- s.name,
                        isAbstract <- s.isAbstract
                )
}

rule ChildClass
{
        from
                s : UML2!Class (s.superClass <> null)
        to
                t : UML2Target!Class (
                        name <- s.name,
                        superClass <- s.superClass.name
                )
}
```

Listing 8 ATL code for interface inheritance

```
module InterfaceInheritance;

create OUT : UML2Target from IN : UML2;

rule ParentInterface
{
        from
                s : UML2!Interface
        to
                t : UML2Target!Interface (
                        name <- s.name
                )
}

rule ChildInterface
{
        from
                s : UML2!Interface (s.superInterfaces->notEmpty())
        to
                t : UML2Target !Interface (
                        name <- s.name,
                        superInterfaces <- s.superInterfaces.name
                )
}
```

This code defines a module named "ImplementationInheritance".

It creates an output model of type "Target" from an input UML metamodel of type "Source". The rule ParentClass specifies the transformation from elements of type Class in the "Source" UML metamodel to elements of type Class in the "Target" UML metamodel for parent classes. The rule ChildClass specifies the transformation for child classes that inherit from a parent class.

In this example, the name and abstract status of each class in the "Target" UML metamodel is set to the name and abstract status of the corresponding class in the "Source" UML metamodel. For child classes, the name of the super class in the "Target" UML metamodel is set to the name of the corresponding super class in the "Source" UML metamodel.

This code defines a module named "InterfaceInheritance". It creates an output model of type "Target" from an input UML metamodel of type "Source". The rule ParentInterface specifies the transformation from elements of type Interface in the "Source" UML metamodel to elements of type Interface in the "Target" UML metamodel for parent interfaces. The rule ChildInterface specifies the transformation for child interfaces that inherit from a parent interface.

In this example, the name of each interface in the "Target" UML metamodel is set to the name of the corresponding interface in the "Source" UML metamodel. For child interfaces, the names of the super interfaces in the "Target" UML metamodel is set to the names of the corresponding super interfaces in the "Source" UML metamodel.

The other proposed refactoring rules are not presented because their complexity is identical to the preceding instances, and it is beyond the scope of this work to detail all of them.

7.2 Application software for refactoring

In this research work an application software has been developed that performs the proposed refactoring, this application software allows loading the Ecore file of UML metamodel in zone 1.

After loading the Ecore file the designer must select the series of rules used during the refactoring (are displayed in area 3 (the rules are executed in the order chosen by the designer).

The description of the rule will be presented in zone 2.

The designer can add a new rule using the ***'add new Rule'*** button, he can also delete a selected rule using the ***'delete selected Rule'*** button or modify the rule using the ***'update selected Rule'*** button.

Once the Ecore file has been loaded and the series of rules chosen, the designer can launch the refactoring execution and see the result in zone 1 or cancel the refactoring using the ***'cancel refactoring'*** button, the Application software is presented in Figure 12.

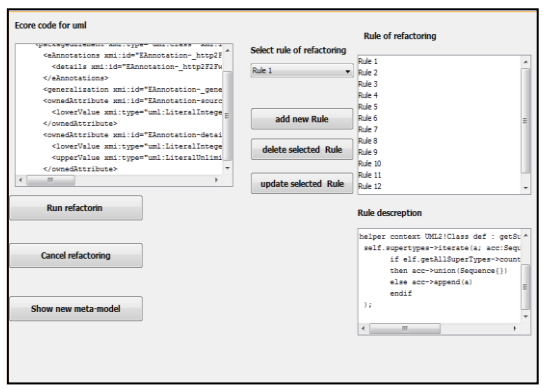

Figure 12. Application software for refactoring

8. CONCLUSIONS

Very often UML metamodel designer has to modify and refactor the existing UML metamodel.

The MDA proposes a refactoring of the UML metamodel in several steps; however, this requires the management of UML metamodel and the copying of data between the corresponding UML metamodel. If large parts of the UML metamodel remain unchanged, designers have to specify many copy operations to avoid this problem.

The primary aim of this paper is to shed light on the refactoring of the UML metamodel. Firstly, a thorough investigation of existing approaches to UML metamodel refactoring was conducted. The study resulted in the identification of different classifications of these approaches. Subsequently, critical evaluation criteria were selected and applied to the studied approaches. Upon completion of the comparative analysis, it was observed that no approach fulfilled all the selected criteria.

As a result of the analysis, guidelines were established. These guidelines aim to solve the refactoring problem with more expressiveness.

Clarity and support the change and scalability of the refactoring strategy to ensure its accuracy.

In addition, the use of standard tools such as EMF and ATL allows the solution to be widely distributed and facilitates its interoperability with other systems.

In this paper, the results of the initial experimentation utilizing the in-place refactoring approach for UML metamodel refactoring have been reported. The experiences indicate that refactoring of UML metamodel may be easily described with current in place refactoring languages.

As future work, it is planned to address the consistence problems caused by the UML metamodel refactoring and managing the refactoring via a graphic interface and improve the refactoring tools. Also, it is intended to measure the performance of UML metamodel refactoring rules.

REFERENCES

[1] Opdyke, W.F. (1992). Refactoring object-oriented frameworks. University of Illinois at Urbana-Champaign. https://hdl.handle.net/2142/72072.

[2] Sidhu, B.K., Singh, K., Sharma, N. (2018). Refactoring UML models of object-oriented software: A systematic review. International Journal of Software Engineering and Knowledge Engineering, 28(9): 1287-1319 https://doi.org/10.1142/S0218194018500365

[3] Bettini, L., Di Ruscio, D., Iovino, L., Pierantonio, A. (2022). Supporting safe metamodel evolution with edelta. International Journal on Software Tools for Technology Transfer, 24(2): 247-260. https://doi.org/10.1007/s10009-022-00646-2

[4] Langer, P., Wimmer M., Brosch, P., Markus Herrmannsdörfer, M., Seidl M., Wieland K., Kappel G. (2013). A posteriori operation detection in evolving software models. Journal of Systems and Software, 86(2): 551-566. https://doi.org/10.1016/j.jss.2012.09.037

[5] Brant, J., Roberts, D. (1999). Refactoring techniques and tools (Plenary talk). In Smalltalk Solutions, New York, NY.

[6] D'Souza, D.F., Wills, A.C. (1998). Objects, components, and frameworks with UML: The catalysis approach. Addison-Wesley Reading, vol. 1.

[7] Tokuda, L., Batory, D. (2001). Evolving object-oriented designs with refactorings. Automated Software Engineering, 8(1): 89-120. https://doi.org/10.1023/A:1008715808855

[8] Tip, F., Kiezun, A., Bäumer, D. (2003). Refactoring for generalization using type constraints. In Object-Oriented Programming Systems, Languages, and Applications (OOPSLA 2003), 38(11): 13-26. https://doi.org/10.1145/949343.949308

[9] Garces, K., Jouault, F., Cointe, P., Bézivin, J. (2009). Managing model adaptation by precise detection of metamodel changes. ECMDA-FA 2009. Lecture Notes in Computer Science Springer, 5562: 34-49. https://doi.org/10.1007/978-3-642-02674-4_4

[10] Tisi, M., Jouault, F., Fraternali, P., Ceri, S., B´ezivin, J. (2009). On the use of higher order model transformations. ECMDA-FA 2009. Lecture Notes in Computer Science Springer, 5562: 18-33. https://doi.org/10.1007/978-3-642-02674-4_3

[11] Cicchetti, A., Ruscio, D.D., Eramo, R., Pierantonio, A. (2008). Automating co-evolution in model-driven engineering. In 2008 12th International IEEE Enterprise Distributed Object Computing Conference, Munich, Germany, pp. 222-231. https://doi.org/10.1109/EDOC.2008.44

[12] Wachsmuth, G. (2007). Metamodel adaptation and model co-adaptation. In Proceedings of the 21rd European Conference on Object-Oriented Programming (ECOOP'07), 4609: 600-624. https://doi.org/10.1007/978-3-540-73589-2_28

[13] Narayanan, A., Levendovszky, T., Balasubramanian, D., Karsai, G. (2009). Automatic domain model migration to manage metamodel evolution. In Proceedings of the 12th International Conference on Model Driven Engineering Languages and Systems (MoDELS'09), 5795: 706-711. https://doi.org/10.1007/978-3-642-04425-0_57

[14] Herrmannsdoerfer, M. (2011). COPE – a workbench for the coupled evolution of metamodels and models. In: Malloy, B., Staab, S., van den Brand, M. (eds) Software Language Engineering. SLE 2010. Lecture Notes in Computer Science, vol. 6563. Springer, Berlin, Heidelberg. https://doi.org/10.1007/978-3-642-19440-5_18

[15] Wimmer, M., Kusel, A., Schönböck, J., Retschitzegger, W., Schwinger, W., Kappel, G. (2010). On using inplace transformations for model co-evolution. In Event2nd International Workshop on Model Transformation with ATL, 711: 65-78. https://www.researchgate.net/publication/283612710_On_using_inplace_transformations_for_model_co-evolution.

[16] Khelladi, D.E., Hebig, R., Bendraou, R., Robin, J., Gervais, M.P. (2015). Detecting complex changes during metamodel evolution. In International Conference on Advanced Information Systems Engineering, 9097: 263-278. https://doi.org/10.1007/978-3-319-19069-3_17

[17] Barišić, A., Debreceni, C., Varro, D., Amaral, V., Goulão, M. (2018). Evaluating the efficiency of using a search-based automated model merge technique. In 2018 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC), Lisbon, Portugal, pp. 193-197. https://doi.org/10.1109/VLHCC.2018.8506512

[18] Tsantalis, N., Ketkar, A., Dig, D. (2020). Refactoring miner 2.0. IEEE Transactions on Software Engineering, 48(3): 930-950. https://doi.org/10.1109/TSE.2020.3007722
[19] Mehmood, W., Shafiq, M., Saleem, M.Q., Alowayr, A.S., Aslam, W. (2020). A feature-based evaluation of model merge methods for e-health solutions. Journal of Medical Imaging and Health Informatics, 10(10): 2473-2480. https://doi.org/10.1166/jmihi.2020.3273
[20] Williams, J.R., Paige, R.F., Polack, F.A. (2012). Searching for model migration strategies. In Proceedings of the 6th International Workshop on Models and Evolution, ACM, pp. 39-44. https://doi.org/10.1145/2523599.2523607
[21] Di Ruscio, D., Iovino, L., Pierantonio, A. (2011). What is needed for managing co-evolution in MDE. In: Proceedings of the 2nd International Workshop on Model Comparison in Practice, ACM, pp. 30-38. https://doi.org/0.1145/2000410.2000416
[22] Levendovszky, T., Balasubramanian D., Narayanan A., Shi F. VanBuskirk, C., Karsai G., (2014). A semi-formal description of migrating domain-specific models with evolving domains, Software and Systems Modeling (SoSyM), 13(2): 807-823 https://doi.org/10.1007/s10270-012-0313-5
[23] Garcés, K., Jouault, F., Cointe, P., Bézivin, J. (2009). Managing model adaptation by precise detection of metamodel changes. In Model Driven Architecture-Foundations and Applications: 5th European Conference, ECMDA-FA 2009, Enschede, The Netherlands, June 23-26. https://doi.org/10.1007/978-3-642-02674-4_4
[24] Garcia, J., Diaz, O., Azanza, M. (2012) Model transformation co-evolution: A semi-automatic approach. In: Czarnecki, K., Hedin, G. (Eds.), Software Language Engineering, pp. 144-163. https://doi.org/10.1007/978-3-642-36089-3_9
[25] Mumtaz, H., Alshayeb, M., Mahmood, S., Niazi, M. (2019). A survey on UML model smells detection techniques for software refactoring. Journal of Software: Evolution and Process, 31(3): e2154. https://dx.doi.org/10.1002/smr.2154
[26] The Unified Modeling Language; Kirill Fakhroutdinov; http://www.uml-diagrams.org/, accessed on Jun. 22, 2022.
[27] OMG Unified Modeling Language; http://www.omg.org/spec/UML/, accessed on Jun. 23, 2022.