Metamodel and Constraints Co-evolution: A Semi Automatic Maintenance of OCL Constraints

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

Metamodel and Constraints Co-evolution: A Semi Automatic Maintenance of OCL Constraints

Djamel Eddine Khelladi[1], Regina Hebig[2], Reda Bendraou[1], Jacques Robin[1], Marie-Pierre Gervais[1]

[1] Sorbonne Universités, UPMC Univ Paris 06, UMR 7606, F-75005, Paris, France.
[2] Chalmers and University of Technology Gothenburg, Sweden
{firstname.lastname}@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 investigate the 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 the evolution of the UML case study. The results confirm the need of alternative resolutions along with user decision to cope with real co-evolution scenarios. The results show that our approach reaches 80% of semantically correct co-evolution.

1 Introduction

In *Model-Driven Engineering*, metamodels are core components of a modeling language ecosystem [10]. They define the structural aspects of a business domain, i.e. the main concepts, their properties, and the relationships between them [4]. However, a metamodel alone is insufficient to capture all the relevant aspects and information of a domain specification [18]. To overcome this limitation, the Object Constraint Language (OCL) [21] is used to define constraints on top of the metamodel. For instance, the wide-spread Unified Modeling Language (UML) [23] in version 2.4.1 contains more than 750 OCL constraints expressing well-formedness rules to be enforced at the model instances level.

A challenge hereby arises when the metamodel is evolved causing the invalidation of some OCL constraints that need to be co-evolved (i.e. maintained to remain reusable [19]). For instance, the UML metamodel officially evolved 10 times in the past that led to manually adapting the OCL constraints.

Manual co-evolution is a tedious, time-consuming, and error-prone task, in particular when hundreds of OCL constraints exist. In such a context, it is crucial to support an automatic co-evolution.

Problem Statement. 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.

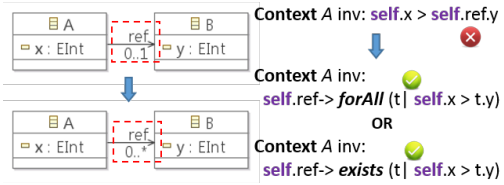


Fig. 1: Existence of multiple solutions

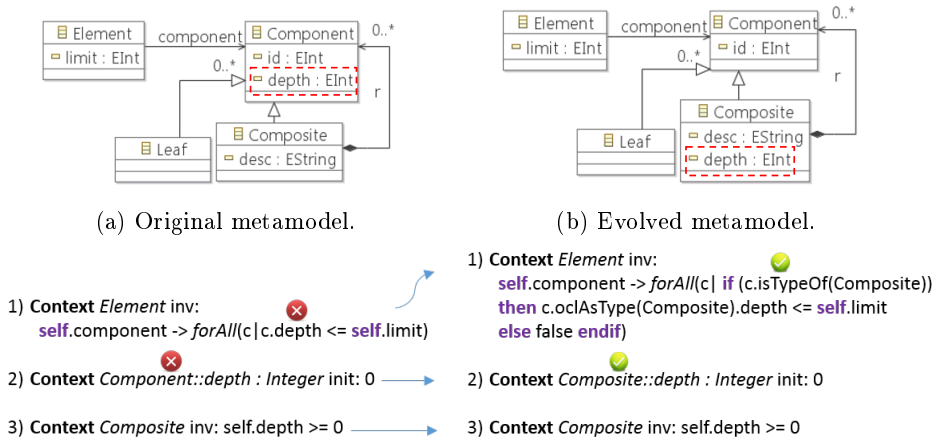
Multiple resolutions can be applied here as depicted in Figure 1, since multiple iterators with *different semantics* exist, e.g. forAll(), exists() etc. Proposing a unique resolution reduces the applicability of the co-evolution approach and limits its benefit. Final decision can only be specified by 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" [9]. 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 reference *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 [7, 5, 6, 16, 17, 13] propose a unique resolution per metamodel change. They neither consider the two above issues, nor interact with the user.

Contributions. We thus addressed these challenges by four contributions:

– First, we systematically investigated what are the *influencing factors* that lead to define alternative resolution strategies and when to apply them. Thus,



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

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

Fig. 2: An evolution example of a composite pattern.

we established 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 multiple resolutions per impacted part of an OCL constraint based on the influencing factors. 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.
- Third, we offer the user the option to choose the appropriate resolutions to be applied during co-evolution among the ones we propose. The user can also decide to not apply a resolution if it does not suit her needs. Involving the user greatly contributes to avoid applying unintended resolution strategies.
- Fourth, we evaluate our approach by comparing a set of manually co-evolved OCL constraints in practice against the same co-evolved ones with our tool. The evaluation confirms that alternative resolutions along with users final decision are required in practice for a correct co-evolution. The results on the UML Class Diagram case study show that our approach can handle a real case study, reaching 80% of semantically correct OCL co-evolution.

For a better understanding of the current approach, this paper first discusses the factors that influence the application of the resolution strategies in Section 2. 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 of Figure 2.

Factor 1. First of all, the metamodel change is fundamental to choose which resolution to apply, similarly as in model co-evolution (e.g. [15, 25]). 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**.

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**.

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 the *factor 1*, only. To identify the additional *factors 2* and *3*, we systematically studied the different uses of the metamodel elements in the OCL language and we analyzed why a resolution can be applied to some constraints and not to others.

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.

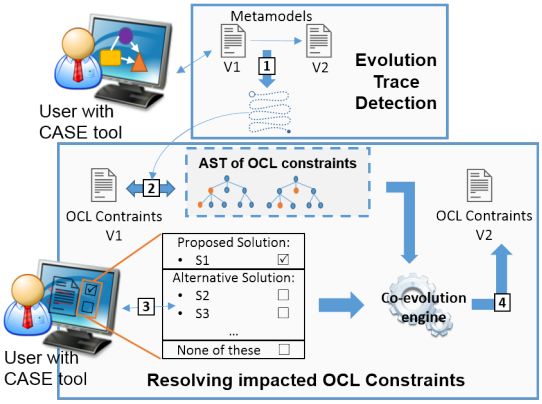


Fig. 3: Overall Approach.

Finally, we show how alternative resolutions are proposed to the user [3] and how they are automatically applied [4].

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 [9].

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 [24] and the MOF [20] standards. In the literature, over sixty complex changes are proposed [9]. 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 [3] metamodel showed that these seven changes are the most used ones and constitute 72% of the applied complex changes [8, 14]. In our case study 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 [12, 11], an extension of the Praxis tool [1]. It first records at run-time all atomic changes applied by users within a modeling tool. The sequence of recorded atomic changes then serves as input for the detection of complex changes. Our tool [12, 11] has been designed to detect all applied changes. This is confirmed in the evaluation results by always reaching a 100% recall (i.e. all complex changes are detected) and a precision (i.e. correct detection) of 91% and 100%. Our tool [12, 11] 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.

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 all the elements used in an OCL constraint. Thus, we first parse the OCL constraints to use the Abstract

[3] 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	$OCL_1, ...$	$ref_1, ref_2, ...$
...	...	...

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 use pre-order tree traversal algorithm through 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.

3.3 Obtaining the Influencing Factors

As discussed in section 3.1 the metamodel changes are given as input from our detection [12, 11]. 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 3.5.

3.4 Resolution Strategies

In this section we present our resolutions and the influencing factors under which they are applied. As an example, we present the resolution strategies associated with the metamodel change *"generalize property multiplicity (GPM) from a single value to multiple values"* that is applied 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.

`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 using the values of p is moved to the body of the *"forAll"*

Table 2: Resolutions proposed in our co-evolution approach.

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	container	#S8	3
	navigation path	n/a	#S9 #S10	
◇ Push property	context	container	#S11	3
	navigation path	container	#S12	
	navigation path	not via the subclasses	#S13	
◇ Extract class	context	container	#S8	2
	navigation path	n/a	#S9	
◇ Inline class	context	container	#S14	3
	navigation path	n/a	#S15 #S16	
◇ Flatten hierarchy	n/a	container	#S17	1

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.

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

`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 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.

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

Table 2 presents the metamodel changes that have an impact on OCL constraint which can be automatically resolved, and their associated resolutions while specifying the two new influencing factors. As shown in Table 2, for 8 metamodel changes we propose 17 resolutions. Note that we do not attempt to define all possible resolutions. Indeed, 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. Description and examples of all our resolutions can be found in our companion web page [4].

3.5 Proposing Resolution Strategies

In our approach we define and we implement a set of fixed resolutions that can be applied during co-evolution. When defining the resolutions we already specify under which influencing factors each resolution is applied (see section 3.4).

Figure 4 depicts our process of selecting the appropriate resolutions. It starts with all implemented resolutions and excludes a subset of resolutions based on

[4] https://pages.lip6.fr/Djamel.Khelladi/ICSR2016/

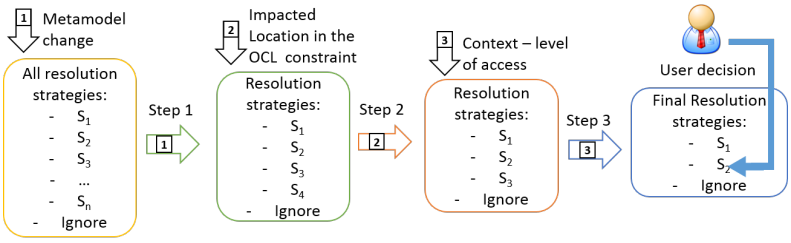


Fig. 4: Process of selecting the appropriate resolution strategies per impacted part of a constraint and per metamodel change.

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). 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.

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 × 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 in the evolution trace. It ensures consistency in the co-evolution since the resolutions are applied in the order of their associated metamodel changes. To remain flexible and to not introduce unintended solutions, our approach also proposes the possibility to *ignore* (in Figure 4) the proposed resolutions.

3.6 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. Note that 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 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

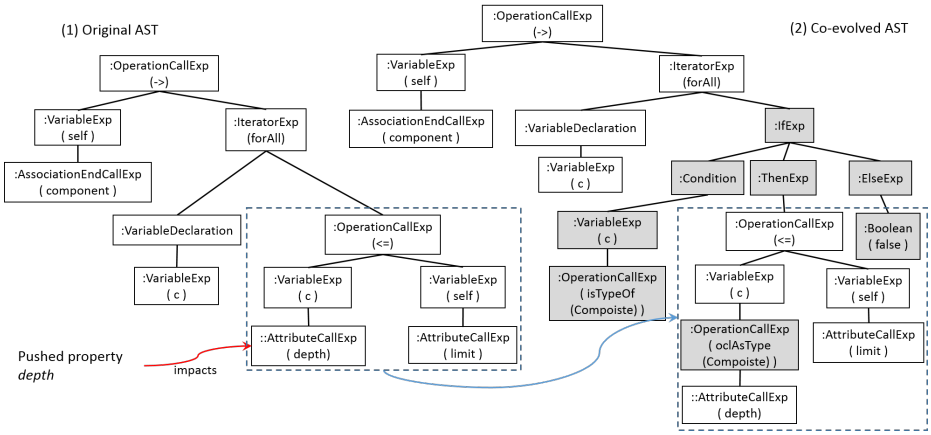


Fig. 5: AST of the original and co-evolved OCL constraint

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*.

4 Implementation

Our tool manipulates Ecore/EMF metamodels and OCL files for the constraints. After identifying the metamodel evolution trace with the detection tool [12, 11]. Our tool runs the impact analysis on the OCL constraints 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 and are packaged into an Eclipse plugin that is chained with the external plugins of Blanc et al. [1] and [12, 11].

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

This section presents a qualitative evaluation of our approach by comparing for the same set of constraints how they are manually co-evolved in practice against how they are co-evolved by our tool. This allows us to measure the precision of our approach. We first present our dataset. Then, we present the co-evolution

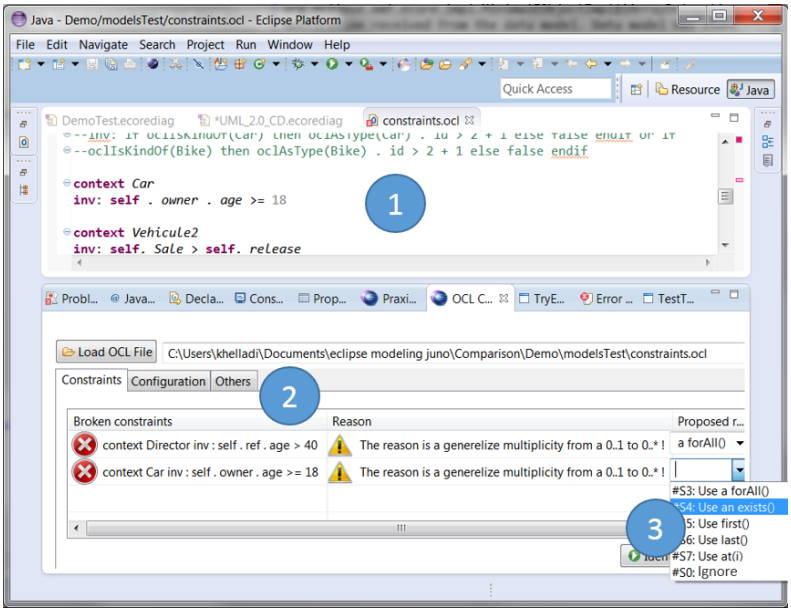


Fig. 6: Screenshot of the Eclipse plugin Tool.

as it occurred in practice. After that, the co-evolution results of our approach are illustrated. Finally, we compare our results against the ones in practice. Time performances of the co-evolution are measured as well. The goals of this evaluation are the following:

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

#G2: Show that our 17 resolutions are close to 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

We evaluate on a real evolution case study, namely: the UML Class Diagram (CD) metamodel from version 1.5 to 2.0 with their respective 73 and 110 OCL constraints. We collected the OCL constraints that are associated to the metamodel's versions 1.5 and 2.0. We put the constraints into a canonical form, e.g. by adding the keyword "self" to remove any ambiguity.

5.2 Co-Evolution Results as Occurred in Practice

As a first step of our evaluation, we studied how the OCL constraints are co-evolved in practice in response to the metamodel evolution. To this end, we first studied the evolution of the UML CD from 1.5 to 2.0 to determine the atomic and complex changes. In order to study how the OCL constraints are co-evolved in practice, we followed the next procedure:

Table 3: 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
◇ Constraints that are not impacted	19
◇ Constraints not impacted and present in both versions	7
◇ Constraints not impacted and deleted in the new version	12
◇ Constraints impacted in the first version (to be co-evolved)	54
◇ Constraints co-evolved by deletion in the new version	35
◇ Constraints co-evolved and present in the new version	19
◇ New constraints in the new version	84

- 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 as 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.

Our analysis' results of the co-evolution in practice are presented in Table 3.

5.3 Co-Evolution Results by our Approach

We first detected with our tool [12, 11] the evolution traces of the UML CD metamodel that is given as input to our co-evolution tool. In the experiment, the authors play the user role. 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 results of our applied co-evolution are presented in Table 4.

Several constraints are impacted by more than one metamodel change. Thus, more than one resolution is applied for several constraints. For instance, rename #S1 (see Table 2) is several times applied along with #S2, #S3, #S9 #S12, or #S13 on the same constraint. As mentioned previously, the resolutions are applied following the chronological order of the detected metamodel changes.

Performances. We ran our experiment on a PC VAIO with i7 1.80 GHz Processor and 8GB of RAM with Windows 7 as OS. 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.

Table 4: Co-evolution of OCL constraints by our approach

Co-evolution of OCL constraints by our approach	UML CD 1.5 to 2.0
◇ Constraints that are not impacted	19
◇ Constraints impacted in the first version (to be co-evolved)	54
◇ Constraints co-evolved by deletion	28
◇ Constraints co-evolved by other resolution strategies	26

5.4 Comparison of the Results: "our Approach" VS "in Practice"

Following our procedure of section 3.2 we were able to identify all the 54 impacted constraints.

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

Undeleted constraints. In our approach we co-evolved 26 constraints with various resolutions. 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 are 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, making 15 semantically correct co-evolved constraints with our tool.

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

In practice the context of constraint (1) was changed to the subclass `Classifier` instead of or after applying the rename. Thus, 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 syntactically and semantically correct co-evolution are respectively 72% and 80% [5].

Maintained constraints. In our approach, 7 impacted constraints are co-evolved whereas they are deleted in practice $\boxed{A}$. We applied 8 times the rename strategy #S1 for six of the constraints and 1 time the strategy #S16 of an inline

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

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

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. Furthermore, it is also surprising to find deletion of 12 non-impacted constraints $\boxed{B}$. For example, constraint (5) expressing that an interface can only contain operations is deleted.

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

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.

5.5 Discussion and Threats to Validity

The preliminary evaluation shows that our approach is able to cope with real co-evolution of OCL constraints. In the following we discuss the observed results.

1) First of all, as mentioned previously multiple resolutions are used in particular for the metamodel changes *push property* (#S12, #S13) and *inline class* (#S15, #S16). These results emphasize and confirm the necessity to propose alternative resolutions in order to cope with realistic scenarios of OCL constraints' co-evolution. Otherwise, the rates of automatic co-evolution would be lower than the ones in this paper, with a higher risk of introducing inappropriate solutions.

2) Furthermore, the first 7 cases of maintained constraints $\boxed{A}$ underline the need to also propose the delete strategy #S2 whenever a constraint is impacted, and not always try to maintain the constraint. Moreover, the second 12 cases of deleted constraints $\boxed{B}$ 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.

3) Finally, the cases of the semantically not matching constraints in UML CD (e.g. constraint (3)) 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 refactor it w.r.t. her intent.

We now discuss threats to validity (internal, external, and conclusion) after Wohlin et al. [26] w.r.t. our three evaluation goals #G1, #G2, and #G3.

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 17 resolutions. However, in our evaluation the 7 complex changes we considered as well as the 17 resolutions were sufficient in our case studies. Therefore, this threat is acceptable here.

External Validity. We evaluated our tool on a case study of metamodels and its OCL constraints. However, it is difficult to generalize our obtained results for other metamodels and OCL constraints. Nonetheless, the UML CD case study provided a representative and a complex evolution trace that had a significant impact on the OCL constraints.

Conclusion Validity. Our evaluation gives promising results demonstrating that alternative resolutions are used in real cases. The results also indicate that our 17 resolutions are semantically close to the user need during the co-evolution. Thus, our evaluation results meet our goals #G1 and #G2. However, we cannot estimate the quality of the resolutions only based on our UML CD case study. Third goal #G3 is also met since our tool covers all metamodel changes that occurred in the UML CD evolution. 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 contrast to models and transformations co-evolution where many works exist (e.g. [15, 25]), co-evolution of OCL constraints has received little attention so far. Demuth et al. [5, 6] proposed an approach for OCL co-evolution based on templates. They provided 11 templates that define a fixed structure for OCL constraints. Thus, the co-evolution in this case is a re-instantiation of the templates to update the 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.

Hassam et al. [7] proposed to co-evolve OCL constraints using QVT [22] a transformation language. Similarly, Markovic et al. [16, 17] proposed to refactor, based on QVT, OCL constraints annotated on UML class diagrams when these last evolve. Kusel et al. [13] discussed the impact of metamodel evolution on OCL expressions and proposed to resolve impacted expressions. However, they do not consider an OCL constraint as a whole. In particular, the context is ignored whereas it can be the impacted part that requires a resolution.

Cabot et al. [3] focused on the metamodel change delete element. In particular, they aimed at 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). Buttner et

al. [2] discussed the impact of changing the multiplicity of a property on OCL constraints. In our approach we also address this issue in our resolution (#S3-7).

All existing approaches [5–7, 16, 17, 3, 2, 13] consider only the metamodel change as a factor to propose a resolution. Thus, they define for each metamodel change only a unique resolution. In contrast, we identified two additional factors that lead to propose alternative resolutions.

To the best of our knowledge, we are the first to consider these issues and to show that multiple resolutions are needed in practice. We therefore are the first to propose alternative resolution strategies while considering the metamodel change, the impact location in an OCL constraint, and the context.

7 Conclusion and Future Work

In this paper, we addressed the topic of metamodel and OCL constraints co-evolutions and proposed a semi-automatic approach with alternative resolution strategies. We identified two new *factors* that lead us to propose alternative resolutions to the user to chose from. This has the advantage to co-evolve OCL constraints w.r.t. the user intent and to avoid applying unintended resolutions. We evaluated our approach on a big-medium sized case study: the UML CD metamodel with its OCL constraints. The results show that our approach is suitable to handle complex co-evolution scenarios of metamodels and OCL constraints. It reached 72% of 80% of syntactically and semantically correct co-evolution.

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. We further plan to explore the possibility to allow the user to import external resolutions in particular to be used along with the *ignore* option.

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

References

1. X. Blanc, I. Mounier, A. Mougenot, and T. Mens. Detecting model inconsistency through operation-based model construction. In *ACM/IEEE 30th ICSE'08*, pages 511–520, 2008.
2. F. Buttner, H. Bauerdick, and M. Gogolla. Towards transformation of integrity constraints and database states. In *DEXA*, pages 823–828, 2005.
3. 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.
4. J. Cabot and M. Gogolla. Object constraint language (OCL): A definitive guide. In *12th SFM Bertinoro, Italy*, pages 58–90, 2012.
5. A. Demuth, R. Lopez-Herrejon, and A. Egyed. Automatically generating and adapting model constraints to support co-evolution of design models. In *27th IEEE/ACM ASE*, pages 302–305, 2012.

6. A. Demuth, R. E. Lopez-Herrejon, and A. Egyed. Supporting the co-evolution of metamodels and constraints through incremental constraint management. In *MODELS*, pages 287–303. Jan. 2013.
7. K. Hassam, S. Sadou, V. L. Gloahec, and R. Fleurquin. Assistance system for ocl constraints adaptation during metamodel evolution. In *CSMR*, pages 151–160. IEEE, 2011.
8. M. Herrmannsdoerfer, D. Ratiu, and G. Wachsmuth. Language evolution in practice: The history of GMF. In *Software Language Engineering*, pages 3–22. Springer, Jan. 2010.
9. M. Herrmannsdoerfer, S. D. Vermolen, and G. Wachsmuth. An extensive catalog of operators for the coupled evolution of metamodels and models. In B. Malloy, S. Staab, and M. v. d. Brand, editors, *Software Language Engineering*, pages 163–182. Jan. 2011.
10. 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.
11. D. E. Khelladi, R. Bendraou, and M.-P. Gervais. Ad-room : a tool for automatic detection of refactorings in object-oriented models. In *The 38th ICSE*, 2016.
12. D. E. Khelladi, R. Hebig, R. Bendraou, J. Robin, and M.-P. Gervais. Detecting complex changes during metamodel evolution. In *The 27th CAISE*, pages 264–278. Springer, 2015.
13. 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.
14. P. Langer, M. Wimmer, P. Brosch, M. Herrmannsdorfer, 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.
15. F. Mantz, G. Taentzer, Y. Lamo, and U. Wolter. Co-evolving meta-models and their instance models: A formal approach based on graph transformation. *Science of Computer Programming*, 104:2–43, 2015.
16. S. Markovic and T. Baar. Refactoring OCL annotated UML class diagrams. In *MODELS*, pages 280–294. 2005.
17. S. Markovic and T. Baar. Refactoring OCL annotated UML class diagrams. *Softw Syst Model*, 7(1):25–47, Feb. 2008.
18. 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, 2006.
19. M. Morisio, M. Ezran, and C. Tully. Success and failure factors in software reuse. *Software Engineering, IEEE Transactions on*, 28(4):340–357, 2002.
20. OMG. Meta object facility (mof). `www.omg.org/spec/MOF/`, 2011.
21. OMG. Object constraints language (ocl). `www.omg.org/spec/OCL/`, 2015.
22. OMG. Query/views/transformations (qvt). `www.omg.org/spec/QVT/`, 2015.
23. OMG. Unified modeling language (uml). `www.omg.org/spec/UML/`, 2015.
24. D. Steinberg, F. Budinsky, E. Merks, and M. Paternostro. *EMF: eclipse modeling framework*. Pearson Education, 2008.
25. G. Wachsmuth. Metamodel adaptation and model co-adaptation. In E. Ernst, editor, *ECOOP*, pages 600–624. Jan. 2007.
26. C. Wohlin, P. Runeson, M. Höst, M. C. Ohlsson, B. Regnell, and A. Wesslén. *Experimentation in software engineering*. Springer Science & Business Media, 2012.