A semi-automatic maintenance and co-evolution of OCL constraints with (meta)model evolution

Djamel Eddine Khelladi, Reda Bendraou, Regina Hebig, Marie-Pierre Gervais

A Semi-Automatic Maintenance and Co-evolution of OCL Constraints with (Meta)model Evolution

Djamel Eddine Khelladi[1] *, Reda Bendraou[2,3], Regina Hebig[4], Marie-Pierre Gervais[2,3]

[1]*Institute for Software Systems Engineering, Johannes Kepler University Linz, Austria.*
[2]*Université Paris Ouest Nanterre La Defense, F-92001, Nanterre, France.*
[3]*Sorbonne Universités, UPMC Univ Paris 06, UMR 7606, F-75005, Paris, France.*
[4]*Chalmers and University of Technology Gothenburg, Sweden.*
djamel_eddine.khelladi@jku.at, {reda.bendrao, marie-pierre.gervais}@lip6.fr, hebig@chalmers.se

Abstract

Metamodels are core components of modeling languages to define structural aspects of a business domain. As a complement, OCL constraints are used to specify detailed aspects of the business domain, e.g. more than 750 constraints come with the UML metamodel. As the metamodel evolves, its OCL constraints may need to be co-evolved too. Our systematic analysis shows that semantically different resolutions can be applied depending not only on the metamodel changes, but also on the user intent and on the structure of the impacted constraints. In this paper, we first investigate the syntactical reasons that lead to apply different resolutions. We then propose a co-evolution approach that offers alternative resolutions while allowing the user to choose the best applicable one. We evaluated our approach on six case studies of metamodel evolution and their OCL constraints co-evolution. The results show the usefulness of alternative resolutions along with user decision to cope with real co-evolution scenarios. Within our six case studies our approach led to an average of 92% (syntactically) and 93% (semantically) matching co-evolution w.r.t. the user intent.

Keywords:
Metamodel, Evolution, OCL, Constraints, Co-evolution.

*Corresponding author. Email: djamel_eddine.khelladi@jku.at

1. Introduction

In the past years, *Model-Driven Engineering (MDE)* has proven to be effective in the development and maintenance of large scale and embedded systems [18, 28]. As a consequence, modeling languages have emerged in the industry [47] supporting all typical phases of development processes [45].

At the heart of any modeling language ecosystem, there is a *metamodel* [4]. Metamodels are core components to the well-functioning of a modeling language ecosystem [18]. They define the structural aspects of a business domain, i.e. the main concepts, their properties, and relationships between them [19]. However, a metamodel alone is insufficient to capture all the relevant aspects and information of a domain specification [32]. To overcome this limitation, the Object Constraint Language (OCL) [36] is used to define constraints on top of the metamodel. For instance, the wide-spread Unified Modeling Language (UML) [38] in version 2.4.1 contains more than 750 OCL constraints expressing well-formedness rules to be enforced at the model instance level.

Evolution of modeling languages is inevitable and necessary. One of the main evolution dimensions in a modeling language is the metamodel evolution. A challenge hereby arises when the metamodel is evolved causing the invalidation of some OCL constraints that may need to be co-evolved as well (i.e. maintained to remain reusable [33]). For instance, the UML metamodel officially evolved 10 times in the past [1], and these evolutions led to manually adapting the impacted OCL constraints among the ones associated to the UML metamodel. This is not only the case of UML, but also for many modeling languages such as BPMN[2] and GMF[3] who respectively evolved 4 times and 24 times.

Manual co-evolution can be tedious, time-consuming, and an error-prone task [13, 2], in particular when hundreds of OCL constraints are impacted by the changes. In such context, it is crucial to support software engineers with an automatic co-evolution for two reasons. The first one relates to the need to maintain the legacy, i.e. OCL constraints. Indeed, already defined OCL constraints fulfill part of the domain's specification that must naturally be maintained w.r.t. the new evolved version of the metamodel. The second one is the need to avoid introducing errors in source code and test cases that

[1] http://www.omg.org/spec/UML/
[2] http://www.omg.org/spec/BPMN/
[3] http://www.eclipse.org/modeling/gmp/downloads/?project=gmf-tooling

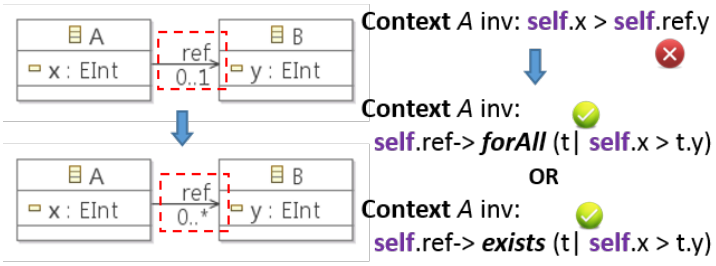


Figure 1: Existence of multiple solutions : An Example.

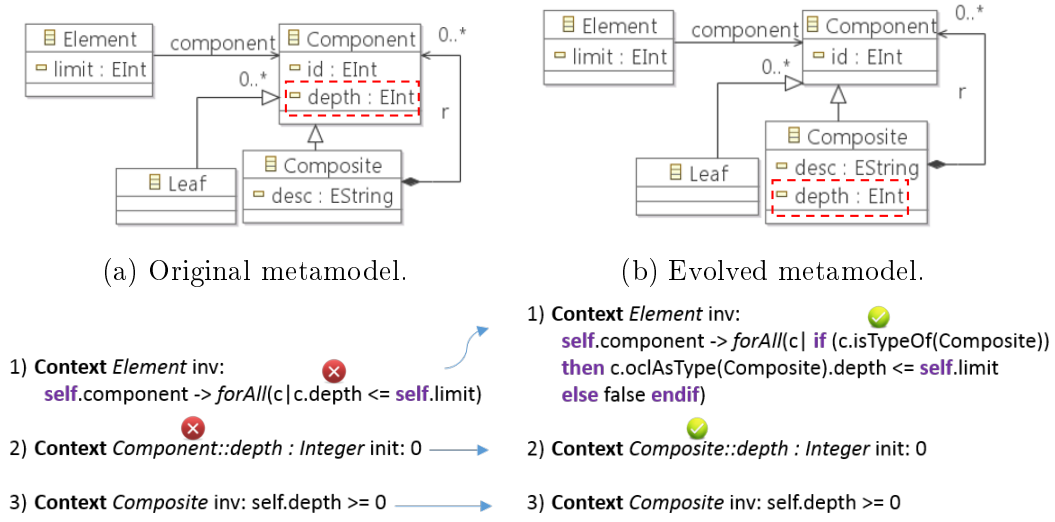
can be automatically generated from models and their OCL constraints (e.g. [14, 1]). This automatic generation would not be trustworthy if it was based on non co-evolved OCL constraints.

Problems. Automatically co-evolving OCL constraints remains challenging, mainly because of two issues: 1) *the existence of multiple and semantically different resolutions*, and 2) *a resolution can be applicable only to a subset of OCL constraints*. In the following, we detail these issues.

1) The impact of a metamodel change on an OCL constraint can be resolvable using resolutions that are syntactically and/or semantically different. For instance, the metamodel change *"multiplicity generalization of a property* `p` *from a single value to multiple values"* requires the OCL constraints to work on a collection of values, e.g. by introducing an iterator. Figure 1 gives an example of this change for the property *ref* with a simple OCL constraint.

Multiple resolutions can be applied here as depicted in Figure 1, since multiple iterators with *different semantics* exist, e.g. forAll(), exists() etc. Proposing a unique resolution reduces the applicability of the co-evolution approach and limits its benefit. Final decision can only be specified by the user herself, to avoid unintended co-evolution changes.

2) A given resolution strategy is not always applicable for all OCL constraints. The complex nature of OCL requires different resolution strategies, each one applicable for only a subset of OCL constraints based on: *a)* the *location* of the impacted part in an OCL constraint, and *b)* its *context* (i.e. the metamodel element on which an OCL constraint is defined). Figure 2 illustrates this issue. It depicts an evolution of a metamodel where the property *depth* is deleted from the superclass `Component` and added to the subclass `Composite`, which fits the definition of a "push property" [17]. The first two constraints become invalid because *depth* is no more accessible in `Component`. The first constraint that uses the pushed property *depth* through the refer-



(a) Original metamodel. (b) Evolved metamodel.

(c) Original OCL. (d) Evolved OCL.

Figure 2: An evolution of a composite pattern and a co-evolution of OCL constraints: An example.

ence *component*, can be co-evolved by introducing an `If` expression that first checks whether *component* references an instance of the subclass `Composite` so that *depth* is accessible. In contrast, the second OCL constraint whose context is defined on the pushed property *depth*, is co-evolved differently by duplicating it for the subclass `Composite`. The original constraint is then removed as depicted in Figure 2d. Note that the third constraint defined on the context of the subclass `Composite` that uses *depth* is not impacted. Clearly, a unique resolution strategy cannot be applied whatever the OCL constraint, for the three constraints in our example.

Consequently, it is crucial to consider the two above issues when co-evolving OCL constraints. However, existing approaches [13, 9, 10, 30, 31, 25, 6] neither consider the two above issues, nor interact with the user. We think that a co-evolution approach should find a balance between manual intervention and full automation. On the one hand, final decision regarding which resolution strategy to apply among a set of possible resolutions should be left to human. On the other hand, its application must be automated to reduce the effort of the co-evolution from the users.

Contributions. We addressed these challenges by four contributions:

- First, we systematically investigate what are the *influencing factors*

that lead to define alternative resolution strategies and when to apply them. Thus, we establish that the metamodel changes alone are insufficient to propose the appropriate resolutions, and additional factors must be considered.

- Second, we propose an approach that considers alternative resolutions per impacted part of an OCL constraint. Thus, for a metamodel change, we propose various resolutions for different subsets of OCL constraints. It allows us to cover different alternatives of co-evolution. A minimal set of resolutions is proposed so far in our approach.

- Third, based on the influencing factors and from our set of resolutions, we define a process that aims at proposing only the appropriate resolutions that can be applied to an impacted OCL constraint. The process takes the influencing factors for each impacted OCL constraint and excludes resolutions that cannot be applied on it. The user can then confirm which resolutions to be applied among the proposed alternative ones. Involving the user greatly contributes to avoid applying unintended resolution strategies.

- Fourth, we investigate the possibility to co-evolve non-impacted OCL constraints. Indeed, non-impacted OCL constraints may as well be co-evolved and/or refactored due in response to the metamodel evolution. Herein we propose to extend our co-evolution approach to also co-evolve non-impacted OCL constraints.

We validated the co-evolution approach on six case studies. The results show that we are able to co-evolve the impacted OCL constraints w.r.t. the user intent, which in 92% of the cases are syntactically (and in 93% of the cases are semantically) equivalent to the actual co-evolved constraints. Our approach is implemented as plugin for the Eclipse IDE a wide-spread software development environment.

In *Model-Driven Engineering* OCL constraints can be defined on object-oriented models, e.g., metamodels, UML models etc. While we focus in this paper on the co-evolution of OCL constraint defined on metamodels, the current approach theoretically applies on OCL constraints defined on object-oriented models in general.

This paper extends our previous work [24]. Additional contributions include a more detailed description of all resolutions that we propose, considering the co-evolution of non-impacted constraints, five additional case studies

used to show the feasibility of the approach with various evolution and co-evolution scenarios, a prototype that supports OCL co-evolution defined on both the Ecore modeling tool and the UML Class Diagram (CD) Papyrus modeling tool, and a more comprehensive and extensive discussion of our results and the related work.

For a better understanding of the current approach, this paper first discusses in Section 2 the factors we identified that influence the application of the resolution strategies. Section 3 then presents the overall approach introduces some of the proposed resolutions. Section 4 illustrates our implementation. The evaluation, results, discussion, and threats to validity are presented in Section 5. Finally, Section 6 and 7 present respectively the related work and the conclusion.

2. Factors Influencing the Resolution Strategies

In this section, we identify the factors that influence the choice of the resolution strategies. To illustrate the influencing factors we reuse the example used in the introduction as depicted in Figure 2.

2.1. Factor 1

First of all, the type of a metamodel change is fundamental to choose which resolution to apply, similarly as in model co-evolution approaches (e.g. [12, 49, 15, 8]). The impacts of a rename property and a push property cannot be fixed using the same resolution. Thus, the first influencing factor is: **the metamodel change**.

2.2. Factor 2

We further investigated which locations in an OCL constraint can influence the choice of a resolution. We identified two locations that have an influence: *navigation path* and *context*. For example, in Figure 2 the two first OCL constraints need to be resolved differently since the pushed property *depth* is used in different locations. In the first constraint *depth* is located in the OCL expression (i.e. body) through a *navigation path*. Whereas, in the second constraint *depth* is located in the *context*. Thus, the second influencing factor is: **the location of the impacted metamodel element e in an OCL constraint**.

2.3. Factor 3

Finally, we found a third factor that is the *context* of the constraint, which can influence the choice of a resolution. In Figure 2, the first constraint is co-evolved by introducing an `If` expression and not by duplicating the constraint to the subclasses where *depth* is pushed, as we did with the second constraint. This is due to the fact that the context of the first constraint is not the superclass `Component`. When changing the context, all accessible properties from the old context must remain accessible from the new context. If the context of the first constraint is changed from `Element` to `Composite`, the property *limit* will not be accessed anymore. For the third constraint which has the sub class `Composite` as context no resolution strategy is applied, since *depth* is still accessible. Therefore, the third influencing factor is: **the *context* of the impacted constraint**.

Our analysis of the state-of-the-art led us to identify only the *factor 1* that is considered by all co-evolution approaches. To identify additional factors, we systematically studied the different uses of the metamodel elements in the OCL language (last version 2.4). Thus, we investigated: 1) the uses of each metamodel element in all possible structures that an OCL constraint can have (in the OCL grammar [36]) and further 2) the reasons why a resolution can be applied to some constraints and not to others. It allowed us to identify the additional *factors 2* and *3*. Our systematic analysis gives us confidence that the above factors are the only ones w.r.t. the current version of OCL. However, we cannot prove it formally that there is no other factors. So far, we have not found a counter-example. Note that the above factors consider syntactic aspects of the OCL constraints and less the semantic of the OCL constraints.

In summary, to propose appropriate resolutions we must consider the three identified influencing factors that are highlighted in the framebox here under.

1. The metamodel change of an element e.
2. Location of the impacted element e in an OCL constraint:
   (a) e is used in the context of an OCL constraint
   (b) e is used in the OCL expression (i.e. body) through a navigation path.

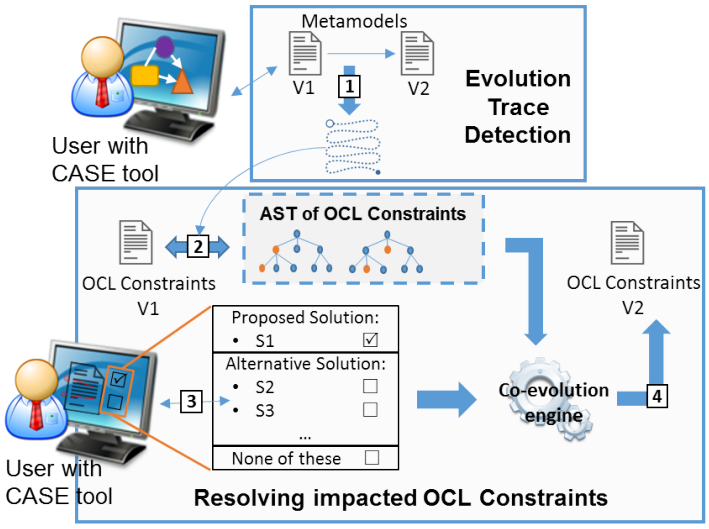


Figure 3: Overall Approach.

3. The context of the impacted constraint, to know from which class the impacted element is accessed.

3. A Co-Evolution Approach of OCL Constraints

This section presents our approach to co-evolve OCL constraints. Figure 3 depicts an overview of our approach. We first present the metamodel changes that we consider during an evolution and we present how they are identified [1]. After that, we discuss the identification of impacted OCL constraints, in particular the localization of the impacted parts in the constraints [2]. Then, we explain how we obtain the three influencing factors for each impacted OCL constraint. Finally, we show how alternative resolutions are proposed to the user [3] and how they are automatically applied [4].

Based on the discussion of challenges in the introduction of this work, we formulate three requirements that must be fulfilled in our approach:

- *R1. All OCL constraints that must be co-evolved are identified.*
- *R2. Alternative resolutions are always proposed whenever it is possible.*
- *R3. Only the appropriate resolutions that can be applied to the impacted OCL constraint are proposed.*

3.1. Metamodel Changes During Evolution

During a metamodel evolution two types of changes are distinguished: a) *Atomic changes* that are additions, removals, and updates of a metamodel element, and b) *Complex changes* that consist in a sequence of atomic changes combined together. For example, *move property* is a complex change where a property is moved from one class to another via a reference. This is composed of two atomic changes: delete property and add property [17].

We consider the following set of atomic changes: *add, delete, and update* of metamodel elements. An *update*, changes the value of a property of an element, such as 'type', 'name', 'upper/lower bounds'. The metametamodel elements that are considered in this work are: *package, class, attribute, reference, operation, parameter, and generalization*. Those elements represent the core features of a metamodel in the EMF/Ecore [46] and the MOF [35] standards. In the literature, over sixty complex changes are proposed [17]. Among them, we focus on seven complex changes: *move property, pull property, push property, extract super class, flatten hierarchy, extract class, and inline class*. A study of the evolution of GMF[4] metamodel showed that these seven changes are the most used ones and constitute 72% of the applied complex changes [16, 27]. In our case studies they constitute 100% of the applied complex changes in the evolutions.

In our co-evolution approach we must first identify metamodel changes that led from version *n* to version *n+1*, as shown by the step 1 in Figure 3. This is a prerequisite for both impact analysis and automatic support of the co-evolution. We reuse our detection tool [23, 22, 21], an extension of the Praxis tool [3]. It first records at run-time all atomic changes applied by users within a modeling tool. Note that the atomic changes can also be computed by the difference between the original and evolved metamodels. However, this technique has two drawbacks [23]: 1) The chronological order of the applied atomic changes is lost. 2) Some changes are *hidden* by other changes during evolution. For instance, when a move property *p* is followed by a rename *p* to *p'* during the evolution, computing the difference do not detect these last two changes, but sees only two independent changes: delete *p* and add *p'*. The sequence of recorded atomic changes then serves as input for the detection of complex changes. Our tool [23] has been designed to detect all applied changes. This is confirmed in the evaluation results by reaching a 100% recall (i.e. all complex changes are detected) and a precision (i.e.

[4]Graphical Modeling Framework http://www.eclipse.org/modeling/gmf.

Table 1: Impact Identification on OCL constraints.

Metamodel Elements	OCL Constraints	References to AST Nodes
e_1	$OCL_1, ...$	$ref_1, ref_2, ...$
e_i	$OCL_j, ...$	$ref_k, ref_l, ...$
...	...	...

correct detection) of 91% and 100%. Our detection tool [23] allows the user to confirm the list of complex changes that best reflect her intention during the evolution. Therefore, a final precise, complete, and ordered trace of both atomic and complex changes is computed. This trace is taken as input by our herein tool to co-evolve the OCL constraints when the user requires it.

3.2. Identification of Impacted OCL Constraints

The second step of our approach is to identify the OCL constraints impacted by metamodel changes during the evolution. In particular, we identify all the impacted parts of the OCL constraints that need to be co-evolved.

To run the impact analysis we need to access to all the elements used in an OCL constraint. Thus, we first parse the OCL constraints to use the Abstract Syntax Tree (AST) representing a structured view of an OCL constraint. The identification of impacted OCL constraint is then performed on the generated AST. Before identifying where an AST is impacted, we first compute a table that lists for each metamodel element e, all OCL constraints using e with references to the AST nodes using e. Those references will be further used in the resolution step. Table 1 illustrates an example of our computed table. To build Table 1, we apply a pre-order tree traversal algorithm on the AST while filling the table.

For each metamodel change on a metamodel element e_i, we access the set of impacted OCL constraints and we access exactly the impacted AST nodes with the saved references. Note that a metamodel complex change can involve several elements $e_i...e_j$. Thus, the set of impacted OCL constraints are accessed naturally for each element e_k where $i \leq k \leq j$. During the co-evolution process Table 1 is also updated accordingly with the applied resolutions. For instance, when a rename element e occurs, it is also renamed in the table. Using the Table 1 answers the requirement R1 to detect all impacted constraints.

By using Table 1, we might also have false positives by identifying non-impacted OCL constraints. In particular, when a *push property* or *flatten*

hierarchy are applied, already defined constraints on the subclasses are identified as impacted. However, they are filtered with the third influencing factor. No resolutions will be proposed since they are in fact non-impacted. Alternatively, one can use an OCL interpreter/evaluator to retrieve the OCL constraints that are invalid on which we run the impact analysis and we propose the appropriate resolutions for each impacted OCL constraint.

3.3. Obtaining the Influencing Factors

As discussed in section 3.1 the metamodel changes are given as input from our detection tool. The two last factors, i.e. location and context, are obtained from the impacted AST and AST nodes from Table 1. Each AST node has a type and information that we can use to determine the impacted location in the constraints as well as the context of a constraint.

For the context, we further identify whether the impacted element is accessed from the level of its container, the sub classes, or the super class. At this point, once the three influencing factors for an impacted part of an OCL constraint are determined, we can propose a set of possible resolutions, as we will describe it in Section 3.6.

3.4. Resolution Strategies

This section introduces our resolution strategies and the influencing factors under which they are applied on impacted OCL constraints. Table 2 shows the changes we consider in this paper as well as their classification, i.e. whether they do impact OCL constraints. The changes are evaluated as impacting, if there exists at least one situation where the metamodel changes impacts the syntactic correctness of the OCL constraints. All metamodel changes that are non-impacting do not need to be handled (n/a), i.e. no resolutions are defined for them. Among all impacting metamodel changes we handle those whose impact can be automatically co-evolved. Two changes are not handled since they require user intervention. For instance, when a parameter p is added to an operation *op()*. In all calls of the operation *op()*, a variable v of the same type of the parameter p must be added (in the right position when several parameters are defined) resulting in the new operation calls *op(v)*. This decision can only be manual, in particular when several parameters are defined with the same type. Although, the change modify property type in some cases does not impact OCL constraints, e.g., changing it from Integer to Double. However, it may also impact the OCL constraints in case the old and new types are incompatible, e.g., from *Date* type to *Enumeration* type, and hence requiring manual intervention.

Table 2: Metamodel change impact on OCL constraints.

	Metamodel changes	Impact on OCL constraints	Handled in our approach
Atomic changes	Add optional/mandatory class	no	n/a
	Add optional/mandatory property	no	n/a
	Add enumeration/enumeration literal	no	n/a
	Generalize property multiplicity from a set to a larger set	no	n/a
	Restrict property multiplicity	no	n/a
	Add parameter	yes	no
	Rename class	yes	yes
	Rename property	yes	yes
	Rename enumeration/enumeration literal	yes	yes
	Rename parameter	yes	yes
	Delete class	yes	yes
	Generalize property multiplicity from zero or one occurrence to a set	yes	yes
	Delete property	yes	yes
	Delete enumeration/enumeration literal	yes	yes
	Delete parameter	yes	yes
	Modify property type	yes	no
Complex changes	Move property	yes	yes
	Pull property	no	yes
	Push property	yes	yes
	Extract class	yes	yes
	Inline class	yes	yes
	Extract super class	no	yes
	Flatten hierarchy	yes	yes

As mentioned previously, in the following strategies, there is always the empty resolution strategy that does nothing. For sake of readability, we do not duplicate it for all changes in the rest of this section.

3.4.1. Rename Element

This strategy applies for a class, a property (i.e. attribute, reference, operation), a parameter, an enumeration, and an enumeration literal.

```
Id : #S1
Context : n/a
Location in the constraint : n/a
```

Description : When an element e in the metamodel is renamed to e', a rename is also applied in the OCL constraints using e.

3.4.2. Delete Element

Similarly, this strategy applies for a class, a property (i.e. attribute, reference, operation), a parameter, an enumeration, and an enumeration literal.

Id : #S2
Context : n/a
Location in the constraint : n/a
Description : When an element e in the metamodel is deleted, a delete of the constraints using e is also applied.

Note that for both rename and delete element, we do not need the information of the location of the element on the impacted OCL constraint and the context. These strategies applies to any constraint whatever the two last factors.

3.4.3. Generalize Property Multiplicity (GPM) from a Single Value to Multiple Values

An example of this metamodel change is already shown in Figure 1. This metamodel change requires the OCL constraint to work on a collection of values of a property p and not a single value anymore. Multiple solutions can be proposed all with a slightly different semantic and are presented below. Note that the operation *any(exp)* is not used since is requires a boolean condition as a parameter that only the user can specify..

Id : #S3.
Context : n/a.
Location in the constraint : navigation path.
Description : An iterator *"forAll"* is added to access the property p, and the subexpression (i.e., *restExp*) using the values of p is moved to the body of the *"forAll"* while replacing the access path with a temporary variable. The given semantic here is that the OCL constraint is satisfied if it is satisfied for all the values of p.

$$Exp.p.restExp \Rightarrow Exp.\mathbf{p}\text{->}\mathbf{forAll}(x|x.restExp)$$

Id : #S4.
Context : n/a.
Location in the constraint : navigation path.
Description : An iterator *"exists"* is added to access the property p, and the subexpression (i.e., *restExp*) using the values of p is moved to the body of the *"exists"* while replacing the access path with a temporary variable. The given semantic is that the OCL constraint is satisfied if at least it is satisfied for one value of p.

$$Exp.p.restExp \Rightarrow Exp.\mathbf{p}\text{->}\mathbf{exists}(x|x.restExp)$$

Id : #S5
Context : n/a
Location in the constraint : navigation path
Description : The first value of the collection of the property p is accessed.

$$Exp.p.restExp \Rightarrow Exp.\mathbf{p}\text{->}\mathbf{asOrderedSet()}\text{->}\mathbf{first()}.restExp$$

Id : #S6
Context : n/a
Location in the constraint : navigation path
Description : The last value of the collection of the property p is accessed.

$$Exp.p.restExp \Rightarrow Exp.\mathbf{p}\text{->}\mathbf{asOrderedSet()}\text{->}\mathbf{last()}.restExp$$

Id : #S7
Context : n/a
Location in the constraint : navigation path
Description : The value at the i-th position (given by the user) of the collection of the property p is accessed.

$$Exp.p.restExp \Rightarrow Exp.\mathbf{p}\text{->}\mathbf{asOrderedSet()}\text{->}\mathbf{at(i)}.restExp$$

3.4.4. Move Property

This change consists of moving a property p from a class A to a class B through a reference r from A.

When the moved property is used in the context, we propose the following resolution.

Id : #S8
Context : A (container)
Location in the constraint : context
Description : The context of the constraint is updated, by changing the source class to the target class where the property p is moved. Then, for each keyword *self* (equivalent to *this* in Java) referencing the old context, a navigation back to it is added through an existing reference *r'* given by the user as input.

*context A::p init: self.a*2 =>*
context ***B****::p init: self.****r'****.a*2*

When the moved property p is used in a navigation path, we distinguish two cases of move property : 1) through a single-valued reference and 2) through a multi-valued reference.

A) Through a single-valued reference

Id : #S9
Context : n/a
Location in the constraint : navigation path
Description : The navigation path to the property p is extended with the reference r.

*Exp.p.restExp => Exp.****r****.p.restExp*

B) Through a multi-valued reference

Id : #S10
Context : n/a
Location in the constraint : navigation path
Description : The navigation path to the property p is first extended with the reference r. Then, one of the strategies among #S3, #S4, #S5, #S6, and #S7 is chosen by the user to be applied, since r is returning a collection. This resolution strategy is a combination of #S9 and one of #S3, #S4, #S5, #S6, #S7.

*Exp.p.restExp => Exp.****r****->****forAll****(x|x.p.restExp)*
or
*Exp.p.restExp => Exp.****r****->****exists****(x|x.p.restExp)*
or
*Exp.p.restExp => Exp.****r****->****first().p****.restExp*
or

$$Exp.p.restExp \Rightarrow Exp.\boldsymbol{r}\text{->}\boldsymbol{last()}.\boldsymbol{p}.restExp$$
$$or$$
$$Exp.p.restExp \Rightarrow Exp.\boldsymbol{r}\text{->}\boldsymbol{at(i)}.\boldsymbol{p}.restExp$$

3.4.5. Push Property

This change consists in a delete of a property p from a superclass A and its addition in subclasses B_1,... B_n.

Id : #S11.
Context : A (container)
Location in the constraint : context

Description : The constraint is duplicated to the subclasses where the property p is pushed, while updating the context with the subclasses. The original constraint is deleted.

Id : #S12
Context : A (container)
Location in the constraint : navigation path
Description : If the context is on the superclass A, then duplicate the constraint to the subclasses where the property p is pushed, while updating the context with the subclasses. The original constraint is then deleted.

Id : #S13
Context : not via the subclasses
Location in the constraint : navigation path
Description : An If expression is introduced for each subclass where the property p is pushed. The condition tests whether the navigation path until p is a type of the subclass. The subexpression is copied in the then branch while casting to the subclass before accessing p. Note that all the If expressions are joined with the operator "or". This strategy is applied in Figure 5 in our example of Figure 2.

$$(Exp.p.restExp) \Rightarrow$$
$$(if(Exp.oclIsKindOf(subclass_1)) then$$
$$Exp.oclAsType(subclass_1).p.restExp \ else \ false$$
$$or \ ...$$

if(Exp.oclIsKindOf(subclass$_n$)) then
Exp.oclAsType(subclass$_n$).p.restExp else false)

Note that the resolution #S13 (or in combination of other resolutions), in complex situations, may lead to a final co-evolved constraint that might be hard to comprehend. Thus, as with any automatic activity the user might need to re-write and re-structure the constraint in seldom cases. We will further assess whether in our case studies such cases occurs.

3.4.6. Extract Class

An Extract Class is performed as follows: a new class B is added, and a single-valued reference r is added from a class A to class B. Then a set of properties $p_1, ..., p_n$ are deleted from A and added to B. It can be seen as a group of move properties through a single-valued reference. Thus, the resolution strategies are the same as the #S8 and #S9 for each p_i.

3.4.7. Inline Class

An Inline Class is the opposite of Extract class, a set of properties $p_1, ..., p_n$ are deleted from a class B and added to a class A. Then the class B (non-referenced by any other class) as well as the single-valued reference r are deleted.

Id : #S14
Context : B (container)
Location in the constraint : context
Description : The context of the constraint is updated, by changing the source class B to the target class A where B is inlined. References from B to A are deleted in *exp* (if used in *exp*).

context B::p_i init: exp => context ***A****::p_i init: exp*

Id : #S15
Context : n/a
Location in the constraint : navigation path
Description : The navigation path to the property p_i is reduced with the reference r.

Exp. ***r****.p_i.restExp => Exp.p_i.restExp*

Id : #S16
Context : n/a
Location in the constraint : n/a
Description : When the inlined class B is used as a type, it is changed to the target class A where B is inlined.

3.4.8. Flatten Hierarchy

This change consists of a deletion of a set of properties $p_1, ..., p_n$ from a superclass A, and their addition in the subclasses B_1,... B_n. Then the superclass A (non-referenced by any other class) is deleted. This change can be seen as a group of push properties from A to B_1,... B_n, before to delete A.

Id : #S17
Context : A (container)
Location in the constraint : n/a
Description : If the context is on the superclass A, then duplicate the constraint to the subclasses. The original constraint is then deleted.

Table 3 presents the metamodel changes that have an impact on OCL constraints that can be automatically resolved, and their associated resolutions while specifying the two new influencing factors. The changes are evaluated as impacting, if there exists at least one situation where the metamodel change impacts the syntactic correctness of the OCL constraints. The resolutions aim at restoring the syntactic correctness of the impacted OCL constraints while expressing the same OCL constraints. The idea of each resolution is to propagate the metamodel change knowledge on the OCL constraints. However, some alternative resolutions may express different semantics. In particular, for the change *Generalize Property Multiplicity (GPM) from a Single Value to Multiple Values*, the proposed resolutions (first, last, etc.) to access a collection of values have different semantics. Thus, the resulted OCL constraint still verifies the same original condition but either for the first value, for the last value, etc.

Among the 7 complex changes we consider (section 3.1), we analyzed the impact they can have on OCL constraints. Except for *pull property* and *extract super class*, they all have an impact that can be automatically resolved.

Instead of proposing a unique resolution per metamodel change, we propose the minimal number of resolutions that consider the three influencing

factors. In particular, we propose the minimal set of resolutions so that: 1) the alternative operations that the OCL language offers are all considered, e.g. the alternative iterators forAll() and exists(), and 2) an arbitrary impacted OCL constraint (with an arbitrary impacted location) can always be co-evolved. Note that our resolutions are designed independently from the types of the OCL expressions but rather as a response to the effect of the impacting metamodel changes. However, for each of the resolutions we simulated (during the design phase of the resolutions) their application on different OCL expressions with the different available types to identify cases where thy are not applicable. Further in our evaluation we did not face a case where a resolution strategy could not be applied. Nonetheless, we cannot generalize it to all possible constraints as stated in the threats to validity.

However, we do not attempt to define all possible resolutions. There will always be a situation in which the user might apply a manual resolution or a particular refactoring. This is handled in our approach by the *ignore option* since we allow the user to not apply a specific resolution when desired. Search-based techniques can be used to explore all possible solution space, i.e. all possible co-evolved OCL constraints as it is done for models' co-evolution [20, 11, 44]. In particular, a search-based technique can be used when the user chooses to ignore the proposed resolutions. This is left as future work. As shown in Table 3, for 8 metamodel changes we propose 17 resolutions, thus answering the requirement R2. Note that in our case studies these 17 resolutions covered the co-evolution of all the impacted OCL constraints.

3.5. Co-evolving Non-impacted OCL Constraints

Up to now we focused on resolving OCL constraints that are impacted by metamodel changes. As shown in Table 2 several changes do not impact OCL constraints and thus no resolution is associated to those changes. Nonetheless, a co-evolution in the form of a refactoring can be proposed to be applied on OCL constraints. Indeed, non-impacted OCL constraints can always be improved or even deleted in case they become meaningless. However, it is very challenging to first detect the non-impacted constraint that may need to be co-evolved since they are still valid constraints.

In the following we propose to co-evolve non-impacted OCL constraints with resolutions that are associated to *Pull property* and *Extract super class*. In fact, these two changes have the intention to make the properties more accessible to all subclasses. So while a constraint would stay valid, there could suddenly be instances of the property that are no more under the constraint that before did hold for all instances of the property. Thus, the additional

knowledge of these two changes can be valuable to co-evolve OCL constraints that use a pulled property or a property extracted in the superclass.

3.5.1. Pull Property

This change consists in an add of a property p in a superclass A and its deletion in subclasses B_1,..., B_n.

`Id :` #S18

`Context :` n/a

`Location in the constraint :` n/a

`Description :` The context of the constraint is updated, by changing the subclass B_i to the superclass A. This will make the constraint applicable to the superclass and all subclasses.

context B_i*::*p_j *init: exp => context* ***A****::*p_j *init: exp*

context B_i *inv:* $Exp.p_j.restExp$ *=> context* ***A*** *inv:* $Exp.p_j.restExp$

The above resolution #S18 should be applied if the constraint uses only the pulled property p from the subclass and does not use other properties that are not pulled from B_1,..., B_n. Otherwise, the constraint would become invalid since other non-pulled properties are missioning in the superclass.

3.5.2. Extract Super Class

This change consists of an addition of a set of properties $p_1, ..., p_n$ in a superclass A, and their deletion in the subclasses B_1,... B_n. The superclass A is a newly added class. This change can be seen a group of pull properties from B_1,... B_n to A, after adding A. For this change, the resolution strategy is the same as #S18.

Table 4 presents the resolutions that are associated to non-impacting metamodel changes. In the evaluation section, we will further assess whether the resolution #S18 is used in our case studies. This resolution is proposed to the user in the same manner the resolutions of Table 3 are proposed to the user. The next section illustrates the process of proposing the appropriate resolutions for each impacted OCL constraint.

3.6. Proposing Resolution Strategies

In our approach we define and implement a set of resolutions that can be applied during co-evolution. The resolutions and influencing factors were co-jointly defined in section 3.4.

Table 3: Resolutions proposed to co-evolve impacted OCL constraints.

Metamodel change *(Factor 1)*	Location in the OCL constraint *(Factor 2)*	Context *(Factor 3)*	Resolution strategies	Total N^o of proposed resolutions
◇ Rename element	n/a	n/a	#S1	1
◇ Delete element	n/a	n/a	#S2	1
◇ GPM from a single value to multiple values	navigation path	n/a	#S3 #S4 #S5 #S6 #S7	5
◇ Move property	context navigation path	container n/a	#S8 #S9 #S10	3
◇ Push property	context navigation path navigation path	container container not via the subclasses	#S11 #S12 #S13	3
◇ Extract class	context navigation path	container n/a	#S8 #S9	2
◇ Inline class	context navigation path	container n/a	#S14 #S15 #S16	3
◇ Flatten hierarchy	n/a	container	#S17	1

Table 4: Resolutions proposed to co-evolve non-impacted OCL constraints.

Metamodel change *(Factor 1)*	Location in the OCL constraint *(Factor 2)*	Context *(Factor 3)*	Resolution strategies	Total N^o of proposed resolutions
◇ Pull property	n/a	n/a	#S18	1
◇ Extract super class	n/a	n/a	#S18	1

Figure 4 depicts our process of selecting the appropriate resolutions. It starts with all implemented resolutions and excludes a subset of resolutions based on the influencing factors. The final subset of applicable resolutions is then proposed to the user. The first factor we consider to exclude resolutions is the metamodel change that reduces the possible applicable resolutions (step 1). For example, if we encounter a rename change we exclude the resolutions defined for other metamodel changes. After that, the impacted location is considered to also reduce the subset of the possible applicable resolutions (step 2). Finally, the context of the impacted constraint allows us to further reduce the resolutions to a final subset (step 3) that is proposed to the user who decides which one to apply. This answers the requirement R3.

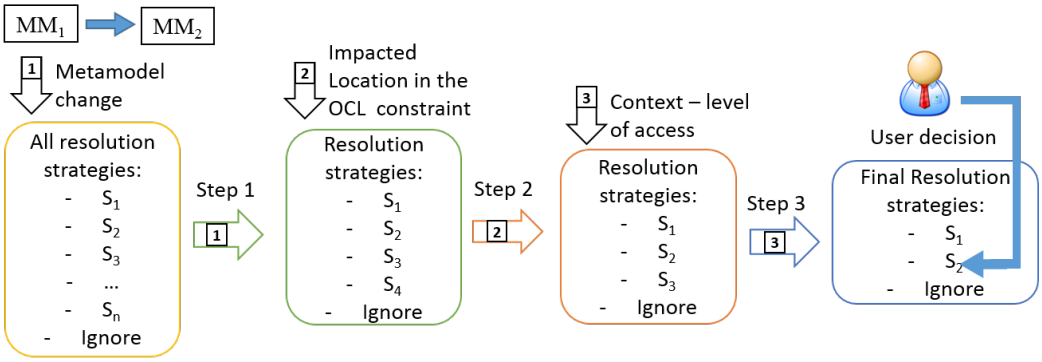


Figure 4: Process of selecting the appropriate resolution strategies per impacted part of a constraint and per metamodel change.

A constraint can be impacted in different parts, i.e. different AST nodes, by either the same or different metamodel changes. The process of Figure 4 is applied for each impacted part of an OCL constraint, i.e. for each tuple of {impacted OCL constraint $\times$ impacted AST node}.

Note that when a constraint is impacted by several metamodel changes, the resolutions are proposed and applied following the chronological order of the changes (detected in section 3.1). It ensures consistency in the co-evolution. For instance, suppose the OCL expression "self.p ≥ 0" is impacted by a move property *p* through a reference *ref*, followed by a rename *p* to *p'*. If the resolutions associated to the metamodel changes (respectively #S9 and #S1) are applied following the order of the changes, the constraint is correctly co-evolved (I) by first extending the navigation path of the property *p* before to be renamed. However, if the resolution #S1 is first applied, then #S9 cannot be applied (II) since *p* is no longer present in the constraint.

self.p ≥ 0 $\Rightarrow$ self.ref.p ≥ 0 $\Rightarrow$ self.ref.p' ≥ 0 ✓ (I)

self.p ≥ 0 $\Rightarrow$ self.p' ≥ 0 ✗ (II)

To remain flexible and to not introduce unintended solutions, our approach also proposes the possibility to *ignore* (in Figure 4) the proposed resolutions.

3.7. Automated Application of the Constraints' Resolutions

At this stage, we can propose, for each impacted part of an OCL constraint, a set of resolution strategies among which the user can choose. A resolution updates the AST by adding, removing, or updating nodes. Each resolution is implemented as a transformation function applied on the ASTs. Figure 5 depicts the co-evolution of the first constraint in Figure 2c to the first

constraint in Figure 2d at the AST level. The identified impacted AST node by the push property *depth* is represented with an arrow labeled "impacts" in Figure 5.

Some resolutions can be applied directly on the impacted AST node such as for a *rename*. Other resolutions can be applied on a subtree composing the OCL subexpression that includes the impacted AST node. The resolution for the pushed property *depth* cannot be applied on the AST node `AttributeCallExp` of *depth* alone. To this end, the first OCL subexpression in the AST containing the impacted AST node is identified on which the resolution strategy is applied locally.

In Figure 5, the subtree on which the resolution #S13 applies is surrounded by the dashed square. The resolution is represented by the gray nodes, and it consists in introducing an `If` expression that tests whether the current instance of the container is of type `Composite`. The `Then` branch contains the found subtree while introducing a conversion to `Composite` before to call the property *depth*.

As stated earlier, complex situations can occur when a single constraint is impacted several times requiring different resolutions. Although the resolutions are applied following the order of impacting changes, the user still controls which resolutions are applied among the alternative possible ones and hence controlling the possible complexity of the co-evolution process. Nonetheless, no complex co-evolution occurred in our evaluation although some constraints were impacted by several changes.

3.8. Improvement and Optimization of the Co-evolution

Throughout this section, except for the delete changes where we delete OCL constraints, we aim at resolving and maintaining the impacted OCL constraints. Nevertheless, the user may as well decide to delete an impacted constraint instead of maintaining it, for example by applying a rename strategy. Therefore, in the evaluation section, we will estimate whether proposing the delete strategy #S2 whenever an OCL constraint is impacted could increase the percentage of co-evolution w.r.t. the user intent.

Furthermore, in our co-evolution approach, after the user has chosen the resolution to be applied for each constraint, we first apply the delete resolutions #S2 before we apply the rest of the resolutions while following the chronological order of the changes as explained in section 3.6. In case a constraint is impacted by a delete and another metamodel change, this optimization allows us to not apply the additional resolution on the already deleted constraint, and thus improving the performances.

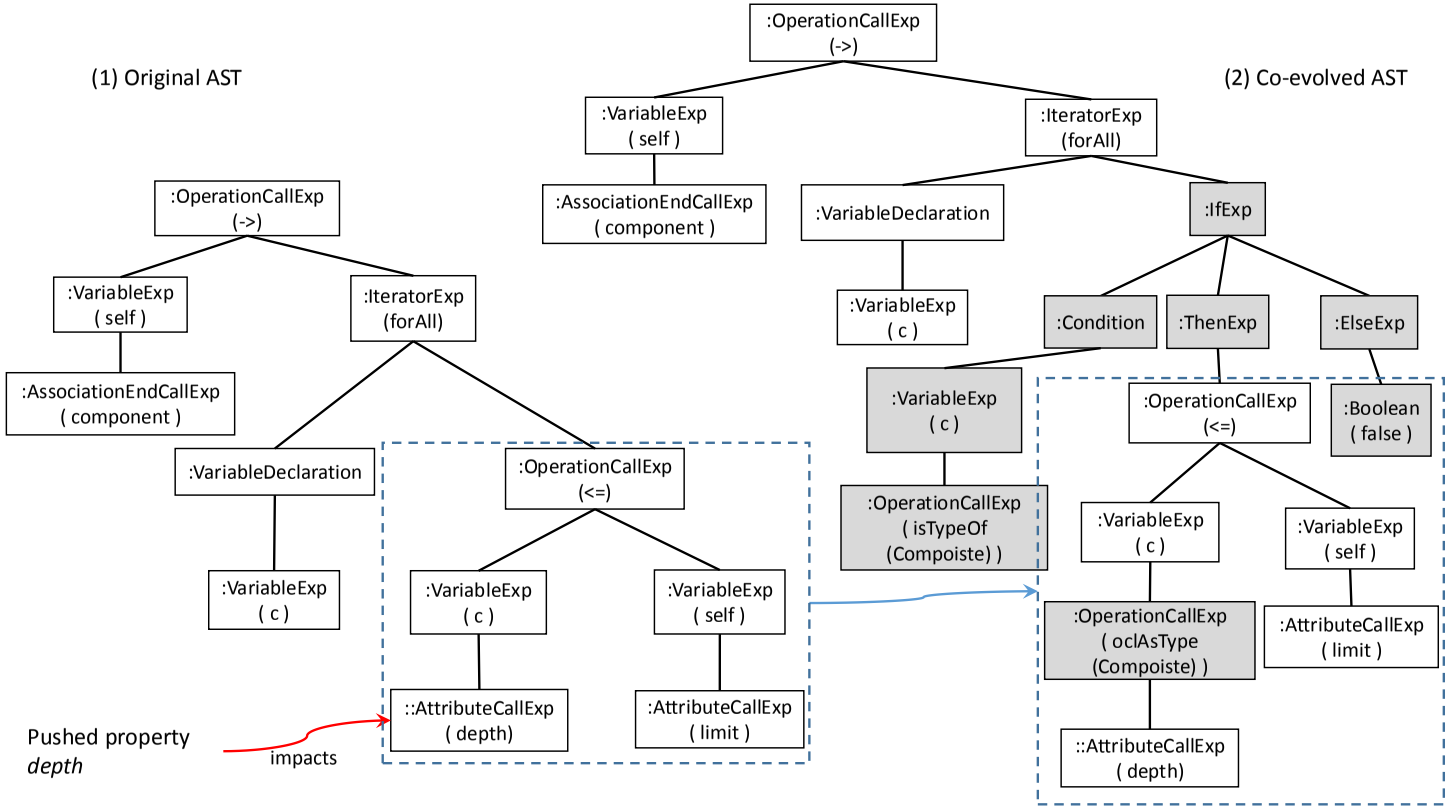


Figure 5: ASTs of the original and co-evolved OCL constraint

4. Implementation

Our tool manipulates Ecore/EMF metamodels and OCL files for the constraints. After identifying the metamodel evolution trace with the detection tool, the impact analysis on the OCL constraints is performed and for each impacted part we propose alternative resolutions. The user can then choose the appropriate resolution among the proposed ones or can decide to apply none of them. Then, our co-evolution engine applies the chosen resolution for each impacted part of an OCL constraint at the AST level. The core functionalities of this component are implemented with Java (4568 LoC) and are packaged into an Eclipse plugin that is chained with the detection plugin [23, 21].

Figure 6 displays a screenshot of our tool. Window (1) shows the OCL constraints that are co-evolved. In window (2) we present the impacted constraints and the cause of the impact in a textual message (the metamodel change and the location of the used element). In Window (3) a set of resolutions is proposed in a dropdown menu to the user along with the *ignore* option. Then, each resolution is applied to each impacted part of an OCL constraint.

5. Evaluation of the OCL constraints' co-evolution

This section presents a qualitative evaluation of our co-evolution approach. We first present our dataset and the evaluation process we follow. Then, we present the co-evolution of OCL constraints as it occurred in practice. After that, the co-evolution results when using our approach are illustrated. Finally, we compare our results against the ones in practice. Time performances of the co-evolution are measured as well.

Throughout this evaluation, we set three goals that are the following:

#G1: Demonstrate that alternative resolutions are required in practice.

#G2: Show that our 18 resolutions reach most of the user's needs.

#G3: Show that the set of initial metamodel changes we support already allows handling a realistic co-evolution scenario.

5.1. Dataset

At this stage of our research, we aim at validating empirically on 6 real case studies namely: Unified Modeling Language Class Diagram (UML CD) [38], Structured Metrics metamodel (SMM) [41], Unified Profile for

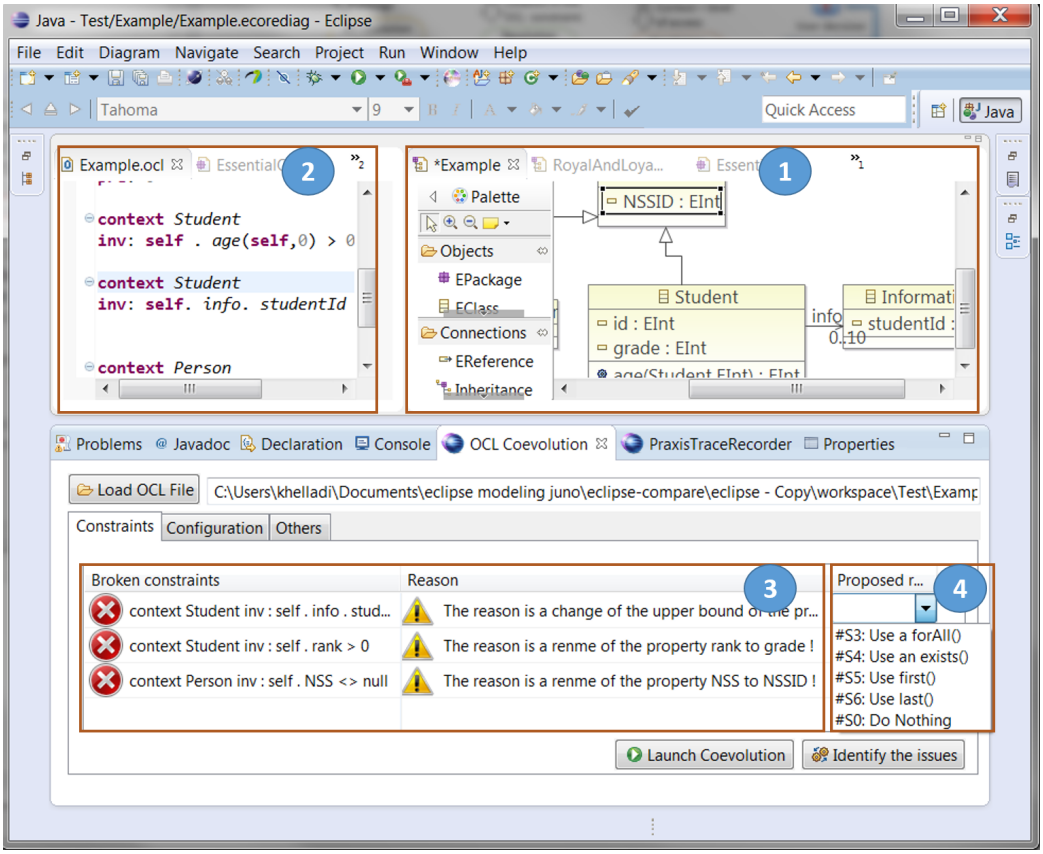


Figure 6: Screenshot of the Eclipse plugin Tool.

DoDAF/MoDAF (UPDM) [43], UML Profile for National Information Exchange Model (NIEM) [42], the EXPRESS Information Modeling Language (EXPRESS) [39], and the Requirements Interchange Format (ReqIF) [40].

In this evaluation, we have studied variety of metamodels of modeling languages from the OMG standards to metamodel profiles that are proposed by the OMG [34]. We were interested in finding in practice OCL constraints that are co-evolved manually in at least two versions so that we can compare them to our semi-automatic co-evolution of the same constraints. In so doing, we show the capability of our current approach of OCL co-evolution. We collected the OCL constraints from the different versions. We put the constraints into a canonical form, e.g. by adding the keyword "self" to remove any ambiguity. We also corrected errors in some constraints.

Since we rely on metamodel change for the co-evolution, we also manually studied the evolution of the different metamodels UML CD, SMM,

Table 5: Co-evolution case studies.

	UML CD		SMM		UPDM		NIEM		EXPRESS		ReqIF	
	1.5	2.0	3.2.2	3.4.4	1.0	2.1	1.0	3.0	1.0	1.1	1.0	1.1
Number of OCL constraints	73	110	59	87	29	39	49	135	60	58	26	25

UPDM, NIEM, EXPRESS, and ReqIF to determine the atomic and complex changes. Thus, in our evaluation we re-applied those changes on the original metamodel to detect them with our tool [23]. This particular aspect will be discussed as a threat to validity. Alternatively, those changes can also be detected with a difference-based technique (e.g., EMF Compare [48], UMLDiff [51], or DSMDiff [29] etc.).

Table 5 shows the selected case studies and the number of the OCL constraints in each of the studies version.

5.2. Process of the Evaluation

To run this experiment, we first detect atomic and complex changes that occurred in the metamodels' evolution from version *n* until version *n+1*. After that, based on the evolution changes we identify the impacted OCL constraints and we co-evolve them from version *n* until version *n+1* as well.

In this experiment, we measured the accuracy of our co-evolution tool by comparing for the same set of constraints how they are manually co-evolved in practice against how they are automatically co-evolved by our tool. This allows us to measure the precision of our co-evolution approach.

Furthermore, we will assess whether our resolution strategy #S18 can be helpful to co-evolve non-impacted OCL constraints. Finally, we also evaluate whether always proposing the delete strategy #S2 among the alternative ones would increase the co-evolution percentage (i.e., whether it is useful).

5.3. Co-Evolution Results as Occurred in Practice

As a first step of our evaluation, we studied how the OCL constraints are manually co-evolved in practice in response to the metamodel evolution. In order to study how the OCL constraints are co-evolved accordingly in practice, we followed the next procedure:

- Identify non-impacted constraints in the original version that are present in the new version.

- Identify non-impacted constraints in the original version that are deleted in the new version.
- Identify impacted constraints in the original version that should be co-evolved.
- For these impacted constraints, we systematically verify in the new version whether it exists a constraint that:
  - Has the same objective. We judge based on the comments describing the constraint's purpose (given in the specification). We also check returned type equality.
  - Has the same structure, using similar OCL operators, and/or using the same metamodel elements.
  - Final decision is made manually:
    1. If no constraint is found, we consider the impacted constraint to be deleted.
    2. If a constraint is found, we consider it to be the co-evolved version of the impacted constraint.
- Identify new constraints added in the evolved version.

The results of our analysis are given in Table 7. It gives the number of non-impacted and impacted constraints fro each case study.

5.4. Co-Evolution' Results by our Approach

We first detected with our tool the evolution traces of the UML CD, SMM, UPDM, NIEM, EXPRESS, and ReqIF metamodels that are given as input to our co-evolution tool. In the experiment, the authors play the user role by selecting the resolutions to apply. We aimed at co-evolving the OCL constraints *as close as possible* to the co-evolution in practice while also avoiding the use of the ignore solution (for an objective comparison in the next section). The goal is to assess to what extent our approach can get near to the user intent when co-evolving OCL constraints. Hence, after that we could compare to what extent our approach cover the manual co-evolution of our case studies. Note that we did not apply randomly the resolutions. The results of our applied co-evolution are presented in Table 8 that will be compared to the results of Table 7 in the next section.

Table 6: Number of applied resolutions in our co-evolution for each case study.

Case studies	Applied resolutions	N^o of application
UML CD	◇ Delete #S2	28
	◇ Rename #S1	34
	◇ Add iterator #S3	2
	◇ Push property #S12	1
	◇ Push property #S13	3
	◇ Inline class #S15	3
	◇ Inline class #S16	4
SMM	◇ Delete #S2	1
	◇ Rename #S1	29
	◇ Extract class #S9	17
UPDM	◇ Delete #S2	9
	◇ Rename #S1	3
NIEM	◇ Delete #S2	15
EXPRESS	◇ Delete #S2	4
	◇ Rename #S1	3
ReqIF	◇ Delete #S2	1
	◇ Rename #S1	1

Table 6 displays the applied resolution strategies for each case study during our co-evolution. In most of the case studies, several constraints are impacted by more than one metamodel change. Thus, more than one resolution is applied for several constraints, in particular in the UML CD, SMM, and EXPRESS case studies. For instance, the rename #S1 (see Table 3) is applied several times along with #S1, #S3, #S9 #S12, #S13, #S15, or #S16 on the same constraint. As mentioned previously in section 3.6, the resolutions are applied following the chronological order of the detected metamodel changes in the evolution trace. Since the authors evolved themselves the metamodels, the applied order of changes may have an influence on the co-evolved constraints. For the constraints that were impacted by multiple changes, we investigated whether changing the order of changes can lead to different versions of co-evolved constraints (e.g., comparing the effect of a *rename + move* and *move + rename* changes). This was not the case in our case studies, where changing the order of changes (and applying the resolutions in accordance to each of the different orders) resulted in the same

constraints. This means that using a difference-based technique to detect the changes would not affect the co-evolution result in our case studies. Nonetheless, we cannot generalize it to other case studies. Note that apart from the above cases we did not face complex situations where an impacted constraint was not co-evolved properly.

5.4.1. Performances.

Similarly as for the detection, we ran our experiment on a PC VAIO with an i7 1.80 GHz Processor and 8GB of RAM on Windows 7. The evaluation experiment using our approach for OCL co-evolution returned instant runs. In particular, after selecting the resolutions to be applied among the proposed ones, all impacted OCL constraints were co-evolved in less than 1 second in each of the case studies.

5.5. Comparison of the observed and obtained Results: "in Practice" VS "our Approach"

This section discusses the evaluation's results and compares the observed co-evolution in practice with the obtained OCL co-evolved constraints by our tool.

5.5.1. UML CD case study

Following the procedure described in section 3.2 we were able to identify all the 54 impacted constraints.

Table 7: Co-Evolution of the OCL constraints as they occurred in practice.

Co-evolution of OCL constraints in practice	UML CD 1.5 to 2.0	SMM 1.0 to 1.1	UPDM 1.0 to 2.1	NIEM 1.0 to 3.0	EXPRESS 1.0 to 1.1	ReqIF 1.0 to 1.1
◇ Constraints that are not impacted	19	8	17	34	55	24
◇ Constraints not impacted and present in both versions	7	8	17	11	55	24
◇ Constraints not impacted and deleted in the new version	12	0	0	23	0	0
◇ Constraints impacted in the first version (to be co-evolved)	54	12	12	15	5	2
◇ Constraints co-evolved by deletion in the new version	35	1	9	15	4	1
◇ Constraints co-evolved and present in the new version	19	11	3	0	1	1
◇ New constraints in the new version	84	13	19	124	2	0

Table 8: Co-evolution of OCL constraints by our approach

Co-evolution of OCL constraints by our approach	UML CD 1.5 to 2.0	SMM 1.0 to 1.1	UPDM 1.0 to 2.1	NIEM 1.0 to 3.0	EXPRESS 1.0 to 1.1	ReqIF 1.0 to 1.1
◇ Constraints that are not impacted	19	8	17	34	55	24
◇ Constraints impacted in the first version (to be co-evolved)	54	12	12	15	5	2
◇ Constraints co-evolved by deletion	28	1	9	15	4	1
◇ Constraints co-evolved by other resolution strategies	26	11	3	0	1	1

Deleted constraints. Among those 54 constraints, 28 constraints are co-evolved by deletion. Indeed, some properties and/or classes used in those 28 constraints were deleted during the metamodel evolution. Those 28 deleted constraints were also deleted in the real case study, i.e. they are included in the 35 deleted constraints in practice.

Undeleted constraints. Using our approach we co-evolved 26 constraints with various resolutions as displayed in Table 6. Among those 26 constraints, 19 constraints are co-evolved in our approach that correspond to the 19 co-evolved constraints in practice. Moreover, 7 constraints are co-evolved in our approach and correspond to the 7 impacted constraints that were deleted in practice.

Regarding the 19 constraints that are co-evolved with our tool, 11 co-evolved constraints are syntactically equal to 11 of the the 19 constraints that resulted from the co-evolution in practice. Additional 4 constraints are not syntactically but are semantically equal to 4 of the 19 constraints that are co-evolved in practice. Thus, 15 constraints have been correctly co-evolved using our tool. E.g. one definition "def: allConnections: ..." is changed to "def: allConnections(): ..." by adding the "()" in practice.

However, the last 4 constraints are non-syntactically and non-semantically matching. They are refactored in practice with a different semantic. For example, one original constraint that checks absence of circular inheritance is impacted by the renaming of `GeneralizableElement` to `PackageableElement`. It is co-evolved by our approach as follows from (1) to (2) by applying the rename strategy #S1.

context `GeneralizableElement` inv: not self.allParents()−> includes(self) (1)
context `PackageableElement` inv: not self.allParents()−> includes(self) (2)

In practice the context of constraint (2) was changed to the subclass `Classifier` instead of applying a rename. Therefore, the semantic is slightly changed by the manual co-evolution since the applicability scope of the new constraint is reduced to elements of type `Classifier`. The rates of both syntactically and semantically correct co-evolution are respectively 72% and 80%[5].

Maintained constraints. In our approach, 7 impacted constraints (referred to them later as $\boxed{A}$) are co-evolved whereas they were deleted in practice . We applied 8 times the rename strategy #S1 for six of the constraints

[5]% = ((deleted constraints + syntactically (respectively semantically) correct co-evolved constraints) / impacted constraints)

and 1 time the strategy #S16 of an inline class for one constraint. Thus, only 35% (19/54) of the impacted constraints are maintained in practice, while 48% (26/54) of the impacted constraints are maintained in our approach. For example, constraint (3) is an operation defined on a `ModelElement` returning a set of all direct suppliers of the `ModelElement`; it is impacted by the rename of `ModelElement` to `NamedElement`. We co-evolved it to (4) simply by applying the rename strategy #S1.

context `ModelElement` def: supplier : (3)
Set(*ModelElement*) = self.clientDependency.supplier

context `NamedElement` def: supplier : (4)
Set(*NamedElement*) = self.clientDependency.supplier

Furthermore, we noticed the deletion of 12 non-impacted constraints (referred to them as $\boxed{B}$). For example, constraint (5) expressing that an interface can only contain operations is deleted although it is not impacted.

context `Interface` inv: (5)
self.allFeatures()−>forAll(f | f.oclIsKindOf(Operation))

5.5.2. SMM case study

Following the procedure described in section 3.2 all the 12 impacted constraints were identified and successfully co-evolved.

Deleted constraints. Only 1 constraint is co-evolved by deletion due to the delete class `RecursiveMeasureRelationship`.

Undeleted constraints. Our tool also co-evolved 11 constraints that are impacted by several renames of properties and/or classes, and by one extract class. Those 11 constraints correspond to the 11 manually co-evolved constraints in practice. 9 co-evolved constraints are syntactically equal to 9 of the the 11 constraints that resulted from the co-evolution in practice.

However, 2 constraints are non-syntactically and non-semantically matching. Those 2 constraints are modified and further refactored in practice. For example, the following constraint (6) is impacted by an extract class of the property *measurand* from class `Measurement` to `ObservedMeasure` through the reference *observedMeasure*, and by the rename of *measurand* to *measure*. It is co-evolved by our tool as follows from (6) to (7) by extending the navigation path of *measurand* to *observedMeasure.measurand* (i.e. #S9) before renaming the property (i.e. #S1). Note that this constraint is impacted in three locations by the same above changes, and thus each of #S9 and is applied three times.

context `RescaledMeasurement` inv: (6)
self.*measurand*.oclIsTypeOf(RescaledMeasure) and

self.isBaseSupplied implies not self.baseMeasurement−>isEmpty()
and (self.baseMeasurement.*measurand* = self.*measurand*.baseMeasure)

context `RescaledMeasurement` inv: (7)

self.*observedMeasure.measure*.oclIsTypeOf(RescaledMeasure) and
self.isBaseSupplied implies not self.baseMeasurement−>isEmpty()
and (self.baseMeasurement.*observedMeasure.measure* =
self.*observedMeasure.measure*.baseMeasure)

In practice the property *baseMeasurement* in the constraint (7) is replaced by the property *rescaleFrom*, resulting in a semantically different constraint. Thus, both the rates of syntactically and semantically correct co-evolution are 83%.

5.5.3. UPDM case study

We identified all the 12 impacted constraints in the UPDM case study following the procedure of section 3.2.

Deleted constraints. Among the 12 impacted constraints, 9 constraints are co-evolved by deletion, mainly due to a delete of several classes (e.g. `PhysicalLocation`, `OrganizationalExchange`, `OperationalNode`, `ArtifactResourceArtifact`).

Undeleted constraints. Our approach also co-evolved 3 constraints by applying three times the rename resolution #S1. For instance, the constraint (8) is co-evolved to (9) by renaming the class Location to LocationType.

context `uml::DataType` inv: (8)

UPDM::UPDML1::UPDML0::Core::AllElements::Environment::
Location.allInstances()−>exists(n|n.base_DataType=self)
implies SysML::ValueType.allInstances()−>exists(b|b.base_DataType = self)

context `uml::DataType` inv: (9)

UPDM::UPDML1::UPDML0::Core::AllElements::Environment::
LocationType.allInstances()−>exists(n|n.base_DataType=self)
implies SysML::ValueType.allInstances()−>exists(b|b.base_DataType = self)

The impacted OCL constraints are co-evolved similarly in our tool as in practice. Thus, both the rates of syntactically and semantically correct co-evolution are 100%.

5.5.4. NIEM case study

Following the procedure described in section 3.2 all the 15 impacted constraints were identified and successfully co-evolved.

Deleted constraints. In the NIEM case study, all 15 impacted constraints are co-evolved by deletion due to a delete of sev-

eral classes and properties (e.g. `ModelPackageDescriptionFile`, `ModelPackageDescriptionFileSet`). Thus, both the rates of syntactically and semantically correct co-evolution are 100%.

Maintained constraints. Furthermore, similarly as in UML CD case study, it is surprising to find deletion of 23 non-impacted OCL constraints (referred to them as $\boxed{C}$). For example, constraint (10) expressing that the ownedAttributes must have a multiplicity of "0..1" is deleted.

context `Choice` inv: (10)
self.base_Class.ownedAttribute−>forAll(a|(a.lower=0) and(a.upper=1))

5.5.5. EXPRESS case study

In the EXPRESS case study we were able to identify all the 5 impacted constraints by following the procedure of section 3.2.

Deleted constraints. Among the 5 impacted OCL constraints, 4 constraints are co-evolved by deletion, mainly due to the removal of the two properties *ofentity* and *playsdomainrole*.

Undeleted constraints. Only 1 constraint is co-evolved from (11) to (12) with a rename resolution #S1 due to a rename of the property *formalparameter* to *forparameter*. This constraint is impacted three times by the same change, and hence the resolution #S1 is applied three times on the same constraint.

context `ActualParameter::formalparameter : Parameter` derive: (11)
if self.oclIsTypeOf(PassByValue) then
self.oclAsType(PassByValue).formalparameter
else self.oclAsType(ExpressMetamodel::Statements::PassByReference)
.formalparameter endif

context `ActualParameter::forparameter : Parameter` derive: (12)
if self.oclIsTypeOf(PassByValue) then
self.oclAsType(PassByValue).forparameter
else self.oclAsType(ExpressMetamodel::Statements::PassByReference)
.forparameter endif

The impacted OCL constraints are co-evolved similarly in our tool as in practice. Thus, both the rates of syntactically and semantically correct co-evolution are 100%.

5.5.6. ReqIF case study

Following the procedure described in section 3.2 the 2 impacted constraints were identified and successfully co-evolved.

Table 9: Correct Co-evolution Percentage (%).

	UML CD	SMM	UPDM	NIEM	EXPRESS	ReqIF
Syntactically correct co-evolution %	72%	83%	100%	100%	100%	100%
Semantically correct co-evolution %	80%	83%	100%	100%	100%	100%

Deleted constraints. Only 1 OCL constraints is co-evolved by deletion due a delete of the property *defaultValue*.

Undeleted constraints. Only 1 constraint is co-evolved with a rename resolution #S1 due to a rename of the property *maxlength* to *maxLength*.

The impacted OCL constraints are co-evolved similarly in our tool as in practice. Thus, both the rates of syntactically and semantically correct co-evolution are 100%.

5.6. Discussion

This section presents a comparison with the existing works and discusses the evaluation results and observations.

5.6.1. Comparison with existing co-evolution approaches

In this work, we proposed an approach for OCL constraints' co-evolution when the metamodel evolves. Table 9 highlights the results of the measured co-evolution percentages. For four case studies, we reached 100% co-evolution percentage, whereas in UML CD and SMM, we had respectively 72%, 80% and 83% of co-evolution percentage. An average of 92% (syntactically) and 93% (semantically) co-evolution equivalent to the actual co-evolved constraints. These results show that our approach can support the user with a co-evolution near to her intent and needs.

Furthermore, we wanted to compare our work with existing approaches. In particular, to compare the co-evolution percentage of the different approaches on our case studies. We simulated the co-evolution of each approach on our case studies based on the clear content of the paper and the following procedure:

1. We identified the considered *changes* and proposed *resolutions* for each existing approaches.

2. For all impacted constraints in our case studies, we assessed whether they can be co-evolved by the existing approaches, i.e., whether they are detected and most importantly whether they can be resolved.

Table 10: Comparison with existing OCL co-evolution approaches

Approaches	UML CD 1.5 to 2.0	SMM 1.0 to 1.1	UPDM 1.0 to 2.1	NIEM 1.0 to 3.0	EXPRESS 1.0 to 1.1	ReqIF 1.0 to 1.1
Demuth et al. [9, 10]	59%	25%	100%	100%	100%	100%
Hassam et al. [13]	6%	0%	0%	0%	0%	0%
Kusel et al. [25]	7%-17%	17%	25%	0%	20%	50%
Markovic et al. [30, 31]	7%-17%	17%	25%	0%	20%	50%
Cabot et al. [6]	52%	9%	75%	100%	80%	50%
Current approach	72%-80%	83%	100%	100%	100%	100%

3. If no, the impacted constraints are considered as non co-evolved.

4. If yes, we co-evolve manually the impacted constraints as it is described in the exiting approaches (i.e., their respective papers).

5. We finally computed the co-evolution percentage for all existing works.

Following the above procedure we simulated the automatic co-evolution of the existing works and computed their co-evolution percentages. Table 10 shows the comparative results. Note that if a change `C` is not addressed by an existing approach, we classify the impacted OCL constraint by the change `C` as not co-evolved. For example, Hassam et al. [13] did not consider the changes *delete* and *rename* elements and did not proposed resolutions for them, although mentioned in [13]. Thus, impacted constraints by *delete* and *rename* elements are counted as non-co-evolved in this case, which is why Hassam et al. [13] reached 0% in some of our case studies where OCL constraints were only impacted by *delete* and *rename* changes. Table 10 shows that our reached co-evolution percentage exceeds the one simulated for the existing approaches if they were used on the same case study. Overall results shows the benefit of our approach, although we cannot generalize the comparative observations. Further comparison of the existing approaches will be discussed in the related work section 6.

5.6.2. Alternative resolutions, maintained and non-necessary deleted constraints

In the following we discuss the observed results in our evaluation. First of all, as shown in Table 6, multiple resolutions are used for the metamodel changes *push property* (#S12, #S13) and *inline class* (#S15, #S16), in particular for UML CD. These results emphasize and confirm the usefulness

to propose alternative resolutions in order to cope with realistic scenarios of OCL constraints' co-evolution. Otherwise, not considering them would reduce the co-evolution percentage with a risk of introducing inappropriate solutions.

Moreover, we observed interesting 7 cases of maintained constraints $\boxed{A}$ by our tool in the UML CD. From our point of view, it is surprising to delete a constraint, whereas it would have been possible to rename the impacted element or to apply another resolution. One possible explanation is that the constraints became meaningless in the new version of the metamodel. Another arguable explanation is that the lack of a (semi) automated support for co-evolution was the cause of the loss of those constraints. Otherwise, they would have been easily maintained in the new version. Those first 7 cases of maintained constraints $\boxed{A}$ in the UML CD underline the need to also propose the delete strategy #S2 whenever a constraint is impacted, and not always try to maintain the constraint.

We also observed cases of unnecessary removals of constraints that were not impacted. The second 12 cases of deleted constraints $\boxed{B}$ in UML CD and 23 deleted constraints $\boxed{C}$ in NIEM are examples of unnecessary removals. Similarly, a possible explanation is that those constraints are no more necessary. As a further investigation, we had a look at later versions of UML CD specifications (versions 2.1, 2.2, and 2.3), and the constraints are indeed missing[6]. Those cases $\boxed{B}$ and $\boxed{C}$ emphasize the fact that even if all impacted constraints are correctly co-evolved, user intervention would still be needed to decide whether to keep or to remove some of the non-impacted constraints.

As mentioned in section 3.8, we also are interested to assess whether always proposing the delete resolution would increase the co-evolution percentage. In our case studies, apart from the UML CD, the co-evolution percentage in the rest of the case studies remain unchanged. However, in the UML CD by deleting the 7 maintained constraints $\boxed{A}$ instead of co-evolving them, the co-evolution percentage significantly improves from 72% to 85% of syntactically correct co-evolution[7], and from 80% to 93% of semantically

[6]For NIEM case study no new version is currently available after version 3.0.

[7]85% = ((28 + 7 deleted constraints + 11 syntactically correct co-evolved constraints) / 54 impacted constraints)

Table 11: Number of refactorings on non-impacted OCL constraints.

	UML CD	SMM	UPDM	NIEM	EXPRESS	ReqIF
N^o or refactorings of non-impacted constraints	1	1	2	9	1	0

correct co-evolution[8]. This confirms our hypothesis that always proposing the resolution #S2 among the existing alternatives can be beneficial for the user in some cases when used appropriately.

5.6.3. Semantically not matching constraints

It is worth noting that the cases of the semantically not matching constraints obtained in UML CD (e.g. constraint (1)) and in SMM (e.g. constraint (6)) underline the need to let the user *ignore* a proposed co-evolution. By not applying a particular resolution, the user can manually co-evolve it and further refactors it w.r.t. her intent. We further analyzed these cases of semantically not matching constraints to understand the reasons. We observed two reasons *1)* either the context was changed or *2)* new OCL subexpression with new elements are added to the constraints. The former could be handled by changing the context. However, navigation paths must be carefully updated to still access the properties in the old context, as we do in resolution #S8 (see section 3.4). This issue is detailed and addressed by Cabot et al. [7]. The latter, on the contrary is hard to be handled automatically by a co-evolution approach. The new added parts are the result of the engineering knowledge that cannot be grasped only with the metamodel changes during evolution. Manual intervention remains necessary in this case.

5.6.4. Non-impacted constraints and refactorings

Furthermore, in our case studies we also studied the non-impacted OCL constraints to evaluate whether they were refactored and changed. Table 11 shows the number of refactored non-impacted OCL constraints that we encountered in our case studies.

Among the non-impacted OCL constraints one constraint in the SMM case study was refactored with the resolution #S18 that we proposed in

[8]93% = ((28 + 7 deleted constraints + 15 semantically correct co-evolved constraints) / 54 impacted constraints)

Section 3.5. Indeed, a complex change Extract super class occurred from the class `RankingInterval` to `Interval`. The constraint (13) was refactored to (14) by changing the context to the superclass.

context `RankingInterval` derive: (13)
(self.maximumEndpoint >= self.minimumEndpoint) and
((self.maximumOpen or self.minimumOpen) implies)
self.maximumEndpoint > self.minimumEndpoint)
context `Interval` derive: (14)
(self.maximumEndpoint >= self.minimumEndpoint) and
((self.maximumOpen or self.minimumOpen) implies
self.maximumEndpoint > self.minimumEndpoint)

Although the above example shows the usefulness of the resolution #S18, more case studies are required to assess the occurrence frequency of #S18 and it real benefit.

Moreover, in UML CD, UPDM, NIEM, EXPRESS case studies, several cases of non-impacted constraints' refactoring were observed. For instance, in the EXPRESS case study the constraint (15) expressing that there should be a unique instance of `Indeterminate`, was refactored to (16).

context `Indeterminate` derive: (15)
self = Indeterminate.allInstances()->asOrderedSet()->at(1)
context `Indeterminate` derive: (16)
Indeterminate.allInstances()->size() = 1

Another type of refactoring was observed in the NIEM case study where several constraints were merged into a single one. For example, the constraint (17) and (18) were refactored to (19) by merging them with the logical operator *"and"*.

context `Union` derive: (17)
self = self.base_DataType.generalization->isEmpty()
context `Union` derive: (18)
self.base_DataType.ownedAttribute->isEmpty()
context `Union` derive: (19)
self.base_DataType.attribute->isEmpty() and
self.base_DataType.generalization->isEmpty()

At this stage of our research, we only observed those refactorings, but we still lack enough case studies to better understand the reasons of those refactorings and their benefits. Nevertheless, we can observe that some refactorings changed the semantic of the constraint whereas others did not. For example, the constraint (15) in the specification document aimed at express-

ing the uniqueness of the `Indeterminate`'s instance. However, the original constraint checks only that the current instance is equal to the first instance in the set of all instances of `Indeterminate` class. This constraints does not verify the uniqueness characteristic. Therefore, the constraints was refactored to (16) to clearly express that there is only one instance of `Indeterminate` class. As a future work, this refactoring can be captured as a pattern to be reused in other case studies, but there is no guarantee of its usefulness in practice. As for the constraints (17) and (18), this refactoring could be applied to all constraints that share the same context. Yet this refactoring may be explained by the different writing styles of the software engineers, since both constraints (17) and (18) are equivalent to (19).

In this paper we mainly focused on co-evolving impacted OCL constraints so they can be reused in the evolved metamodel version. These cases of non-impacted constraints' co-evolution emphasize the need to examine the challenging topic of co-evolving non-impacted OCL constraints.

5.7. Threats to Validity

We now discuss internal, external, and conclusion threats to validity after Wohlin et al. [50] w.r.t. our three evaluation goals #G1, #G2, and #G3.

5.7.1. Internal Validity.

During the analysis of the co-evolution of the OCL constraints in practice, it is possible that we could have missed a correspondence between an original constraint and a co-evolved constraint when the latter is subject to a strong refactoring. To reduce this risk, in the procedure of our analysis for each impacted constraint, we investigated the constraints of the new versions one by one to avoid missing any correspondence. Moreover, other complex changes than the 7 ones we considered may occur in the evolution requiring additional resolutions that are not in our 18 resolutions. However, in our evaluation the 7 complex changes we considered as well as the 18 resolutions were sufficient in our case studies.

Furthermore, in our evaluation the authors played the user role by selecting the resolutions to be applied during the co-evolution of the OCL constraints. This would deliver better results since the authors are familiar with the case studies. However, as our goal was to assess to what extent our approach can get near to user intent when co-evolving OCL constraints, we aimed at reproducing the user intention with our resolutions as closely as possible. Hence, after that we could compare to what extent our approach

covers the user manual co-evolution in our case studies. Therefore, this threat is acceptable here.

5.7.2. External Validity.

We evaluated our tool on six case studies of metamodels and their OCL constraints. However, it is difficult to generalize our obtained results for other metamodels and OCL constraints. Nonetheless, the UML CD, SMM, UPDM, NIEM, EXPRESS, and ReqIF case studies provided a representative and a complex evolution trace that had a significant impact on the OCL constraints. In future work, we plan to evaluate our approach on an industrial case study that will be provided by our industrial partners.

5.7.3. Conclusion Validity.

Our evaluation gives promising results demonstrating that alternative resolutions are used in real life cases. The results also indicate that our 18 resolutions are semantically close to the user needs during the co-evolution. Thus, our evaluation's results meet our goals #G1 and #G2. However, we cannot estimate the quality of the resolutions only based on our six case studies. Third goal #G3 is also met since our co-evolution tool covers all metamodel changes that occurred in the evolutions of our case studies. Yet, more experiments are necessary to retrieve a more precise measure of the resolutions' quality, their occurrence frequency, and the benefit of the ignore option in practice.

6. Related Work

In this section, we present the main approaches that co-evolve OCL constraints and we discuss them w.r.t. our current work.

Demuth et al. [9, 10] proposed an approach for OCL co-evolution based on templates. They provided 11 templates that define a fixed structure for OCL constraints. The templates are then re-instantiated to generate the co-evolved OCL constraints. However, their approach is not applicable for arbitrary OCL constraints, and is limited to 11 templates only. They do not handle metamodel changes that impact the structure of the constraints. The impact analysis and the co-evolution of the 11 templates (i.e. constraints) is performed during the metamodel evolution after each change.

Hassam et al. [13] proposed a tool *METAEVOL* to co-evolve OCL constraints using Query/View/Transformation (QVT) [37], a transformation

language. The impact analysis is performed during the metamodel evolution, however, the OCL co-evolution is done at the end. To manage the constraints states, i.e. whether they are impacted and whether they are co-evolved, an intermediate table is used for this purpose. However, it is unclear whether the metamodel changes are recorded at run-time, or are manually specified by the user.

Markovic et al. [30, 31] proposed to refactor, based on QVT, OCL constraints annotated on UML class diagrams when these last evolve. Although their approach focuses on the model level with UML class diagrams, the theoretical basis also apply for OCL constraints defined on metamodels. This approach, proposes to apply a refactoring of the UML CD and immediately after it a co-evolution of the impacted OCL constraints.

Kusel et al. [26] first analyzed the impact of metamodel evolution on OCL expressions, and then proposed to resolve the impacted OCL expressions in [25]. However, they do not consider an OCL constraint as a whole but only its body. In particular, the context is ignored whereas it can be the impacted part (by a rename class for example) that requires a resolution.

Cabot et al. [6] focused on a single metamodel change, the delete element. In particular, they aimed at preserving the impacted constraints instead of deleting them, by removing only a sub part of the OCL constraint that is using the deleted element. However, the approach is applicable only to OCL constraints written in the form of Conjunctive Normal Form (CNF). In addition, Cabot et al. [7] studied the different possibilities of changing the constraint's context while maintaining the constraints.

Buttner et al. [5] discussed the impact of changing the multiplicity of a property in the metamodel on OCL constraints, which is addressed in our resolutions (see section 3.4). In addition, transformation languages like ATL share many similarities with OCL, and hence co-evolution approaches such as Garcia et al. [12] can be to a certain extent similar to OCL co-evolution.

Three OCL co-evolution approaches are *offline*[9] [13, 25, 6] including ours, and two are *online*[10] approaches [9, 10, 30, 31]. Note that *offline* co-evolutions, including ours, do apply an *offline impact analysis*[11]. Similarly with *online* co-evolutions that apply an *online impact analysis*[12]. Hassam

[9] Offline: The co-evolution is performed after finishing the metamodel evolution.
[10] Online: The co-evolution is performed while evolving the metamodel.
[11] Offline ia: impacted OCL constraints are identified after the metamodel evolution.
[12] Online ia: impacted OCL constraints are identified while evolving the metamodel.

et al. [13] is the unique *offline* co-evolution approach that applies an *online impact analysis*.

As shown in Table 12 among the existing approaches only [9, 10, 25] consider all atomic changes, whereas [13, 30, 31, 6] consider a subset of atomic changes only. In this work, we also consider all set of atomic changes.

Moreover, only [9, 10, 6] do not consider complex changes. The complex changes that are considered by the rest of the approaches differs for each of them [13, 25, 30, 31], as highlighted in the fifth column of Table 12. In our work we initially consider seven well-known and most used complex changes (see section 3.1).

Regarding the degree of automation, Demuth et al. [9, 10], Markovic et al. [30, 31], and Cabot et al. [6] all apply a fully automatic co-evolution. Hassam et al. [13], Kusel et al. [25], and our current approach are all semi-automatic requiring user decision to confirm which resolutions to apply and whether to apply a resolution.

All existing approaches [9, 10, 13, 30, 31, 6, 5, 25] consider only the metamodel change as a factor to propose a resolution. Thus, they define for each metamodel change only a unique resolution that will be applied to any arbitrary impacted OCL constraint. Only Cabot et al. [6] partially addresses the issue of multiple resolutions by either deleting the constraints or just deleting part of them, i.e. the parts using the deleted element. In contrast, we identified two additional factors (see section 2) that lead to propose multiple resolutions. For the application order of resolutions, Demuth et al. [9, 10], Hassam et al. [13], Markovic et al. [30, 31], and our current approach all rely on the chronological order of the metamodel changes. Kusel et al. [25] and Cabot et al. [6] apply the resolution on a random order in comparison to the metamodel changes' order. We further apply a partial optimization by first applying the deletes before the rest of resolutions. This allows to not apply resolutions to a constraint if it is going to be deleted anyway at the end.

Finally, in contrast to our work, none of the existing approaches addressed the issues of semantically alternative resolutions and the application scope of a resolution on OCL constraints.

To the best of our knowledge, we are the first to propose alternative resolution strategies while coping with the application scope of the resolutions based on the structure of the impacted OCL constraint and the impacted location.

Table 12: Comparison of the different approaches that aim at OCL co-evolution. Legend [✓: handled, ×: not handled, *Partially*: partially handled].

Approaches	OCL co-evolution	Impact analysis	Atomic changes	Complex changes	Automation degree	Supported resolutions	Order of application	Semantically alternative resolutions	Application scope
Demuth et al. [9, 10]	Online	Online	✓	×	Fully automatic	Unique	Ordered (chronological)	×	×
Hassam et al. [13]	Offline	Online	−Generalize property	−Pull property, −Push property, −Association to class, −Inheritance to composition	Semi-automatic	Unique	Ordered (chronological)	×	×
Kusel et al. [25]	Offline	Offline	✓	−Move property (through a single valued reference)	Semi-automatic	Unique	Random (diff-based)	×	×
Markovic et al. [30, 31]	Online	Online	−Rename elements	−Pull property, −Push property, −Move property, −Extract class, −Extract super class	Fully automatic	Unique	Ordered (chronological)	×	×
Cabot et al. [6]	Offline	Offline	−Delete elements	×	Fully automatic	Unique	Random	✓ *Partially*	×
Current work	Offline	Offline	✓	−Move property, −Pull property, −Push property, −Extract super class, −Flatten hierarchy, −Extract class, −Inline class	Semi-automatic	Multiple	Ordered (chronological, partially optimized)	✓	✓

7. Conclusion and Future Works

In this paper, we addressed the topic of metamodel and OCL constraints co-evolutions. We proposed a semi-automatic approach to co-evolve OCL constraints with alternative and appropriate resolution strategies. We identified two new *factors* that lead us to propose alternative resolutions to the user to chose from. We support the user in the application of the appropriate resolutions based on the metamodel changes, the location of the impact in the OCL constraints, and the context. This has the advantage to co-evolve OCL constraints w.r.t. the user intent and to avoid applying unintended resolutions. Our evaluation on six sized case studies of metamodels with their OCL constraints resulted in an average of 92% (syntactically) and 93% (semantically) correct co-evolution is reached in our evaluation.

Although we focused on the co-evolution of OCL constraints defined on top of metamodels, our approach can also handle the co-evolution of OCL constraints defined on top of object-oriented models in general. Thus in future work, we first aim to evaluate our approach on other applications of OCL constraints such as OCL queries, or OCL scripts that express model transformations and to run a user experiment. We further plan to investigate how the different OCL expression types can help in co-evolving the OCL constraints Furthermore, as we proposed to co-evolve non-impacted constraints for the two complex changes *Pull property* and *Extract super class*. We thus plan to further investigate the co-evolution of non-impacted OCL constraints and also the topic of OCL constraints refactoring. Finally, as we have proposed a fixed set of resolutions, it would be interesting to apply search-based techniques for the OCL co-evolution in order to explore all possible space of solutions.

Acknowledgments.

The research leading to these results has received funding from the ANR French Project MoNoGe under grant FUI - AAP no. 15.

References

[1] B. K. Aichernig and P. A. P. Salas. Test case generation by ocl mutation and constraint solving. In *Quality Software, 2005.(QSIC 2005). Fifth International Conference on*, pages 64–71. IEEE, 2005.

[2] S. Andova, M. G. van den Brand, L. J. Engelen, and T. Verhoeff. Mde basics with a dsl focus. In *Formal Methods for Model-Driven Engineering*, pages 21–57. Springer, 2012.

[3] X. Blanc, I. Mounier, A. Mougenot, and T. Mens. Detecting model inconsistency through operation-based model construction. In *Software Engineering, 2008. ICSE'08. ACM/IEEE 30th International Conference on*, pages 511–520. IEEE, 2008.

[4] A. Boronat and J. Meseguer. An algebraic semantics for mof. In *Fundamental Approaches to Software Engineering*, pages 377–391. Springer, 2008.

[5] F. Büttner, H. Bauerdick, and M. Gogolla. Towards transformation of integrity constraints and database states. In *Database and Expert Systems Applications, 2005. Proceedings. Sixteenth International Workshop on*, pages 823–828. IEEE, 2005.

[6] J. Cabot and J. Conesa. Automatic integrity constraint evolution due to model subtract operations. In *Conceptual Modeling for Advanced Application Domains*, pages 350–362. Springer, 2004.

[7] J. Cabot and E. Teniente. Transformation techniques for ocl constraints. *Science of Computer Programming*, 68(3):179–195, 2007.

[8] A. Cicchetti, D. D. Ruscio, R. Eramo, and A. Pierantonio. Automating co-evolution in model-driven engineering. In *Enterprise Distributed Object Computing Conference, 2008. EDOC'08. 12th International IEEE*, pages 222–231. IEEE, 2008.

[9] A. Demuth, R. E. Lopez-Herrejon, and A. Egyed. Automatically generating and adapting model constraints to support co-evolution of design models. In *Automated Software Engineering (ASE), 2012 Proceedings of the 27th IEEE/ACM International Conference on*, pages 302–305. IEEE, 2012.

[10] A. Demuth, R. E. Lopez-Herrejon, and A. Egyed. Supporting the co-evolution of metamodels and constraints through incremental constraint management. In *Model-Driven Engineering Languages and Systems*, pages 287–303. Springer, 2013.

[11] A. Demuth, M. Riedl-Ehrenleitner, R. E. Lopez-Herrejon, and A. Egyed. Co-evolution of metamodels and models through consistent change propagation. *Journal of Systems and Software*, 111:281–297, 2016.

[12] J. García, O. Diaz, and M. Azanza. Model transformation co-evolution: A semi-automatic approach. *Software Language Engineering*, 7745:144–163, 2013.

[13] K. Hassam, S. Sadou, V. L. Gloahec, and R. Fleurquin. Assistance system for ocl constraints adaptation during metamodel evolution. In *Software Maintenance and Reengineering (CSMR), 2011 15th European Conference on*, pages 151–160. IEEE, 2011.

[14] Z. Hemel, L. C. Kats, and E. Visser. Code generation by model transformation. In *ICMT*, pages 183–198. Springer, 2008.

[15] M. Herrmannsdoerfer, S. Benz, and E. Juergens. Cope-automating coupled evolution of metamodels and models. In *ECOOP 2009–Object-Oriented Programming*, pages 52–76. Springer, 2009.

[16] M. Herrmannsdoerfer, D. Ratiu, and G. Wachsmuth. Language evolution in practice: The history of GMF. In M. v. d. Brand, D. Gasevic, and J. Gray, editors, *Software Language Engineering*, pages 3–22. Jan. 2010.

[17] M. Herrmannsdoerfer, S. D. Vermolen, and G. Wachsmuth. An extensive catalog of operators for the coupled evolution of metamodels and models. In Malloy, Staab, and Brand, editors, *Software Language Engineering*, pages 163–182. Jan. 2011.

[18] J. Hutchinson, J. Whittle, M. Rouncefield, and S. Kristoffersen. Empirical assessment of mde in industry. In *Proceedings of the 33rd International Conference on Software Engineering*, pages 471–480. ACM, 2011.

[19] C. Jordi and G. Martin. Object constraint language (OCL): A definitive guide. In *12th SFM Bertinoro, Italy*, pages 58–90, 2012.

[20] W. Kessentini, H. Sahraoui, and M. Wimmer. Automated metamodel/-model co-evolution using a multi-objective optimization approach. In *12th ECMFA*, 2016.

[21] D. E. Khelladi, R. Bendraou, and M.-P. Gervais. Ad-room: a tool for automatic detection of refactorings in object-oriented models. In *Proceedings of the 38th International Conference on Software Engineering Companion*, pages 617–620. ACM, 2016.

[22] D. E. Khelladi, R. Hebig, R. Bendraou, J. Robin, and M.-P. Gervais. Detecting complex changes during metamodel evolution. In *International Conference on Advanced Information Systems Engineering*, pages 263–278. Springer, 2015.

[23] D. E. Khelladi, R. Hebig, R. Bendraou, J. Robin, and M.-P. Gervais. Detecting complex changes and refactorings during (meta) model evolution. *Information Systems*, 2016.

[24] D. E. Khelladi, R. Hebig, R. Bendraou, J. Robin, and M.-P. Gervais. Metamodel and constraints co-evolution: A semi automatic maintenance of ocl constraints. In *International Conference on Software Reuse*, pages 333–349. Springer, 2016.

[25] A. Kusel, J. Etzlstorfer, E. Kapsammer, W. Retschitzegger, J. Schoenboeck, W. Schwinger, and M. Wimmer. Systematic co-evolution of ocl expressions. In *11th APCCM 2015*, volume 27, page 30, 2015.

[26] A. Kusel, J. Etzlstorfer, E. Kapsammer, W. Retschitzegger, J. Schoenboeck, W. Schwinger, and M. Wimmer. A systematic taxonomy of metamodel evolution impacts on ocl expressions. ME@MODELS, 2015.

[27] P. Langer, M. Wimmer, P. Brosch, M. Herrmannsdörfer, M. Seidl, K. Wieland, and G. Kappel. A posteriori operation detection in evolving software models. *Journal of Systems and Software*, 86(2):551–566, 2013.

[28] G. Liebel, N. Marko, M. Tichy, A. Leitner, and J. Hansson. Assessing the state-of-practice of model-based engineering in the embedded systems domain. In *Model-Driven Engineering Languages and Systems*, pages 166–182. Springer, 2014.

[29] Y. Lin, J. Gray, and F. Jouault. Dsmdiff: a differentiation tool for domain-specific models. *European Journal of Information Systems*, 16(4):349–361, 2007.

[30] S. Marković and T. Baar. Refactoring ocl annotated uml class diagrams. In *Model Driven Engineering Languages and Systems*, pages 280–294. Springer, 2005.

[31] S. Marković and T. Baar. Refactoring ocl annotated uml class diagrams. *Software & Systems Modeling*, 7(1):25–47, 2008.

[32] G. Mezei, T. Levendovszky, and H. Charaf. An optimizing ocl compiler for metamodeling and model transformation environments. In *Software Engineering Techniques: Design for Quality*, pages 61–71. Springer, 2007.

[33] M. Morisio, M. Ezran, and C. Tully. Success and failure factors in software reuse. *Software Engineering, IEEE Transactions on*, 28(4):340–357, 2002.

[34] OMG. Object management group. `http://www.omg.org/spec/`, 2015.

[35] OMG. Object management group. meta object facility (mof). `http://www.omg.org/spec/MOF/`, 2015.

[36] OMG. Object management group. object constraints language (ocl). `http://www.omg.org/spec/OCL/`, 2015.

[37] OMG. Object management group. query / views / transformations (qvt). `http://www.omg.org/spec/QVT/`, 2015.

[38] OMG. Object management group. unified modeling language (uml). `http://www.omg.org/spec/UML/`, 2015.

[39] OMG. Reference metamodel for the express information modeling language. `http://www.omg.org/spec/EXPRESS/`, 2015.

[40] OMG. Requirements interchange format. `http://www.omg.org/spec/ReqIF/`, 2015.

[41] OMG. Structured metrics metamodel. `http://www.omg.org/spec/SMM/`, 2015.

[42] OMG. Uml profile for national information exchange model. `http://www.omg.org/spec/NIEM-UML/`, 2015.

[43] OMG. Unified profile for the department of defense architecture framework (dodaf) and the ministry of defence architecture framework (modaf). `http://www.omg.org/spec/UPDM/`, 2015.

[44] J. Schoenboeck, A. Kusel, J. Etzlstorfer, E. Kapsammer, W. Schwinger, M. Wimmer, and M. Wischenbart. Care: A constraint-based approach for re-establishing conformance-relationships. In *Proceedings of the Tenth Asia-Pacific Conference on Conceptual Modelling-Volume 154*, pages 19–28. Australian Computer Society, Inc., 2014.

[45] D. Shirtz, M. Kazakov, and Y. Shaham-Gafni. Adopting model driven development in a large financial organization. In *Model Driven Architecture-Foundations and Applications*, pages 172–183. Springer, 2007.

[46] D. Steinberg, F. Budinsky, E. Merks, and M. Paternostro. *EMF: eclipse modeling framework*. Pearson Education, 2008.

[47] J.-P. Tolvanen and S. Kelly. Metaedit+: defining and using integrated domain-specific modeling languages. In *Proceedings of the 24th ACM SIGPLAN conference companion on Object oriented programming systems languages and applications*, pages 819–820. ACM, 2009.

[48] A. Toulmé and I. Inc. Presentation of emf compare utility. In *Eclipse Modeling Symposium*, pages 1–8, 2006.

[49] G. Wachsmuth. Metamodel adaptation and model co-adaptation. In *ECOOP 2007–Object-Oriented Programming*, pages 600–624. Springer, 2007.

[50] C. Wohlin, P. Runeson, M. Höst, M. C. Ohlsson, B. Regnell, and A. Wesslén. *Experimentation in software engineering*. Springer Science & Business Media, 2012.

[51] Z. Xing and E. Stroulia. Umldiff: an algorithm for object-oriented design differencing. In *Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering*, pages 54–65. ACM, 2005.