Identifying Metamodel Inaccurate Structures During Metamodel/Constraint Co-Evolution

Elyes Cherfa, Soraya Mesli-Kesraoui, Chouki Tibermacine, Régis Fleurquin, Salah Sadou

Identifying Metamodel Inaccurate Structures During Metamodel/Constraint Co-Evolution

Elyes Cherfa
Segula Engineering France
IRISA – University South Britanny
Vannes, France
elyes.cherfa@univ-ubs.fr

Soraya Mesli-Kesraoui
Segula Engineering France
Lanester, France
soraya.kesraoui@segula.fr

Chouki Tibermacine
LIRMM, Univ Montpellier, CNRS
Montpellier, France
chouki.tibermacine@lirmm.fr

Régis Fleurquin
IRISA – University South Britanny
Vannes, France
regis.fleurquin@univ-ubs.fr

Salah Sadou
IRISA – University South Britanny
Vannes, France
salah.sadou@univ-ubs.fr

***Abstract*—Metamodels are subject to evolution over their lifetime. UML metamodel for instance evolved through different versions, ranging from 0.8 to 2.5 minors. These metamodels are sometimes accompanied with constraints defined using OCL (Object Constraint Language). Many works in the literature developed methods for managing and assisting the co-evolution of metamodels and their constraints. These methods enable a developer to update, in an automated (or semi-automated) way, the constraints associated to a metamodel starting from the deltas identified between versions of this metamodel. In this work we complement this assistance by notifying the developer with potential inaccurate structures in the metamodel that may be introduced during evolution. We introduce in this paper an original evolution assistance method which focuses rather on the problem (notifying metamodel inaccurate structures) than on the solution (generating OCL constraints using patterns of them). The ultimate goal of this assistance is not only to enable the developer to complete existing/updated constraints with new ones, but also to accompany her/him to further check existing constraints and to test whether they still hold. A case study is presented to show the relevance of the method.**

***Index Terms*—Model-driven Engineering, Metamodelling, OCL, Co-Evolution**

I. Introduction: Context & Problem Statement

Metamodels define the syntax and part of the semantics of modeling languages. In the Model-Driven Engineering field, these are defined using languages like MOF [11]. Their specification includes sometimes constraints defined using the OMG's Object Constraint Language (OCL [34]). These constraints enable to make the semantics of the metamodel more precise in order to be compliant with the business domain.

For many reasons, metamodels are subject to evolution. Sometimes, this evolution is motivated by a syntax change or the introduction of new modeling elements (eg. the evolution of UML metamodel from version 1.5 to 2.0). It can also be motivated by refactoring the metamodel in order to improve and simplify its structure (eg. from UML 2.4 to UML 2.5). However, to keep the metamodel consistent, it is important to also evolve the constraints associated with it accordingly. This problem is known under the name Metamodel/Constraint coevolution. Many works in the literature proposed methods for coevolving metamodels and their constraints [16], [25], [26], [30]. These works provided efficient tools for automating as much as possible the identification of impacted constraints when a change occurs in the metamodel. Sometimes they even offer a solution for rewriting the constraints impacted by the change [15], [19]. Thus, they offer a precious assistance to developers when changing their metamodels, by freeing them from managing the changes on OCL constraints.

However, these works concern only the constraints that have been already defined. It is true that a modification on the metamodel can certainly induce a modification on the existing constraints, but sometimes it can generate the need to add new constraints. Indeed, if a portion of a metamodel is enough precise without constraints, its modification can be less precise and thus requires constraints. Existing works do not propose any means to identify potential new OCL constraints when the metamodel evolves.

In this work, we tackle the problem of new constraints needed for a consistent Metamodel/Constraint coevolution. Thus, we propose a new method for assisting this coevolution which complements existing solutions. This method considers a new way of providing assistance to developers, by focusing on the problem domain (what are the structures in the metamodel that potentially cause problems and which need OCL constraints?) instead of exploring the solution domain (generating the missing constraints). Concretely we reuse two existing works to provide such assistance: one concerns the needed operators to evolve a metamodel [20] and the other concerns the definition of Metamodel Inaccurate Structures (MIS) [3]. We believe that this way of assistance is informative, because it highlights the potential bad structures in the new version of a metamodel. This implies either writing new OCL constraints or reconsidering some existing ones.

In the rest of the paper, we present in Section II, an example

illustrating the problems tackled by our work. We detail in Section III the approach we have proposed to solve this problem. Section IV presents the application of our approach on the case study: StateMachine metamodel and its evolution from version 1.5 to 2.0. The results of this evaluation are presented and discussed in Section V. Section VI discusses the different threats to the validity of the results of the of this case study. We end this paper with a conclusion (Section VIII) after presenting the related work (Section VII).

II. Problem Illustration

To illustrate in a simple way the problem addressed in our work, we take a small example from the UML StateMachine metamodel and its evolution from version 1.5 to version 2.0 (Fig. 1). This evolution was made by applying the following evolution operations:

- PullUp property *submachineState* from *SubmachineState* to *CompositeState*
- PullUp property *submachineState* from *CompositeState* to *State*

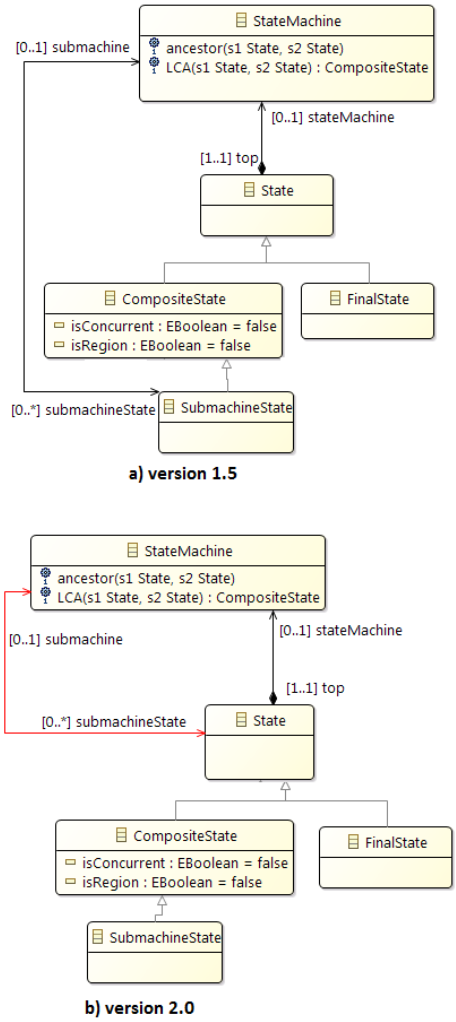

Fig. 1. Evolution of the StateMachine (excerpt)

During this evolution, the reference *submachineState* initially typed by *SubmachineState* class has been Pulled up to the *CompositeState* class and then typed by *State* class. According to the adaptation operations existing in the literature, the application of this evolution operation would require a coevolution of the constraints relating to the *submachineState* reference. In fact, all constraints of type:

OCLExpression. submachineState [Rest], such as *OCLExpression* is of type *StateMachine*, will be coevolved after applying the first PullUp operation in:

OCLExpression. submachineState ->select(v | v. oclIsTypeOf(SubmachineState)) -> forAll(s | s[Rest])

and after applying the second PullUp operation in:

OCLExpression. submachineState -> select(v | v.oclIsTypeOf(CompositeState)) -> select(v | v.oclIsTypeOf(SubmachineState)) -> forAll(s | s[Rest])

But, the PullUp operation also generated a new MIS which may require a new OCL constraint. Indeed, all the sub-classes of the *State* class, which now contains the reference *submachine*, inherit this reference. It may be possible that these sub-classes do not need to inherit the reference and thus require a new constraint to reduce the multiplicity of this inherited reference. Indeed, in the StateMachine 2.0 version, the following constraint has been added. This constraint reduces the multiplicity of the inherited reference *submachine* in the *FinalState* sub-class.

context FinalState
inv inv: self.submachine->isEmpty()

In the same vein, we studied in this paper the different metamodel inaccurate structures that may require the addition of new constraints or the removal of the existing one(s), after the application of an evolution operation.

III. Background and Proposed Approach

Our approach relies on OCL constraint coevolution operators and Metamodel Inaccurate Structures. Thus, for a better understanding of our approach, we need to explain the main elements of these two works from the literature.

A. Metamodel coevolution operators

During the coevolution phase, a metamodel undergoes several changes. Examples of needs leading to changes on the metamodel are adding/removing concepts or refactoring it in order to simplify its comprehension and make it easy to maintain. In the literature, the coevolution problem concerning metamodels has been addressed by many studies aiming at identifying the changes that are usually applied. As a result, a set of 16 evolution operators was introduced by Wachsmuth [30] for coevolving metamodels and models.

Besides this, Marković and Barr [25] proposed 15 rules to coevolve class diagrams and OCL constraints. These have been completed by [15] with 7 operations for coevolving metamodels and OCL constraints. Table I taken from [20] summarises all identified EMOF metamodel evolution operators which help in solving coevolution problems.

TABLE I
OVERVIEW OF THE OPERATORS FOR EMOF METAMODEL EVOLUTION

Type	Operator Name
Add / Remove Element	Add Class
	Remove Class
	Add Package
	Remove Package
	Add DataType, PrimitiveType, Enumeration
	Remove DataType, PrimitiveType, Enumeration
	Add EnumerationLiteral
	Remove EnumerationLiteral
	Add Property
	Remove Property
	Add Association
	Remove Association
	Add Operation (Class / DataType)
	Remove Operation (Class / DataType)
	Introduce Generalization
	Remove Generalization
Property Manipulation	Move Property
	Push Simple Property
	Push Complex Property
	Pull Simple Property
	Pull Complex Property
	Restrict (Unidirectional) Property
	Generalise (Unidirectional) Property
Refactoring Pattern	Extract Class
	Inline Class
	Extract Superclass
	Flatten Hierarchy
	Association to Class
	Generalization to Composition
	Introduce Composite Pattern

The evolution operators are grouped into four categories:

1) Those for adding or removing new modeling elements like: class, association, attribute, operation, generalization, package, etc;
2) Those that help in the manipulation of the properties such as: Move property, PullUp property, etc;
3) The third category groups the operators that manipulate hierarchies (Inheritance relations);
4) The last group represents all complex operators that were proposed to refactor structures based on patterns.

OCL constraints are the main artifacts that are affected by the metamodel changes. For example, removing an attribute needs to be followed by the removal of the OCL constraints that target this attribute. Following this rule, previous research works have identified a set of OCL coevolution operators that are triggered by one or more metamodel coevolution operators. These approaches are efficient when it comes to modification and removal operators. However, when the changes concern a part of the metamodel which was not associated with constraints, these operators give no indication. This is normal since their goal is to identify the constraints that need to be changed after a change in the metamodel. But sometimes, a change in a metamodel may raise the need for new constraints to be more precise, as shown in the example of the previous section.

B. Metamodel Inaccurate Structures

In [3], the authors have analyzed a dataset containing 10 metamodels and more than 800 OCL constraints. The analysis was done by studying all the metamodel structures that are targeted by at least one constraint. These structures have been grouped into sets, each of which can be defined by a generic structure. The latter is named Metamodel Inaccurate Structure (MIS). Thus, the presence of a MIS in a metamodel suggests the presence of weaknesses and therefore the need for OCL constraints to make the relevant part of the metamodel more precise. One of these weaknesses is inheritance. Inheritance (Generalization) allows to generalize common properties from many classes and factorize them into one class which will be inherited from all the other classes which need one or all of these properties. The problem that may occur is that in order to generalize a property that can be found in many classes, the multiplicity and the set of possible values of this property need to be relaxed to contain all the possible multiplicities or values that the property can take in the sub-classes. As a consequence, some classes will inherit a property with a wider multiplicity or a wider range of values, which needs to be precised (restricted) by adding a new OCL constraint. The authors have pointed out 10 MIS as illustrated in Table II.

TABLE II
OVERVIEW OF THE MIS [3]

MIS
Attribute Value Restriction
Enumeration Literals Restriction
Type Relation
Enumeration Type Relation
Inherited Attribute Value Restriction
Inherited Attribute Multiplicity Restriction
Inherited Association Multiplicity Restriction
Inherited Operation Value Restriction
Cycles Restriction
Different Paths Restriction

Figure 2, taken from the StateMachine metamodel, illustrates the *Enumeration Type Relation MIS*. This MIS occurs when a class contains an Enumeration typed attribute, and one or more incoming associations. Often, constraints are specified to precise the attribute literals value that the association must have (i.e. the type of the association is restricted by the enumeration). Here, the *Pseudostate* (Fig. 2) instances that are linked to the *ConnectionPointReference* instances through the association *entry* must be restricted to the literal *entryPoint*. This is also the case for the association *exit* with the literal *exitPoint*. A MIS suggests that a structure is possibly inaccurate. So, the metamodel designer (domain expert) needs to analyze each MIS occurrence in order to determine whether the structure is really not precise and hence needs to be completed with OCL constraints.

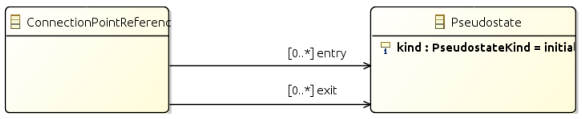

Fig. 2. Enumeration Type Relation from State Machine

C. Proposed Approach

Initially, as illustrated in figure 3, we have identified 30 operators (A) that evolve metamodel from (B) version to a new version (C). To guarantee that the existing OCL constraints (E) are conforming to the new metamodel version, a set of constraints coevolution operators (D) is applied to these constraints to coevolve them to be conforming to the new metamodel version (F). Since the set of constraints (F) is incomplete and hence does not cover all the concepts of the new metamodel version, a set of MIS that can arise from the evolution operators (H) is automatically searched (G). This set of MIS instances is then analysed by metamodel designer (I) to remove all the instances that do not require OCL constraints. After this sorting, the resulted MIS instances set (J) encompasses only instances that require OCL constraints. To this end, we have developped a tool that proposes OCL (K) constraints for each MIS instance. The proposed constraints are instances of OCL patterns that are frequently used to precise MIS. After that, the domain expert will validate the set of constraints (L) relying on its domain knowledge. Consequently, the resulted set of OCL constraints (M) includes all the new OCL constraints that are applied to precise new metamodel elements resulting from the metamodel evolution. This set of constraints (M) union the set of constraints resulted from coevolving the OCL constraints (F) from the old metamodel using coevolution operators give place to a complete set of OCL constraints (N) that are conforming to the new metamodel version.

To take benefit from the knowledge acquired from the MIS and to complete the OCL coevolution, we performed an analysis over each metamodel coevolution operator. This analysis aims at identifying all the MIS that may arise following the application of a metamodel evolution operator. To do so, for each evolution operator, we have taken a set of structures and applied it, then searched for MIS instances to know which MIS may result from which metamodel evolution operator.

As a result, we obtained a set of MIS for each evolution operator. This results show which MIS can be generated after the application of an evolution operator. Table III summarizes our findings.

To be more precise, we have divided some operators such as pullUp property into pullUp association, and pullUp attribute to precise the resulted MIS. The table contains three types of metamodel evolution operators (I). Each type encompasses a set of operators (II) that we have taken from the literature. For each operator, we state if its application may cause a change that impacts OCL constraints, which require the coevolution of at least one constraint (III). For example, adding a class does not affect any OCL constraint, but removing an attribute requires removing/refactoring all the constraints that were referenced. After that, for each operator, we state if it may engender one or many MIS or not, and if it is the case, we list all the MIS that may be triggered from an evolution operator (IV). Finally, we give some conditions predominantly related to the metamodel structure, making the rise of a MIS possible (V). For example, the application of the operator "add association" may trigger the MIS "Cycle" only if the association is reflexive.

Figure 4 illustrates the application of *Add Association* operator in a metamodel structure. From this evolution results the *Inherited Association Multiplicity Restriction* MIS. This MIS occurs when a super-class contains an association with a multiplicity. Often, an OCL constraint is added to one of the sub-classes to restrict the association multiplicity. The operator *Add Association* does not affect the correctness of existing constraints. However, the added association may need constraints. This shows that the existing OCL coevolution operators are not sufficient to make the evolved metamodel precise. Consequently, relying on coevolution operators to coevolve existing constraints and search MIS instances raised from the metamodel evolution is an efficient combo to cover all the metamodel structures with constraints. The strength of our approach lies in its capacity to identify the need for new constraints that did not appear in the old metamodel version.

IV. Case Study

To evaluate the performance of our method in OCL coevolution assistance, we conducted a case study. The purpose of this study is to provide answers to the following research questions.

A. Research Questions

- **RQ1. What is the performance of our method in the coevolution of OCL constraints?**
  This question assesses the ability of our approach to correctly coevolve the OCL constraints during the evolution of their meta-model.
- **RQ2. What is the performance of our method in the notification of potential new constraints related to MIS?**
  This question aims to study the ability of our method to notify structures which will potentially require new constraints or the removal of already existing constraints and thus complete the coevolution of OCL constraints during the evolution of a metamodel. The results of this question will highlight the originality of our method compared to methods from the state of the art.

B. Selected case study

We chose to evaluate our method on the State machine metamodel evolved from version 1.5 (Fig. 5) to version 2.0 (Fig. 6). This choice is guided by: 1) accessibility of the metamodel and its OCL constraints with a complete documentation; 2) the number of significant evolution operations (73 operations that were required to evolve the metamodel from

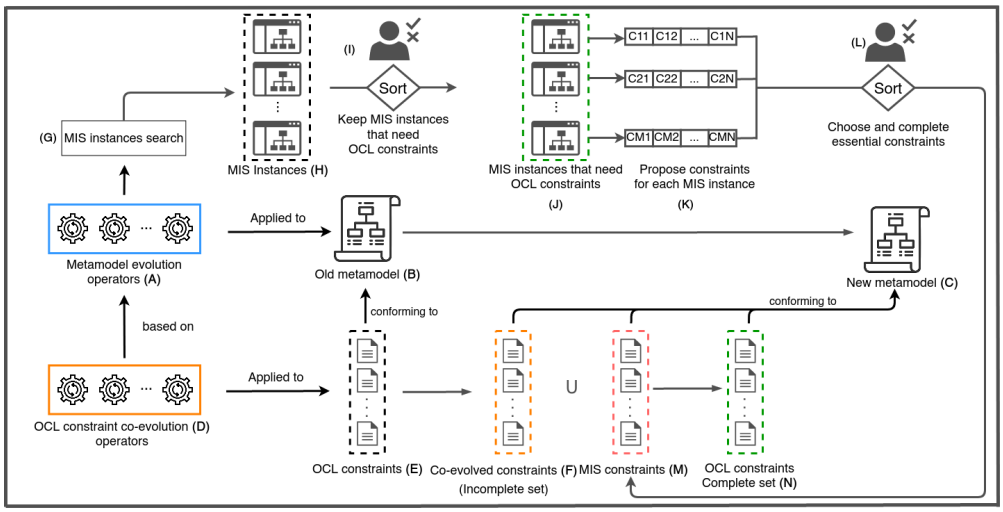

Fig. 3. Proposed approach to coevolve OCL constraints

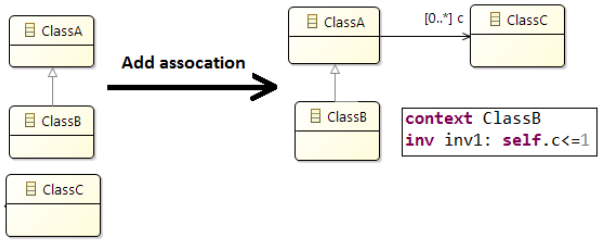

Fig. 4. Inherited Association Multiplicity Rest. resulted from Add Association operator

1.5 to 2.0), this covers many evolution operators (19), which allows us to validate the co-evolution of constraints such as existing approaches from the literature. Also, it enables the identification of new constraints that are needed to refine the new metamodel version; 3) the metamodel is widely used in practice which proves its good construction; 4) maneuverability: the metamodel is concise compared to the whole UML metamodel. In the following, we present the StateMachine metamodel based on its 2.0 version (Fig. 6).

State machines offer a range of concepts to model the discrete behavior of a system as a graph of vertices and transitions. The vertices are of different types: states, pseudostates, and connection point references (Fig. 6). The states describe a situation in which a condition is verified [10]. States can be simple, composite, sub-machines, or final states. Unlike a simple state, a composite state can contain other states and transitions often clustered in regions. A composite state, containing at least one region, may contain several regions. In this case, the state is called orthogonal because the regions can run in parallel. A final state is the last state to go through when running a state machine. Pseudostates are transient states (Execution never stops in these states). The state machine diagram offers 10 pseudostates (entryPoint, exitPoint, initial, deepHistory, shallowHistory, join, fork, junction, terminate, or choice). On the other hand, the connection point characterizes the entry and the exit points of a sub-machine.

A transition is an arc from a source vertex to a target vertex. A transition is fired at the reception of a given event and/or if its related guard is true. On the version 2.0 [10], three types of transitions were defined: internal, local, and external. An internal transition is a transition that connects a state to itself and it does not cause a state change during its execution. A local transition is a transition that does not come out of a composite state, it is always performed in the composite state. On the other hand, the execution of an external transition leads to the exit of the composite state from which it is outgoing.

C. Data

We studied the evolution of state machine diagrams from version 1.5 to 2.0. For both versions, we used the official state machine specification provided in [8]. Then, for both versions, we built an ECORE metamodel and a ".ocl" file containing the OCL constraints associated with this metamodel.

The metamodel 1.5 (Table IV) is composed of 47 modeling elements: 14 classes, 1 enumeration, 5 attributes, 2 operations, and 25 associations. On this metamodel, 30 OCL constraints were specified [9]. Of these 30 constraints, we were able to use only 25 constraints. The other 5 constraints have been removed either because they do not parse (they contain errors) or they relate to modeling elements contained in other packages of the UML metamodel that were not taken into account in this case study.

On the other hand, the metamodel 2.0 (Table IV) contains 69 modeling elements and 49 OCL constraints [10]. Only 38 out of 49 constraints were used in this study. 11 OCL constraints were excluded for the same reasons mentioned above.

Once the OCL metamodels and constraints were collected, we identified the set of evolution operations that make Statemachine metamodel 1.5 evolving to 2.0. We manually compared the two versions of the metamodel and extracted all the evolution operations. We have identified 73 evolution

TABLE III
MAPPING BETWEEN METAMODEL COEVOLUTION OPERATORS AND MIS

Type (I)	Operator Name (II)	OCL coevolution (III)	May engender MIS (IV)	Condition (V)
Add / Remove Element	Add Class	no	no	/
	Remove Class	yes	no	/
	Add Package	no	no	/
	Remove Package	no	no	/
	Add DataType , Primitive Type, Enumeration	no	no	/
	Remove DataType, Primitive Type, Enumeration	yes	no	/
	Add EnumerationLiteral	no	Enumeration Literal Restriction	If the enumeration wherein the literal is added will contain at least two literals
			Enumeration Type Relation	If the class containing an Enumeration attribute contains an incoming association
	Remove EnumerationLiteral	yes	no	/
	Add Attribute	no	Attribute Value Restriction	/
			Inherited Optional Attribute	If the attribute is added with the multiplicity [0..1] to a class containing subclasses
			Inherited Attribute Value Restriction	If the attribute is added to a class containing subclasses
	Remove Attribute	yes	no	
	Add Association	no	Type Relation	If the source or the target class or both containt subclasses
			Association Multiplicity Restriction	If the association multiplicity upper bound is higher than the lower bound and the class containing the association is specialized
			Cycle	If the association is reflexive
			Different Paths	If there is another association with the same source and target class
	Remove Association	yes	no	/
	Add Operation (Class / DataType)	no	Inherited Operation Value Restriction	If the class containing the operation is specialized
			Cycle	If the operation returns elements from same type as its containing class
			Different Paths	if the operation returns class instances as output, and if the class containing the operation have an association targetting the class
	Remove Operation (Class / DataType)	yes	no	/
	Introduce Generalization	yes	Inherited Optional Attribute	If the superclass contains an optional attribute
			Inherited Attribute Value Restriction	If the superclass contains an attribute
			Enumeration Literals Restriction	If the superclass contains an enumeration attribute
			Inherited Operation Value Restriction	If the superclass contains an operation
			Type Relation	If the superclass contains an association
			Association Multiplicity Restriction	If the superclass containts an association
			Cycle	If the subclass contains an association that targets its superclass
			Different Paths	
	Remove Generalization	yes	Remove Cycle	/
			Remove Different Path	/
Property Manipulation	Move Property	yes	Inherited Optional Attribute	If the class in which the attribute is moved is specialized
			Inherited Attribute Value Restriction	If the class in which the attribute is moved is specialized
			Type Relation	If the class in which the attribute is moved is specialized
			Inherited Association Multiplicity Restriction	If the class in which the attribute is moved is specialized
			Attribute Value Restriction	
			Cycle	If the property is a reflexive association
			Different Paths	If the class in which the association is moved contains another association with the same targetted class.
	PushDown attribute/operation	yes	no	
	Push Down association	yes	Remove Type Relation MIS on the other subclasses	/
			Remove Inherited Association Multiplicity Restriction on the other subclasses	/
			Cycle on the subclass	If the association is reflexive
	PullUp Attribute/Operation	yes	Attribute value restriction	
			Inherited Optional Attribute	/
			Inherited Attribute Value Restriction	/
			Inherited Operation Value Restriction	/
			Type Relation	If it is an enumeration attribute
	PullUp Association	yes	Type relation	Between the superclass and the association source class
			Association multiplicity restriction	Between the association source class and the subclasses
			Cycle	/
			Different path	/
	Restrict Property	yes	may trigger the removal of Inherited Association Multiplicity Restriction	If the multiplicity [1..1]
	Generalise Property	yes	Inherited Association Multiplicity Restriction	/
Refactoring Pattern	Extract Class	no	Type Relation	If the source class in specialized
	Inline Class	yes	no	/
	Extract Superclass	no	no	/
	Flatten Hierarchy	yes	no	/
	Class to Association	yes	no	/
	Association to Class	yes	no	/
	Generalization to Composition	yes	Removal of Cycle	/
			Removal of Type relation	/
	Introduce Composite Pattern	yes		/

TABLE IV
STATEMACHINE FEATURES FOR ITS TWO VERSIONS 1.5 AND 2.0

	Metamodel						Ocl		
StateMachine	Class	Enumeration	Attribute	Operation	Association	**Total**	Correct	Incorrect	**Total**
Version 1.5	14	1	5	2	25	**47**	25	5	**30**
Version 2.0	15	2	6	6	40	**69**	38	11	**49**

operations (Table V). 63% (46 out of 73) of these operations aim to add new modeling elements in the metamodel. 14% (10 out of 73) are deletion operations and 10% (7 out of 73) are renaming operations. Other operations (13%) concern property manipulation such as pullUp attribute/property, pushDown attribute/property, move property, etc.

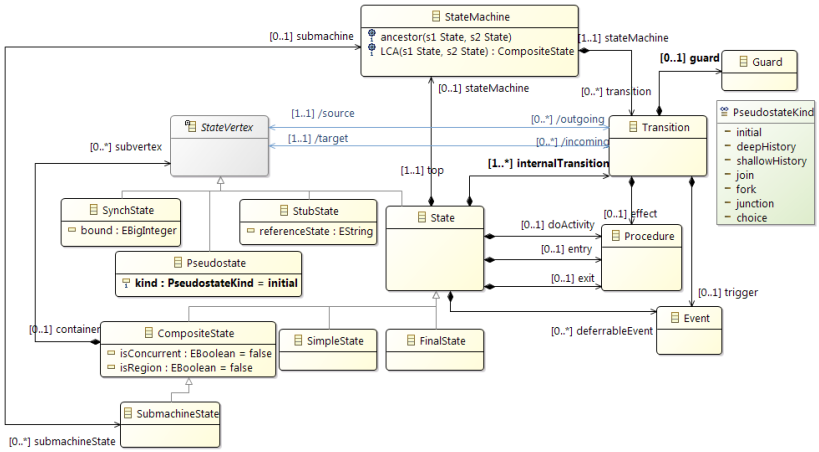

Fig. 5. StateMachine metamodel version 1.5 (from [9])

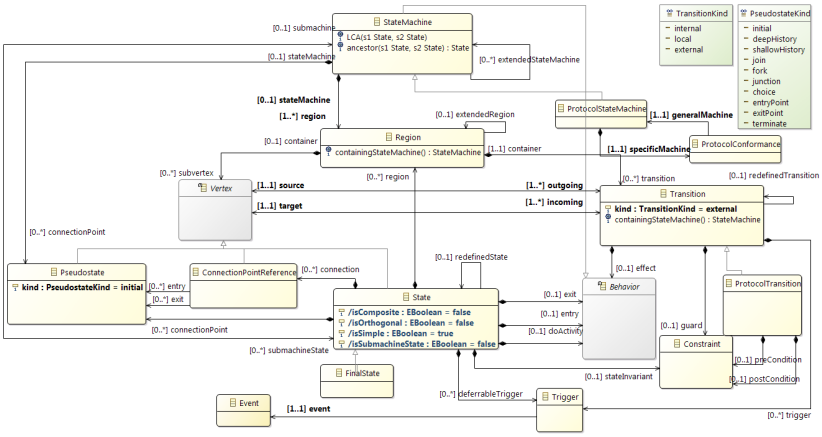

Fig. 6. StateMachine metamodel version 2.0 (from [10])

TABLE V
LIST OF EVOLUTION OPERATIONS.

Add class	4	Remove association	4
Add association	23	Restrict property	2
Add attribute	4	Generalize property	1
Add operation	4	Extract class	1
Add enumeration	1	Move property	2
Add literal	6	PushDown Association	1
Add generalization	4	PullUp attribute	1
Remove class	4	PullUp association	2
Remove attribute	1	Rename Element	7
Remove generalization	1	**Total**	**73**

D. Data processing & method

To automatically execute these 73 operations on our tool, we have developed a Java program. This program loads the metamodel 1.5 with its 25 constraints and applies the evolution operations one by one. The application of each operation returns three outcomes: modification of the metamodel by integrating the changes generated by the operation, adapting the OCL constraints after the application of the operation, and finally, generating a set of notifications related to MIS occurrences. After the application of all 73 operations on metamodel 1.5, we obtained: 1) an evolved metamodel corresponding to version 2.0; 2) a set of adapted constraints; 3) a set of notifications for new potential constraints (corresponding to the identified MIS).

To answer the first research question, we calculated the number of OCL constraints, automatically coevolved ($COEVOL$) by our tool, and compared them to the constraints present in version 2.0 (coevolution made by UML designers). As a result of this comparison, we obtained 4 sets: constraints coevolved by our method and by the UML designers ($COEVOL_{TP}$); constraints coevolved by our tool and not coevolved by UML designers ($COEVOL_{FP}$), constraints not coevolved by our tool and not coevolved by UML designers ($COEVOL_{TN}$), and constraints not coevolved by our tool but coevolved by UML designers ($COEVOL_{FN}$). With these four sets, we calculated the precision and the recall of our approach, as follows:

$$Precision_{COEVOL} = \frac{COEVOL_{TP}}{COEVOL_{TP} + COEVOL_{FP}} \quad (1)$$

$$Recall_{COEVOL} = \frac{COEVOL_{TP}}{COEVOL_{TP} + COEVOL_{FN}} \quad (2)$$

To answer the second search question, we calculated the number of the notified MIS occurrences generated by our tool. Then, for each notification, we looked for an OCL constraint that was added or removed in version 2.0, and that avoids the MIS occurrence indicated by our tool. Indeed, notifications generated by our tool emit a suspicion that a constraint must be added or removed after the application of an evolution operation. This comparison resulted in three sets: 1) a set of MIS for which OCL constraints were added by UML designers in version 2.0 (MIS_{TP}); 2) a set of MIS for which constraints have not been added by the UML designers (MIS_{FP}); 3) a set of constraints added by UML designers that do not correspond to the identified (MIS_{FN}). With these three sets, we calculated the precision and the recall of our method.

$$Precision_{MIS} = \frac{MIS_{TP}}{MIS_{TP} + MIS_{FP}} \quad (3)$$

$$Recall_{MIS} = \frac{MIS_{TP}}{MIS_{TP} + MIS_{FN}} \quad (4)$$

V. Results

A. *RQ1. What are the performances of our approach in the coevolution of OCL constraints to the evolution of their meta-model?*

The application of the 73 evolution operations on the StateMachine metamodel 1.5 has enabled 15 OCL constraints to be adapted from the 25 existing constraints. We obtained a precision of 1 and a recall of 1. The precision shows that the constraints coevolved by our tool have been well coevolved manually by the UML designers. This result is very interesting because the coevolution of these 15 OCL constraints took only a few seconds, which represents a considerable time saving compared to manual coevolution.

During this experiment, we noticed that the UML designers adopted other coevolutions than the ones proposed by our tool. For example, when deleting a metamodeling element, our tool proposes the removal of constraints related to this element. However, UML designers in some cases do not remove the constraints but rewrite them. Taking as an example the following constraint.

context Transition
inv inv:(self.stateMachine-¿notEmpty() **and not**
oclIsKindOf(self.stateMachine, ActivityGraph)) **implies**
self.target**.oclIsKindOf**(Pseudostate) **implies**
((self.target**.oclAsType**(Pseudostate).kind = #join) **implies**
(self.source**.oclIsKindOf**(State)))

Our tool proposed to remove this constraint because it refers to the *stateMachine* association between the *Transition* class and the *StateMachine* class, which was removed in version 2.0 of the StateMachine metamodel. However, UML designers preferred to modify it rather than to delete it as follows.

context Transition
inv inv: ((self.target**.oclAsType**(Pseudostate).kind = #join)
and (self.source**.oclIsKindOf**(State)))

On the other hand, automatic coevolution was able to detect constraints that should be coevolved and which have been omitted by the UML designers. Indeed, during the application of the evolution operation *Remove association "top" from "StateMachine" class*, our tool removed the following constraint.

context Transition
inv inv: self.source**.oclIsKindOf** (Pseudostate) **implies**
(self.source**.oclAsType**(Pseudostate).kind = #initial)
implies (self.source.container = self.stateMachine.top)
implies ((self.trigger-> isEmpty()) **or**
(self.trigger.stereotype.name =' create')

However, this constraint has not been removed in StateMachine version 2.0 and it is one of the 11 unparsed constraints. It was also surprising to note that this constraint dragged into all other versions of StateMachine (from 2.0, 2.1.2, 2.2, 2.3, and 2.4) without correction. It was corrected by UML designers in the latest version of UML, namely version 2.5. It took 10 years for UML designers to correct this constraint when our tool immediately detected it. These results show that the manual coevolution of OCL constraints is a long and error-prone process. The assistance of designers during the evolution of metamodels by automatically coevolving the OCL constraints is then a very useful and helpful process.

B. *RQ2. What are the performances in the notification of potential new constraints related to MIS?*

Regarding MIS, our tool has generated 103 notifications of potential MIS. Out of these 103 notifications (MIS_{TP} + MIS_{FP}), 20 (MIS_{TP}) were actually about constraints that have been added in version 2.0 by UML designers. With these performances, we obtained a precision of 0.20 and a recall of 0.62 ($MIS_{FN} = 12$). In the light of these results, it seems important for us to improve the performance of our approach especially concerning the number of false positives generated (reduction of the number of notifications). Indeed, we found that some notifications overlapped and were therefore counted twice. However, when looking at the results in more detail, other factors also had an impact on the results.

False negatives (constraints that do not match the MIS) are not really false negatives, because our approach looks only for MIS but not for all the constraints added in version 2.0. Indeed, the 12 constraints added by the experts and which do not correspond to MIS, are specific constraints and are highly dependent on the business domain and not MIS-related. If we consider false negatives as all constrained MIS that our tool

could not notify, then we get a recall of 1. Our tool can identify all MIS related to an evolution operation.

We also found that some false positives (notification of MIS but no written constraints for this MIS) may be omitted by UML designers. Indeed, we found that 3 MIS identified by our tool have been completed in version 2.5 by 3 OCL constraints. These MIS were about the addition of the *Kind* enumeration in the *Transition* class. We strongly believe that other MIS require OCL constraints but that experts have forgotten them. For example, the operation *Add association "connectionPoint" from State to "Pseudostate"* generated one potential MIS that points out the need to add an OCL constraint in order to restrict the multiplicity of the *connectionPoint* association for the *FinalState* sub-class. In other words, maybe *FinalState* should not have connection Points. While this seems logical, no OCL constraint was found in the specification that meets this MIS. But, we are convinced that some of this MIS does require a constraint.

In the end, we found that the UML designers added 23 new OCL constraints between version 1.5 and version 2.0. Out of these 23, our tool was able to notify 11 constraints, which corresponds to 48% of new constraints added. Knowing that, we are convinced that some constraints have been omitted in the specification.

VI. Threats to validity

In this section, we discuss below the strategies taken to avoid some threats to the validity of the results and ensure a certain quality of our study. According to [28], the threats to the validity of a case study are: construct validity, internal validity, external validity, and reliability.

A. Construct validity

Construct validity ensures that the carried measures allow to measure what it was intended to be measured. At the level of coevolution of OCL constraints, there are no potential threats of this type. Because, we have compared OCL constraints. As the OCL language is formal, there is no risk of confusion or ambiguity. However, regarding the notification of MIS, a risk may emerge because we have compared the existing constraints, written in OCL, with the notifications (textual form). To avoid this risk, we have defined a constraint template for each MIS that allows us to avoid it. We then compared these templates with the constraints written by the experts.

B. Internal validity

Internal validity concerns the consistency of the conclusions drawn on cause and effect between results. Our study is less affected by this type of validity.

C. External validity

External validity refers to the generalizability of the conclusions to other populations, domains, etc. We have studied this validity along two axes: OCL coevolution and MIS notification. On the OCL coevolution aspect, we can say that the results obtained can be generalized to other metamodels. But, on the MIS notification one, the results cannot be generalized, because they are strongly dependent on the type of the applied evolution operations. Indeed, in [3], studies have shown that the number of MIS in a metamodel is strongly dependent on its metamodeling elements and therefore on the type of the applied evolution operation. To avoid this threat, we chose a case study with a large and varied set of evolution operations. In fact, the case study allowed us to apply 18 types of evolution operation among the 31 previously mentioned.

D. Reliability

The reliability of the study consists in the ability of the results to be reproduced and replicated by other researchers. To increase the reliability of our study, we chose to use a metamodel known by the community and all the documentation of this case study is freely accessible via the Internet. We have also provided in this article, the set of evolutionary operations and their impact on MIS.

VII. Related Work

Related works can be divided into three categories. The first one concerns the approaches that evolve metamodels. The second one encompasses the OCL constraint coevolution approaches. Lastly, the third category concerns the approaches that study the conjunctive use of metamodels, class diagrams, and the Object Constraints Language.

A. Metamodel evolution

The evolution of metamodels is a topic that was studied by many research works. These works can be classified into three categories according to [27]. The first one concerns the approaches where the modelers encode manually the migration strategy using programming languages such as Java, or model transformation languages like QVT. In [7], Garces et al. compared two metamodel versions to capture the differences between them. The transformation rules are then used to adapt models automatically. [13] proposes an approach to coevolve models. It starts by detecting change either by comparing metamodels or by analyzing the change sequence applied to the metamodel old version. Then, the identified changes are classified depending on their impact on the model instances. Finally, an appropriate migration algorithm for model migration is determined. The above mentioned approaches rely on the copy rules, which can be very complex and time-consuming to do manually. Consequently, the second category of metamodel evolution approaches appeared. This category concerns the approaches that use matching techniques to infer migration strategies by comparing the old and new metamodel versions. In these approaches [16], [22], [23], [26], [29], the coevolution rules are specified as transformation rules using a unified metamodel that represents both metamodel versions. Then, an analysis is performed to remove modeling elements that are not present in the metamodel's new version. Hence, the models can be updated to be conform to the new metamodel. The third category encompasses the approaches that capture the metamodel changes as evolution operators [17].

The performance of these approaches depends on the completeness of the set of evolution operators. Wachsmuth [30] recorded a set of operators to evolve metamodels and coevolve models. Marković and Barr [25] proposed rules to coevolve class diagrams and OCL constraints. Hassam and al. [15] proposed operations for coevolving metamodels and OCL constraints. Kruse et al. [20] gathered all the operators present in the literature and summarised them. Also, inspired by Fowler [6] who proposed code refactoring operators, Kruse et al. have proposed some complex metamodel evolution operators such as *Introduce Composite Pattern*. Each of the above mentioned papers have contributed in completing the metamodel evolution library by providing additional operators which improved considerably the operator-based metamodel evolution approach.

B. OCL constraints coevolution

The coevolution of OCL constraints that follows the evolution of metamodels was the center of attention of many research works. We may distinguish semi-automatic coevolution methods. For example, Hassam et al. [14], [15] proposed to highlight obsolete constraints after metamodel evolution. The semi-automatic approach shows the updates that need to be applied to the OCL constraints using the Query View Transformation (QVT) language [12]. Khelladi et al. [18], [19] propose a semi-automatic approach for recording metamodel changes in chronological order, and thus detecting changes and applying resolution strategies to adapt the constraints. Kusel et al. [21] propose semi-automatic resolution actions for the coevolution of OCL expressions in model transformations as a response to metamodel evolution. The common point between these approaches is the fact that they all rely on existing OCL constraints sets. This strategy gives positive results when the set of OCL constraints that undergoes the coevolution is complete. Also, these approaches are straightforward when the metamodel does not undergo too many evolutions that change its structure. The weakness of these approaches lies in the inability to add new constraints that complete new modeling elements.

On the other hand, other works propose fully automated methods. For example, Cabot et al. [2] focus mainly on removal operations. Hence, this automatic approach has as a goal to delete constraints or parts of them. It targets the elements that are not present in the metamodel anymore. This approach focuses on removing constraints that cause inconsistencies with the conceptual schemes after subtracting operations. The approach is very powerful to avoid inconsistencies, but needs to be completed with other methods to take into consideration the other aspects of the OCL coevolution. Demuth et al. [4], [5] propose a template-based approach that defines a generic structure for OCL constraints that are then instantiated to update the constraints after metamodel evolution. Markovic et al. [24] proposed an approach in which they formalize the most important refactoring rules for class diagrams and classify them with respect to their impact on annotated OCL constraints. This approach focuses mainly on classifying the refactoring rules depending on their impact on the OCL constraints. Batot et al. [1] propose an automatic two-steps process to automatically coevolve metamodels and OCL constraints using a genetic algorithm. For each OCL constraint, the genetic algorithm explores the possible changes in the modeling space. The authors have defined a set of objective functions that helps to chose the changes that respect these objectives. The strength of this approach is the use of randomization using genetic operators, which can give rise to multiple OCL constraints. Just like the previous approaches, it does not fully exploit the new metamodel structure to extract all the new information from it.

While all the work that coevolve and refactor OCL constraints rely on the metamodel and the existing constraint set to perform updates, our approach relies only on metamodel structure. In addition to this, our approach can be used to add new constraints that did not appear on the old version of the metamodel.

C. Metamodel structures analysis

In our approach, we relied on the results of the analysis made in [3] on existing metamodels and their OCL constraints. This study had as a goal to identify metamodel structures that are often constrained with the OCL language. Other studies have been performed on metamodels and class diagrams to determine the weaknesses of both languages. Wahler et al. [31]–[33] have captured the class diagram limits in the form of Anti-Patterns, and proposed after that a set of OCL constraints patterns that can be used to complete these imprecise structures. The main reason behind our use of MIS instead of Anti-patterns is due to the large empirical evaluation performed in [3]. From another perspective, Cadavid et al. have performed an analysis over metamodels and their OCL constraints in order to investigate the conjunctive use of both MOF and OCL languages. The study led to the identification of a set of OCL patterns that are repeatedly found in the metamodels. These works are powerful when it comes to writing new OCL constraints, especially during the metamodel creation phase. However, the metamodel structure analysis needs to be complemented with artifacts when it is used for refactoring metamodels.

VIII. Conclusion & Perspectives

In this paper, we have proposed a novel approach to assist the coevolution of metamodels and their related OCL constraints by notifying raised MIS occurrences. This approach aims at completing existing techniques that intend to coevolve metamodels with their existing OCL constraints. Our approach relies on the metamodel structure to identify the changes that may need OCL constraints.

To evaluate our approach, we have performed a case study with the UML's State Machine metamodel by comparing the 1.5 version with the 2.0 version. The results showed that out of 23 new OCL constraints, our approach was able to notify the need to define 48% of them, which is a promising result. Our approach performances are still under improvement, especially

to reduce the number of false positives. Indeed, we did an exhaustive MIS search on all the metamodel, without using any other technique to reduce the false positive without losing precision.

We plan to improve our help system by using machine learning, which exploits all the data already collected, first, by studying metamodels to propose different metamodel categorizations. This may guide MIS search according to the metamodel family. We also think about using intelligent pattern recognition algorithms to search MIS instances provoked by the metamodel evolution only around the new concepts. These ideas are under study and will be presented in our future works.

REFERENCES

[1] E. Batot, W. Kessentini, H. Sahraoui, and M. Famelis. Heuristic-based recommendation for metamodel — ocl coevolution. In *2017 ACM/IEEE 20th International Conference on Model Driven Engineering Languages and Systems (MODELS)*, pages 210–220, Sep. 2017.

[2] Jordi Cabot and Jordi Conesa. Automatic integrity constraint evolution due to model subtract operations. In *International Conference on Conceptual Modeling*, pages 350–362. Springer, 2004.

[3] Elyes Cherfa, Soraya Kesraoui, Chouki Tibermacine, Regis Fleurquin, and Salah Sadou. On investigating metamodel inaccurate structures. In *Proceedings of the 35th Annual ACM Symposium on Applied Computing*, pages 1642–1649, 2020.

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

[5] Andreas Demuth, Roberto E Lopez-Herrejon, and Alexander Egyed. Supporting the co-evolution of metamodels and constraints through incremental constraint management. In *International Conference on Model Driven Engineering Languages and Systems*, pages 287–303. Springer, 2013.

[6] Martin Fowler. *Refactoring: improving the design of existing code*. Addison-Wesley Professional, 2018.

[7] Kelly Garcés, Frédéric Jouault, Pierre Cointe, and Jean Bézivin. Managing model adaptation by precise detection of metamodel changes. In *European Conference on Model Driven Architecture-Foundations and Applications*, pages 34–49. Springer, 2009.

[8] Object Management Group. https://www.omg.org/.

[9] Object Management Group. Unified modeling language 1.5. https://www.omg.org/spec/UML/1.5/, year = 2003.

[10] Object Management Group. Unified modeling language 2.0. https://www.omg.org/spec/UML/2.0/, year = 2005.

[11] Object Management Group. Meta-object facility 2.5.1. https://www.omg.org/spec/MOF/2.5.1/, 2016.

[12] Object Management Group. Mof query/view/transformation 1.3. https://www.omg.org/spec/QVT/1.3/, 2016.

[13] Boris Gruschko, Dimitrios Kolovos, and Richard Paige. Towards synchronizing models with evolving metamodels. In *Proceedings of the International Workshop on Model-Driven Software Evolution*, page 3. Amsterdam, The Netherlands, 2007.

[14] Kahina Hassam, Salah Sadou, and Régis Fleurquin. Adapting ocl constraints after a refactoring of their model using an mde process. In *9th edition of the BElgian-NEtherlands software eVOLution seminar (BENEVOL 2010)*, pages 16–27, 2010.

[15] Kahina Hassam, Salah Sadou, Vincent Le Gloahec, and Regis Fleurquin. Assistance system for ocl constraints adaptation during metamodel evolution. In *2011 15th European Conference on Software Maintenance and Reengineering*, pages 151–160. IEEE, 2011.

[16] Markus Herrmannsdoerfer, Sebastian Benz, Elmar Juergens, et al. Cope: A language for the coupled evolution of metamodels and models. In *1st International Workshop on Model Co-Evolution and Consistency Management*, 2008.

[17] Markus Herrmannsdoerfer, Sander D Vermolen, and Guido Wachsmuth. An extensive catalog of operators for the coupled evolution of metamodels and models. In *International Conference on Software Language Engineering*, pages 163–182. Springer, 2010.

[18] Djamel Eddine Khelladi, Regina Hebig, Reda Bendraou, Jacques Robin, and Marie-Pierre Gervais. Detecting complex changes and refactorings during (meta) model evolution. *Information Systems*, 62:220–241, 2016.

[19] Djamel Eddine Khelladi, Regina Hebig, Reda Bendraou, Jacques Robin, and Marie-Pierre Gervais. Metamodel and constraints co-evolution: A semi automatic maintenance of ocl constraints. In *International Conference on Software Reuse*, pages 333–349. Springer, 2016.

[20] Steffen Kruse. *Co-Evolution of Metamodels and Model Transformations: An operator-based, stepwise approach for the impact resolution of metamodel evolution on model transformations.* BoD–Books on Demand, 2015.

[21] Angelika Kusel, Juergen Etzlstorfer, Elisabeth Kapsammer, Werner Retschitzegger, Johannes Schoenboeck, Wieland Schwinger, and Manuel Wimmer. Systematic co-evolution of ocl expressions. *11th APCCM*, 27:30, 2015.

[22] Florian Mantz, Gabriele Taentzer, and Yngve Lamo. Co-transformation of type and instance graphs supporting merging of types and retyping. *Electronic Communications of the EASST*, 61, 2013.

[23] Florian Mantz, Gabriele Taentzer, Yngve Lamo, and Uwe Wolter. Co-evolving meta-models and their instance models: A formal approach based on graph transformation. *Science of Computer Programming*, 104:2–43, 2015.

[24] Slaviša Marković and Thomas Baar. Refactoring ocl annotated uml class diagrams. In *International Conference On Model Driven Engineering Languages And Systems*, pages 280–294. Springer, 2005.

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

[26] Bart Meyers, Manuel Wimmer, Antonio Cicchetti, and Jonathan Sprinkle. A generic in-place transformation-based approach to structured model co-evolution. *Electronic Communications of the EASST*, 42, 2012.

[27] Louis M Rose, Richard F Paige, Dimitrios S Kolovos, and Fiona AC Polack. An analysis of approaches to model migration. In *Proc. Joint MoDSE-MCCM Workshop*, pages 6–15, 2009.

[28] Per Runeson and Martin Höst. Guidelines for conducting and reporting case study research in software engineering. *Empirical software engineering*, 14(2):131, 2009.

[29] Gabriele Taentzer, Florian Mantz, Thorsten Arendt, and Yngve Lamo. Customizable model migration schemes for meta-model evolutions with multiplicity changes. In *International Conference on Model Driven Engineering Languages and Systems*, pages 254–270. Springer, 2013.

[30] Guido Wachsmuth. Metamodel adaptation and model co-adaptation. In *European Conference on Object-Oriented Programming*, pages 600–624. Springer, 2007.

[31] Michael Wahler, David Basin, Achim D Brucker, and Jana Koehler. Efficient analysis of pattern-based constraint specifications. *Software & Systems Modeling*, 9(2):225–255, 2010.

[32] Michael Wahler, Jana Koehler, and Achim D Brucker. Model-driven constraint engineering. *Electronic Communications of the EASST*, 5, 2007.

[33] Michael S Wahler. *Using patterns to develop consistent design constraints*. PhD thesis, ETH Zurich, 2008.

[34] Jos B Warmer and Anneke G Kleppe. *The object constraint language: getting your models ready for MDA*. Addison-Wesley Professional, 2003.