Heuristic-based Recommendation for Metamodel–OCL Coevolution

Edouard Batot
Université de Montréal
batotedo@iro.umontreal.ca

Wael Kessentini
Université de Montréal
kessentw@iro.umontreal.ca

Houari Sahraoui
Université de Montréal
sahraouh@iro.umontreal.ca

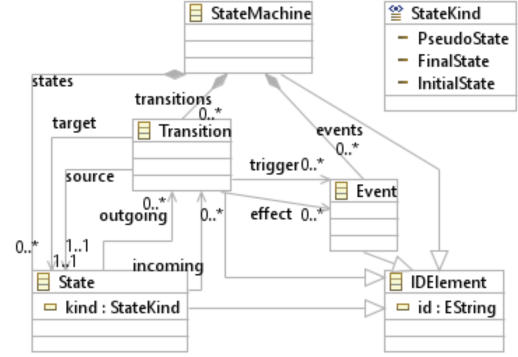
Michalis Famelis
Université de Montréal
famelis@iro.umontreal.ca

***Abstract*—We propose a novel approach for solving the problem of coevolution between metamodels and OCL constraints. Unlike existing solutions, our approach does not rely on predefined update rules and explicit tracking of high level changes to the metamodel. Rather, we pose it as a multi-objective optimization problem, exploring the space of possible OCL modifications to identify solutions that (a) do not violate the structure of the new version of the metamodel, (b) minimize changes to existing constraints, and (c) minimize loss of information. Finally, we recommend an appropriate subset of solutions to the user. We evaluate our approach on three cases of metamodel and OCL coevolution. The results show that we recommend accurate solutions for updating OCL constraints, even for complex evolution changes.**

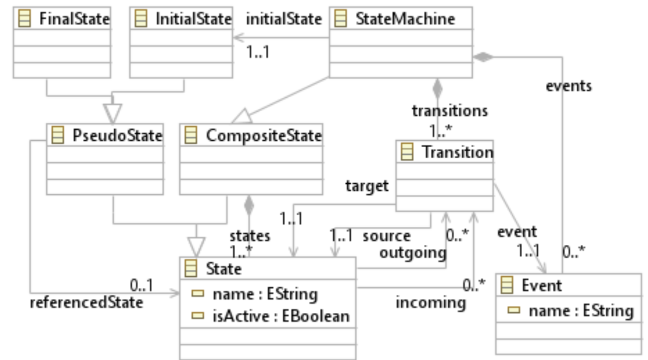
I. Introduction

The creation of modern software systems often depends on the use of domain-specific languages (DSLs) to empower domain experts to define and manipulate software artifacts using familiar abstractions and notations [1]. This is commonly accomplished using the Model-Driven Engineering (MDE) paradigm [2], whereby the abstract syntax of the DSL is defined using a metamodel that captures the main domain concepts, their properties and their relationships. The creation of DSLs typically involves multiple development iterations, meaning that the metamodel is modified often before reaching maturity. Such frequent changes in the metamodel inadvertently render obsolete the versions of dependent artifacts such as models, transformations, and well-formedness rules, typically expressed as OCL constraints [3]. To match the pace of DSL development, such artifacts have to be updated after each metamodel change, which is time-consuming and error-prone. This implies the need for significant automation for the coevolution of metamodels and related artifacts. In this paper, we focus specifically in the coevolution of OCL constraints in response to metamodel change.

To better illustrate the OCL coevolution problem, we introduce a scenario that we will use throughout the paper. Consider the following example, taken from [4] and [5], in which a simplified State Machine metamodel, shown in Figure 1(a) evolves to the one shown in Figure 1(b). Between the two versions, several different kinds of changes have taken place. Some are atomic: *e.g.,* the property name has been added to the class Event, and the reference effect has been removed from the class Transition. Other changes are more sophisticated: *e.g.,* the abstract class IDElement has been removed, the reference trigger in class Transition has been renamed event and its cardinality has changed. Finally, the example contains one complex modelling change: the typing of States using the enumeration StateKind via the attribute kind has been supplanted by a complex class hierarchy, involving the new classes CompositeState, PseudoState, FinalState, and InitialState.



(a) Version 1



(b) Version 2

Fig. 1: Evolution of the State Machine metamodel.

We now examine what constraints are impacted by the metamodel changes. The renaming of the reference trigger affects the constraint C_1:

```
context Transition inv trigger_event:
 Transition.allInstances()->forAll( t |
  t.source=self.source and t.trigger=self.trigger
  implies self=t)
```

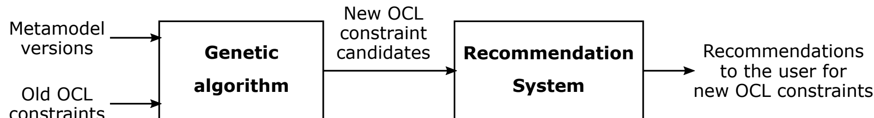


Fig. 2: Approach Overview.

This can be fixed by simply replacing the two occurrences of the string "trigger" by the string "event". However, the following constraint C_2, enforcing that initial states do not have incoming transitions, requires a more subtle update as it involves a complex structural change in the metamodel:

```
context State inv kind:
  self.kind = StateKind::InitialState
    implies self.incoming->size() = 0
```

Rather than testing the value of the attribute kind, the new constraint C_2' must test the type of the State instance:

```
context State inv kind:
  self.oclIsTypeOf(InitialState)
    implies self.incoming->size() = 0
```

Several approaches for coevolving OCL constraints have been developed (*e.g.,* [6], [7], [8], see Section VII). These approaches typically compare the two versions of the metamodel to detect semantically significant high-level changes, such as additions, deletions, refactoring of elements, etc. Then they use predefined rules for each type of change to update the constraints. Some approaches rely on manual change detection [9], [10], which can be difficult when dealing with large metamodels and complex changes. Others attempt to automate change detection [3] and thus rely on important assumptions such as that a change sequence has been recorded (thus precluding static comparisons between versions) or are based on heuristics that may not apply to any evolution context. Regardless of the method of change detection, however, all techniques rely on a set of predefined, context-dependent update rules.

In this paper, we propose a novel two-step approach to the coevolution of metamodels and OCL constraints, shown schematically in Figure 2. Unlike existing approaches, ours does not rely on predefined update rules and explicit tracking of high level metamodel changes. First, we use a multi-objective genetic algorithm to explore the space of possible OCL modifications to identify solutions that (a) do not violate the structure of the new version of the metamodel, (b) minimize changes to existing constraints, and (c) minimize loss of information. Then, we recommend an appropriate subset of solutions to the user. We developed two recommendation strategies: using a clustering algorithm based on the similarities between the identified solutions, and a ranking based on solutions' objective values. Evaluating our approach on three coevolution cases, we find that it produces accurate solution recommendations, even for complex metamodel changes. Specifically, we make the following **contributions**:

1) We propose a novel approach to the problem of coevolving OCL constraints during the evolution of metamodels that does not depend on the detection of changes in the metamodel. The problem is that this sort of detection can be tedious and error prone.

2) We propose to change the output of coevolution tools: instead of producing a single solution, we generate a set of potential candidate solutions and give the user the choice of the most appropriate one. This is advantageous because (a) it does not constrain the user and (b) it allows for serendipity and creativity (e.g., the user could decide to combine one or more solutions to produce something that the automated technique did not generate).

3) We propose to use meta-heuristic search, using genetic algorithms. This means that we expand the space of solutions that an automated tool for coevolution can traverse, allowing for the mechanical creation of potentially innovative solutions. This is because of the randomness introduced by the algorithm.

4) We propose an extensible framework for incorporating new strategies for guiding the heuristic search. These take the form of genetic mutation operators expressed at the meta-metamodel level. This allows incorporating existing and new coevolution strategies for specific contexts.

5) We propose an approach for presenting a large set of acceptable candidates in a way that is user-friendly. We do this by ranking the set of solutions or clustering it and generating representative exemplars for each cluster. In this way, the user can quickly shift through a large set of potential solutions.

The remainder of the paper is organized as follows: in Section II, we give basic background definitions. We describe the generation of solutions in Section IV and the selection of recommendations in Section V. We evaluate our approach in Section VI. We discuss related work in Section VII, and conclude with a discussion on lessons learned in Section VIII.

II. Background

A. Metamodel-OCL Coevolution

Metamodel coevolution refers to the process of adapting and correcting a set of modelling artifacts in response to the evolution of the metamodel on which they are strongly dependent [8]. In this paper, we focus on MOF metamodels [11], implemented in EMF [12] and metamodel-level well-formedness constraints expressed in OCL [13]. In the case of such constraints, the coevolution problem can be posed as follows: given a metamodel M and a set S of OCL constraints over M, if M evolves to a new version M', how should S be evolved to a new version S'? Below, we use primes to refer to concepts relevant to the new version.

Let $ins(M)$ be the –potentially infinite– set of all models that can be instantiated from M. In the following, we call such models "instance models". Then the set of constraints S partitions $ins(M)$ into two disjoint subsets: $ins(M) = acc(M) \sqcup$

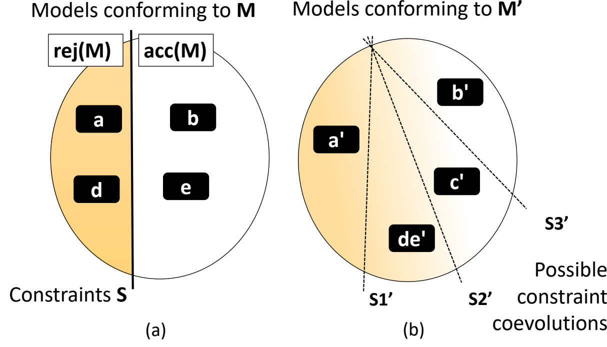


Fig. 3: Metamodel–OCL coevolution.

$rej(M)$, where $acc(M)$ contains the instance models of M that satisfy the constraints in S ("accepted") and $rej(M)$ those that do not ("rejected"). We show this schematically in Fig. 3(a), where $acc(M) = \{b, e\}$ and $rej(M) = \{a, d\}$.

The set of instance models of the new metamodel version M' can be coevolved [14] from $ins(M)$ to the new set $ins(M')$, shown in Fig. 3(b). The problem of coevolving S can thus be naïvely posed as identifying set of OCL constraints S' that partition $ins(M')$ such that $acc(M') = (acc(M))'$ and $rej(M') = (rej(M))'$. However, the coevolution of instance models does not necessarily produce a one-to-one correspondence between $ins(M)$ and $ins(M')$.

Assume for example a toy metamodel M_1 containing a single metaclass K_1. A developer evolves it to M'_1 by adding a new metaclass, such that M'_1 contains the metaclasses K'_1, K'_2. A model c that only contains instances of the metaclass K'_2 conforms to M'_1, and is thus in $ins(M'_1)$. Should it be placed in $acc(M'_1)$ or $rej(M'_1)$? There is no obvious model in $ins(M_1)$ from which to make this decision. Consider also the reverse scenario: the metamodel M_1 contains the metaclasses K_1, K_2 and is evolved by deleting the metaclass K_2 such that M'_1 only contains the metaclass K'_1. Assume two instance models d and e in $ins(M_1)$, where d contains just the elements $\{k_1 : K_1, k_{2d} : K_2\}$ and e contains just the elements $\{k_1 : K_1, k_{2e} : K_2\}$. Assume also that for whatever reason, the OCL constraints S of M_1 partition $ins(M_1)$ such that $d \in acc(M_1)$ and $e \in rej(M_1)$. Since in M'_1 the metaclass K_2 is deleted, their coevolved versions coincide to a model de' that simply contains the element $k_1 : K_1$. Should we coevolve the constraints to a new set S' that place this model in $acc(M'_1)$ (because $d \in acc(M_1)$) or in $rej(M'_1)$ (since $e \in rej(M_1)$)?

We illustrate this ambiguity in Fig. 3(b), where we show that there exist multiple possible partitions of $ins(M')$. The choice of appropriate partition depends on the *intent* of the developer responsible for the evolution of the metamodel. This intent is the developer's intuition about which of the models that can be instantiated from M' should be accepted by the set S' of coevolved OCL constraints and which should be rejected (put in $acc(M')$ and $rej(M')$ respectfully). Unless this intent is made explicit, the coevolution of OCL constraints cannot be fully automated. However, making it explicit may not be possible without rewriting the set of OCL constraints S' from scratch, or doing the coevolution manually. We thus consider the automated coevolution problem as providing support to the developer in order to identify the set S' of OCL constraints that best reflects her intent. In this paper, we use multi-objective optimization to evolve from S a set of candidate solutions for S' and then use a recommendation system to help the developer make a decision.

B. Multi-objective Optimization

In this section, we first introduce concepts related to multi-objective optimization and then describe NSGA-II [15], one of the widely used multi-objective optimization algorithms. Finding the solution to an optimization problem consists of finding optimal or near-optimal solutions with respect to some goals expressed in quality functions to maximize or to minimize a set of *objective functions*.

Definition 1 (MOP). A multi-objective optimization problem (MOP) entails minimizing or maximizing an objective function vector $f(x) = [f_1(x), f_2(x), ..., f_M(x)]$ of M objectives under some constraints.

The set of feasible solutions, *i.e.,* those that satisfy the problem constraints, defines the search space Ω. The resolution of a MOP consists in approximating the whole Pareto front.

Definition 2 (Pareto optimality). In the case of a minimization problem, a solution $x^* \in \Omega$ is *Pareto optimal* if $\forall x \in \Omega$ and $\forall m \in I = \{1, ..., M\}$ either $f_m(x) = f_m(x^*)$ or there is at least one $m \in I$ such that $f_m(x) > f_m(x^*)$.

In other words, x^* is Pareto optimal if no feasible solution exists, which would improve some objective without causing a simultaneous worsening in at least another one.

Definition 3 (Pareto dominance). A solution $u = (u_1, u_2, ..., u_n)$ is said to *dominate* another solution $v = (v_1, v_2, ..., v_n)$ (denoted by $f(u) \preceq f(v)$) iff $f(u)$ is partially less than $f(v)$. In other words, $\forall m \in \{1, ...M\}$ we have $f_m(u) \leq f_m(v)$ and $\exists m \in \{1, ..., M\}$ where $f_m(u) < f_m(v)$.

Definition 4 (Pareto optimal set). For a MOP f(x), the *Pareto optimal set* is $P^* = \{x \in \Omega | \neg \exists x' \in \Omega, f(x') \preceq f(x)\}$.

Definition 5 (Pareto front). For a given MOP f(x) and its Pareto optimal set P* the *Pareto front* (also called *Pareto optimal front*) is $PF^* = \{f(x), x \in P^*\}$.

In this paper we use the Non-dominated Sorting Genetic Algorithm-II (NSGA-II) [15], one of the most-used multi-objective evolutionary algorithms (EAs) in tackling real-world problems, including software engineering ones [16] to find trade-offs between different objectives. Fig. 4 describes the process. It begins by generating an offspring population from a parent one by means of variation operators such that both populations have the same size. NSGA-II defines two types of variation operators: *crossover* and *mutation*. The goal of crossover is to find new, and possibly better, combinations of the genetic material already present in the population. Mutation allows injecting new genetic material that potentially improves the population of solutions.

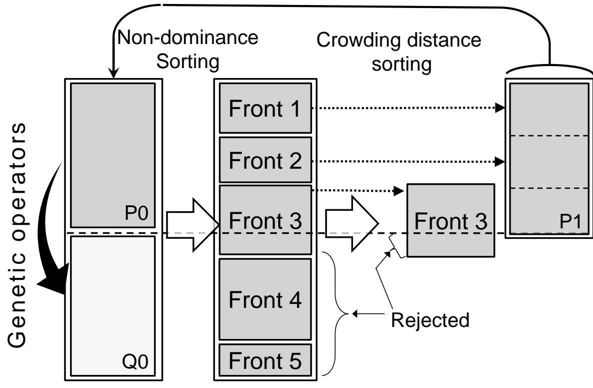


Fig. 4: Overview of the NSGA-II algorithm [15].

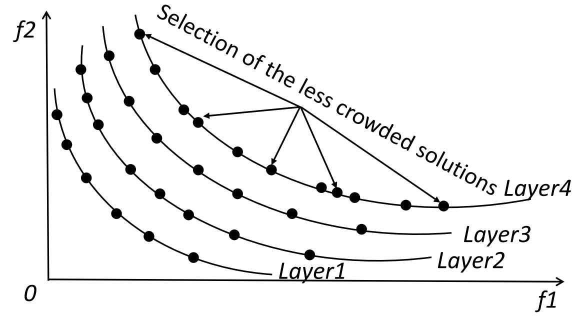


Fig. 5: NSGA-II Selection Mechanism.

After applying variation operators, NSGA-II ranks the merged population (parents and children) into several non-dominance layers, called fronts, as depicted in Fig. 5. Non-dominated solutions are assigned a rank of 1 and constitute the first layer (Pareto front). After removing solutions of the first layer, the non-dominated solutions form the second layer and so on and so forth until no non-dominated solutions remain.

After assigning solutions to fronts, each solution is assigned a diversity score, called *crowding distance*, inside each front. This distance defines a partial ranking inside the front which is used later in the algorithm to favour solutions that are far from the others in terms of objective values. A solution is then characterized by its layer and its crowding distance inside the layer.

To finish an iteration of the evolution, we perform the environmental selection to form the parent population for the next generation by picking half of the solutions. The solutions are included iteratively from the Pareto front to the lowest layers. If half of the population is reached inside a front then the crowding distance is used to complete the parent population. Fig. 5 shows an example of the selection process for two objectives. The solutions of the four first layers are included but not all those of 5th one. Some solutions of the 5th layer are selected based on their crowding distance values. In this way, most crowded solutions are the least likely to be selected; thereby emphasizing population diversification. To sum up, the Pareto ranking encourages convergence towards the near-optimal solution while the crowding ranking emphasizes diversity.

NSGA-II halts once a stopping criterion is satisfied. In this work, we use as stopping criterion a predefined number of iterations (generations). Other criteria can be used, such as convergence values for the objective functions, or a number of iterations without fitness improvement.

In the following, we describe how we adapted the NSGA-II algorithm to the problem of Metamodel–OCL coevolution.

III. Approach Setup

Our approach assumes as input two versions M and M' of a metamodel. We do not assume knowledge of the sequence of changes that were made to M to turn it into M'. However, we assume that the two versions are indeed related via evolution. We additionally assume as input a set S of OCL constraints for M.

As a preprocessing step, we compute the set D of *atomic* differences between the two versions. We do this in order to identify the constraints in S that are affected by the metamodel evolution. Specifically, we compare M and M' and register all metamodel elements that were deleted, added, or had their multiplicity modified between the two versions. Unlike existing work (see Section VII), computing atomic differences does not require the identification of conceptually high-level metamodel changes. Rather, it can be done directly from the two metamodel versions. For example, for the two State Machine metamodel versions in Fig. 1, the set D contains elements that were added, such as the containment association states between State and the new class CompositeState. It also contains elements that were deleted, such as the enumeration StateKind, or the generalization link between the class State and IDElement.

In general, the constraints in the set S are not applicable to M', since they may refer to elements of M that have since changed (*i.e.,* elements in D). For example, the constraint C_1 cannot be applied to the new version of the State Machine metamodel, since it refers to the element trigger, which has been changed to event. Our objective is thus to produce candidate sets of OCL constraints S' for M' that can be evolved from S.

Our approach must obey the following requirements: (a) the parts of OCL constraints in S that are not affected by the evolution (*i.e.,* do not refer to elements in D) should not be modified since information contained in the original constraints in S should be preserved as much as possible, (b) changed elements in D should be prioritized when computing OCL modifications, (c) generated constraints should have as few syntax errors as possible.

IV. Generative Coevolution of Solutions

We adapt the OCL coevolution problem into a multi-objective optimization problem that we solve using NSGA-II. In the following, we use the metaphors of genetic algorithms to show: (a) how a solution is represented and created; (b) how

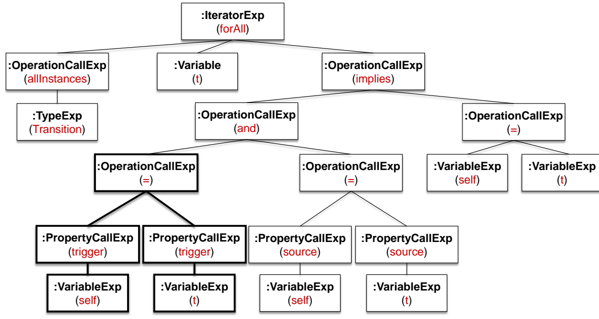


Fig. 6: Abstract syntax tree of the constraint C_1 in Section I

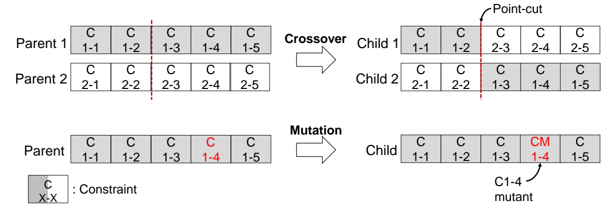


Fig. 7: Crossover and mutation adapted in our framework

solutions reproduce with each other at each iteration, and (c) how the fitness of a solution is measured.

A. Solution Representation and Solution Creation

A solution to our problem is a set of OCL constraints. In our metaphor, a set of constraints is a genetic entity, containing K ordered chromosomes (individual constraints). We only consider constraints affected by the evolution, ignoring all others; in the following, whenever we refer to S, we refer to the set of only affected constraints. The order of the chromosomes helps to uniquely identify each constraint within the solution. Each individual constraint is represented as an Abstract Syntax Tree (AST), using the OCL metamodel provided by the Eclipse Modelling Framework [17]. For example, the AST of the constraint C_1 is shown in Fig. 6.

To derive the initial generation G_0 of N solutions, we first derive $N/2$ solutions by applying random mutations to S. To these, we then apply crossovers to derive $N/2$ more solutions. In subsequent iterations of the NSGA-II algorithm, we apply genetic operations on each generation G_i to derive the new generation G_j.

B. Genetic Operators

When evolving a population of solutions, NSGA-II derives new solutions from existing ones using two kinds of operators: **crossover** and **mutation**.

1) Crossover operator: As illustrated in Fig. 7, our crossover operator uses a single cut-point crossover. Each parent solution is divided into two constraint subsets using a randomly selected cut point. The constraints in each subset must preserve their order. Then the constraint subsets of the parent solutions are exchanged to form two new solutions.

2) Mutation operators: We propose an extensible framework, where various coevolution scenarios can be expressed as individual mutation operators. We have implemented five such operators, based on examples of metamodel evolution available to us. This list is not exhaustive; in the future, we intend to augment it with update rules from published literature on OCL coevolution (see Section VII).

Our approach assumes that a *mutation operator store* containing all possible mutations is at hand. Given a randomly selected OCL element e in a solution, we randomly select a mutation operator from the store and apply it to the solution. This means that, while our framework is capable of producing solutions for any coevolution scenario, it does not enforce a particular coevolution strategy to the user.

We have implemented the following mutation operators:

- **Renaming** We define two renaming strategies: (a) changing the name of a single occurrence of e in the constraint, or (b) changing all its occurrences in the set of constraints. The new name is selected from the vocabulary of the evolved metamodel M', with the following weights: elements added during evolution are considered first (weight 10), followed by elements with same type as e (weight 5), elements with same source (only for edge type elements, weight 5), and all other M' elements (weight 1). This order reflects the intuition that when applying a Rename operator, the new name should most likely be taken from the set of added identifiers, then from the set of elements with the same type or elements with the same source, and finally from the set of all other elements. The precise numerical values where chosen empirically for the evaluation that we performed.
- **Context Change** The class context of the OCL constraint containing e is replaced by some other class in M'. For example, the context of constraint C_1 may change from Transition to Event.
- **Pruning** The element e is deleted, while maintaining the logical skeleton of the constraint. We do this by identifying the first ancestor a of e in the OCL abstract syntax tree that is of type Boolean. That ancestor and its entire sub-tree is then replaced by a boolean primitive, *i.e.,* true or false. Specifically, if a is an operand of a or statement, it is replaced by false; if it is an operand of an and or implies statement, it is replaced by true. For example, assume $e =$ self.trigger in the constraint C_1. Its first Boolean ancestor in the AST of C_1 is the structure: "self.trigger = t.trigger". Applying pruning will therefore result in replacing this structure with true. In Fig. 6, we indicate the elements of the structure to prune with bold borders.
- **Change Typing Method** If e refers to an Enumeration that has been deleted from M, and if M' contains classes whose names are the same as the values of the element referred by e, this mutation can happen. It consists of equality assertions on types, replaced by their homologue in the new type hierarchy using the method oclIsTypeOf.

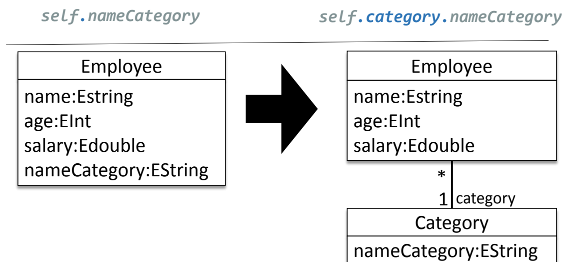


Fig. 8: Example application of the "Indirection Insertion" mutation operator

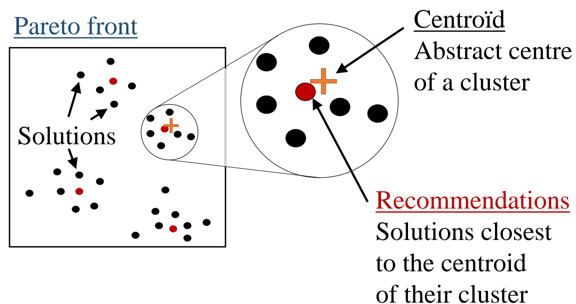


Fig. 9: Clustering of solutions: Solutions in the Pareto front are grouped by relative syntactic distance between each other's

The evolution of C_2 into C_2' in Section I is an example of this change. We do this instead of creating a new constraint with the class context of e's referred element to avoid modifying the chromosomic structure of the genetic entity (*i.e.,* the number of constraints in a solution), while maintaining semantics the same.

- **Indirection Insertion** If (a) e refers to a metamodel element f that was deleted during evolution, and (b) a metamodel element f' with the same name is found somewhere else in M', and (c) a link l exists between the source class of f and the source class of f', then an indirection l can be inserted between the source of e and f'. As illustrated in Fig. 8, in the new version of Employee, the nameCategory (f) has been moved into the class Category (the source of f'). Consequently, the excerpt of OCL constraint referring to that feature should be updated by inserting an indirection through category (l).

Finally, if the genetic reproduction fails to produce new genetic material (*i.e.,* crossover and mutations do not produce new solutions) we re-inject the initial set S. We found that this strategy helped the algorithm step out of certain local minima.

C. Objective Functions

To assess the fitness of each potential solution (*i.e.,* set of constraints, evolved from the initial set) with respect to the coevolution problem, we evaluate it using three objective functions:

- f_1 **– Number of changes:** For each solution, we record the number of mutations since S. Each mutation is additionally assigned a (user configurable) weight. By default each mutation has weight 1, but pruning has weight 2. The algorithm thus favours solutions with fewer changes in order to first explore solutions close to S. The additional default weight of pruning directs the algorithm to first consider other combinations of mutations before resorting to this more radical change.
- f_2 **– Number of syntax errors:** Since the first generation is based on S, it likely contains syntax errors with respect to the evolved metamodel version M'. Additional syntax errors can also be inadvertently introduced when randomly applying mutations. Since the goal is that the target solution S' is correct with respect to M', it should by definition be free of syntax errors. Syntax errors can be fixed by other mutations during evolution. The algorithm thus favours solutions with fewer syntax errors.
- f_3 **– Information loss:** To ensure information is not needlessly lost, we record the metamodel elements removed from the constraints in each solution. If an element of M was used in constraints in S, and it is not removed in M', then it should appear in generated solutions. For each solution we record the number of lost elements. The algorithm thus favours solutions with less information loss

V. Recommending Solutions

Once the NSGA-II algorithm stops, we have in the final generation a near-optimal set of solutions (Pareto), representing potential sets of OCL constraints appropriately adapted for the evolved metamodel. However, this set can be large. For example, in the experiments used in Section VI, the Pareto sets often include more than 50 solutions. It is therefore essential to provide users with additional support for understanding and managing this set. We propose to use a recommendation system that provides users with a few representative solutions. This way, users can quickly peruse the large set of near-optimal solutions and select a desirable candidate. To this end, we explore two alternative recommendation strategies.

A. Ranking strategy

The first strategy consists in taking, for each solution, the weighted average of the three objectives and use the resulting value to rank the solutions. Aggregating the three objectives can be done because all of them are defined within the same order of magnitude. Then the k top-ranked solutions are recommended.

B. Clustering strategy

As illustrated schematically in Fig. 9, the second strategy works by clustering the Pareto set to produce subsets of similar solutions and choosing representative exemplars from each one for presentation to the user.

First, we represent each solution by a N-dimensional difference vector, where N is the size of the Pareto set. The

difference vector encodes how different a solution is from other solutions. As a measure of difference, we use the Levenshtein distance [18] between the textual representations of the OCL constraints in the solutions. The Levenshtein distance of a string s_1 from a string s_2 is the number of deletions, insertions, and substitutions required to transform s_1 to s_2. For example, the two constraints C_2 and C_2' enforcing that initial states do not have incoming transitions in Section I have Levenshtein distance 18.

Then, we use the k-means algorithm [19] to cluster the set of difference vectors. Given a parameter k, which indicates the number of clusters and, hence, the number of solutions to recommend, k-means partitions the set to k clusters. This is done by computing an abstract "mean vector" for each cluster and placing each difference vector in the cluster containing the mean vector nearest to itself.

Finally, we select one representative solution from each cluster. To do this, we select the difference vector with the smallest distance from the mean vector of its cluster. The set of representative solutions is presented to the user. As the number k of clusters is a configuration parameter, the user can control how many solutions she is interested in inspecting before making a decision.

VI. Empirical Evaluation

As discussed in the Section I, existing Metamodel–OCL coevolution approaches are based of detecting conceptually high-level changes in the metamodel and use predetermined OCL updating strategies for each type of change. Our evaluation therefore aims to show that our approach is able to recommend accurate solutions to the OCL coevolution problem without depending on the detection of such high-level changes or predetermined update strategies. More specifically, we investigate the following research questions:

- RQ0: *Are the results of our approach attributable to the search strategy or to the number of explored solutions?* To answer this question, we compare the best solutions generated (a) using our approach, and (b) using random search over the same solution space (*i.e.,* exploring the same amount of solution candidates).
- RQ1: *To which extent our approach finds the expected solution?* To answer this question, we consider known coevolution scenarios, and check if the expected solution is in the Pareto set generated by our approach.
- RQ2: *To which extent our approach recommends the expected solution?* We answer this question by checking whether the expected solution is contained in the subset of solutions selected by our recommender from the Pareto set for presentation to the user.

A. Experimental Setting

Coevolution Cases. We selected three metamodel coevolution cases (*Family Structure*, *State Machine*, and *Project Management*) that demonstrate different levels of change complexity and that require diverse OCL updating alternatives. We then used these as ground truth for our evaluation.

Family Structure: This metamodel defines a schema for representing family structures. It has been used as an illustrative example in various publications in the software modelling research literature, such as [20], [21]. We use this case to experiment with metamodel changes that do not require sophisticated updating of the OCL constraints. The initial version of the metamodel contains 9 OCL constraints. Of these, 4 are affected by metamodel evolution. The required updating operations include: renaming attributes, pruning a portion of a constraint because the involved attribute was removed, and deleting a constraint whose context class was removed from the metamodel.

State Machine: We describe the metamodel evolution of this case in Section I. 7 OCL constraints complement the initial version of the metamodel[1] and 5 are affected by its evolution[2]. In addition to some basic updating operations, OCL coevolution also requires a complex change in the way certain constraints evaluate object types.

Project Management: This case, described in [22], represents a project management structure. It involves 8 OCL constraints from which 6 are impacted by the evolution. We included this case as an example of complex metamodel evolution. Specifically, the metamodel changes include splitting a class into three classes, creating a new abstract class, and moving attributes from the initial classes to the created ones. Thus, in addition to basic updating operations, the constraints require introducing indirection to test the moved attributes.

Algorithm Parameters. We used the following NSGA-II parameters: we set the stop criterion to 300 iterations, population size of 100 solutions per iteration, and crossover and mutation probabilities 0.9 and 0.6 respectively. For the weighted average recommendation strategy, we used the following weights: 40% for the syntax errors (f_2) and the information loss (f_3), and 20% for the number of changes (f_1). The rationale behind these weights is that we regard syntactic correctness and information loss as primary objectives, whereas we consider the number of changes to not diverge much from the initial constraints. Finally, we varied the size of the set of solutions produced by the recommender for presentation to the user from 3 to 20 solutions.

Validation Method. To answer **RQ0**, we performed 30 executions to account for the probabilistic nature of our algorithm. For each execution, our algorithm explores 15050 candidate solutions (initial iteration of 100 solutions plus 299 iterations requiring the creation of 50 solutions each, *i.e.,* half of the population of each generation is borrowed from the previous generation). We then also generated 15050 random solutions using the same technique as for the creation of the initial population, as described in Section IV-A. We then created a Pareto set for the random search using the dominance relationship as defined in Section II.

[1] https://github.com/atlanmod/LazyOcl_StateMachineExample
[2] http://ecariou.perso.univ-pau.fr/contracts/exec-contract.html

TABLE I: Statistical comparison of results between random search and our approach on three metamodel coevolution scenarios.

	Average		Mann Witney p-value	Effect size Cohen's d
	Random search	Our Approach		
Family	2,25	3,97	<0,001	4,35
State Machine	1,17	4,53	<0,001	4,87
Project Management	1,83	4	<0,001	3,11

For each of the two searches (our algorithm and random search) and each execution, we determine how many OCL constraints were fixed correctly by comparing each expected constraint with the candidates in the Pareto set S^P. For a constraint C_i of a metamodel M that needs to be updated and its ground truth equivalent C'_i in the evolved metamodel M', we say that C_i is correctly fixed if $\exists C_{ij} \in S^P \mid C_{ij} = C'_i$. Two OCL constraints are equal if the Levenshtein distance is 0 (cf. Section V).

Finally, we use the Mann-Whitney test to check if there is a significant difference between the results of our algorithm and those of a random search on the 30-executions samples.

To answer **RQ1**, we look at the distribution of the number of correctly updated constraints over the 30 executions of our algorithm, inside the Pareto set. We say that a constraint whose syntax exactly matches the expected evolved version has been *correctly updated*. The more often our algorithm is able to find a high number of correctly updated constraints, the more its results can be used to generate recommendations to the user.

To answer **RQ2**, we compare the two recommendation strategies described in Section V, based on their ability to suggest the expected updating solution within a limited number of recommendations. We want to assess whether presenting more recommendations to the user means that we also show more correctly fixed constraints. We thus compare the accuracy of the two recommendation strategies with the Pareto set for increasing number of recommendations.

B. Results and Interpretation

RQ0: *Is our approach better than a random search?* We summarize our findings in Table I. The results indicate that for the three validation cases, our approach finds far more correct solutions than the random search. For example, in the case of the State Machine metamodel, in which 5 constraints are impacted by the evolution, random search finds on average 1.17 correctly-fixed constraints, compared to 4.53 found by our approach. Similar differences are observed for the two other cases. All differences are statistically significant with a *p-value* < 0.001. In addition to the statistical significance, the effect size, which measures the importance of the difference relatively to the sample distributions, is very large (*i.e.,* between 3.11 and 4.87). Indeed, according to Sawilowsky in [23], an effect greater than 2 is consider as very large.

RQ1: *How efficient is our approach in finding coevolved OCL constraints?* The results in Table I show that we were able to find on average four or more correct constraints out of between four to six impacted constraints in the three considered cases. For the Family case, as indicated in Fig. 10a, we generate correct solutions for all the constraints in the majority of the executions (25 out of 30). For the remaining 5 executions, only one constraint is missing. For the State Machine case, we generated correct solutions for all constraints in 17 out of 30 executions. Out of a total five constraints, one constraint was missing in 12 executions, and two constraints in 1 execution. In the Project Management case, we generated four correct solutions out of a total of six constraints in all 30 executions.

Our approach generated correct solutions in at least one execution for all constraints of the Family and State Machine cases. However, the Project Management case contains two constraints for which our approach did not generate solutions in any execution. This is due to the fact that these two missing constraints require particularly complex changes. In the future, we aim to attempt to address such changes by expanding the mutation operator store of our approach.

RQ2: *How efficient are the proposed recommendation techniques?* We have plotted the accuracy of the two recommendation strategies for the three cases in Fig. 11. First, we observe that both recommendation strategies perform equally well. We thus conclude that simple ranking is a better recommendation strategy, since clustering is computationally costlier. Second, we the accuracy of the recommender (for either strategy) grows with the number of recommendations. We notice however that, for the three cases, the gain in accuracy tapers off after 7 recommendations, becoming negligible after 15 recommendations. In other words, we do not observe a dramatic increase in accuracy by the increase of the size of the set of recommendations beyond 7.

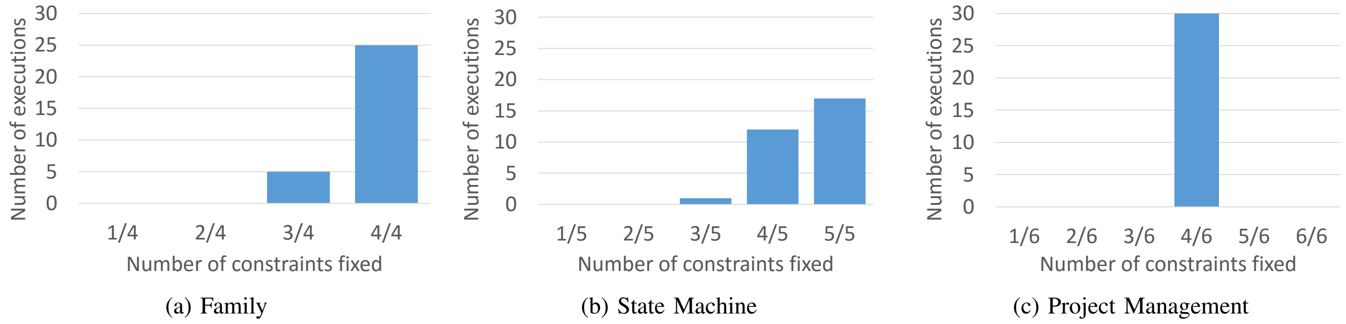
C. Threats to Validity

We aimed to show that our approach can produce good results without depending on the many inputs required by other the state-of-the-art approaches. However, these results have to be considered in light of the following threats to validity.

The first threat to validity is related to the selection of the experimental data. We did our best to select cases covering a wide range of Metamodel–OCL coevolution scenarios. However, additional, and larger cases are necessary to draw a final conclusion about the generalizability of our approach.

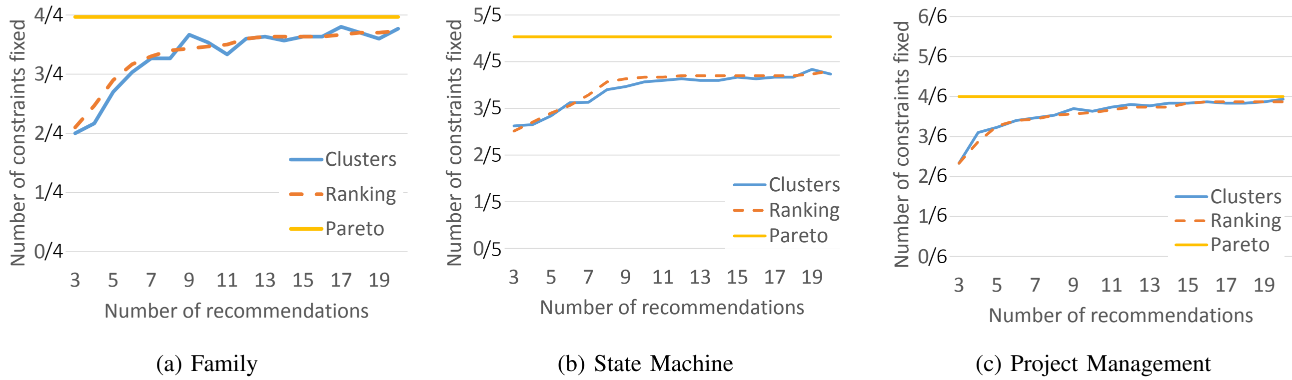
The second threat to validity is related to the probabilistic nature of NSGA-II. Indeed, different executions may produce different solutions. To mitigate this threat, we performed 30 executions to compare our approach with random search and to assess the correctness of the produced results.

We used syntactic comparison to determine whether a candidate constraint is the same as the expected one (ground truth). This is a threat to the validity as two solutions can differ syntactically, but be semantically equivalent. To mitigate this threat, we examined a sample of constraint pairs (generated/expected) across different degrees of Levenshtein distance to ensure that they are indeed different. For example, we encountered a pair in the State Machine metamodel where



(a) Family (b) State Machine (c) Project Management

Fig. 10: Distribution of the accuracy of solutions: number of constraints found in each execution. X-axis shows the number of solutions, Y-axis the number of valid constraints for each solution



(a) Family (b) State Machine (c) Project Management

Fig. 11: Comparison of the effectiveness of the two recommendation strategies. The X-axis shows the number of recommendations presented to the user. The Y-axis shows the number of correct solutions in the recommendation.

the generated constraint had distance 1 from the expected constraint, and yet was ranked as unsatisfactory by our approach. Upon closer inspection, we found that the only difference was the use of the identifier *events* instead of *event*. This was a good decision as the two identifiers refer to two different constructs in the metamodel (see Figure 1).

Finally, we used statistical tests only to answer **RQ0**, but not for **RQ1** and **RQ2**. The reason for this is that it is difficult to compare statistically our approach to the state-of-the-art, as we are not using the same inputs (metamodel changes and manually-defined updating rules). Our goal is not to outperform existing approaches, but rather to obtain comparable results albeit with less required effort.

VII. Related Work

In this section, we discuss approaches for the problem of coevolving metamodels and OCL constraints.

Coevolution has been subject for research for several decades in the database community [24], especially in the field of object-oriented databases [25]. In model-driven engineering, evolution is inevitable over the whole life cycle of complex software-intensive systems. In DSL design, modelling languages are subject to frequent change [26]. A change in one artifact involved in the language definition must be reflected in all other related artifacts such as models, test cases, OCL constraints, etc. Existing approaches can be classified as online or offline approaches. Online approaches perform instant coevolution for each change during the metamodel evolution, whereas offline approaches wait after the metamodel has been evolved to perform coevolution of the OCL constraints.

For online approaches, Demuth et al. [27] [6] provide templates that define a fixed structure for OCL constraints that are then instantiated to update the constraints. However, they are limited to 11 templates that cannot cover all changes at metamodel level. Hassam et al. [28] [22] propose a semi-automatic approach that highlights the constraints that should disappear after evolution and by formalizing the adaptation to be applied on impacted constraint after each operation on a metamodel using the QVT transformation language [29]. Similarly, Markovic et al. [30] [31] proposed an approach using QVT, in which they formalize the most important refactoring rules for class diagrams and classify them with respect to their impact on annotated OCL constraints. The advantage of online approaches is that the order of changes is preserved and no hidden changes are missed. However, the cancelling actions during evolution are apart of the detected changes.

For offline approaches, Kusel et al. [32] analyze the impact of metamodel evolution on OCL, then propose resolution actions in model transformation by means of ATL helpers.

Cabot et al. [7] focused on the metamodel changes that entail deleting elements. In particular, they aimed at removing the parts of OCL constraints that use the deleted elements. Khelladi et al. [8] [28] propose a semi-automatic approach that records in chronological order the changes to the metamodel. Then, they detect high-level changes and apply resolution strategies to adapt OCL constraints based on the structure of the impacted OCL constraint and the impacted location.

With respect to automation, Markovic et al. [30] [31], Cabot et al. [7], Demuth et al. [27] [6] are fully automated approaches. Hassam et al. [28] [22], Khelladi et al. [8] and Kusel et al. [32] are semi-automated approaches. As mentioned in the Section II-A, it is difficult to fully automate the updating of OCL constraints as the evolution intent is not explicit. Our approach, classified as offline, automates the generation of solution sets, but leaves the final decision to the user to select the solutions that better match her intent.

The above-mentioned approaches focus on identifying conceptually high-level changes to the metamodel in order to coevolve OCL constraints. They detect such changes either by manually comparing the two metamodel versions or by recording, matching or calculating their differences. Subsequently, they apply various change-specific strategies aimed at mirroring the high-level conceptual changes. Conversely, our approach, although it uses atomic changes to determine which constraints have to be updated, it does not link these changes to predefined updating strategies.

From another perspective, our approach generates a set of candidate solutions, and thus still requires user input to select the most appropriate candidate. In contrast, published approaches require the user to explicitly encode her preferences up front and produce a single solution. However, committing to a set of explicit preferences in advance might not be possible without an idea about the shape of expected results. In that sense, our approach allows users to avoid over-committing to specific coevolution strategies in the absence of enough information. Instead, the choice of strategy is deferred until the user has more information and can assess the relative quality of candidate solutions. This strategy has previously been successfully applied to managing design uncertainty [33] and to non-deterministic bidirectional transformations [34].

In addition to OCL coevolution, other research contributions have addressed the problem of coevolving artifacts after metamodel changes. The most important body of work targets metamodel-model coevolution [35]. Different approaches were proposed such as exploiting the formal relationship between models and their respective metamodels [36], defining coupled operations for metamodels and models [37], or using a multi-objective optimization strategy, similar to the one of this paper [14]. The coevolution of transformation has also been studied, either on its own [38] or coupled with models [32].

Finally, it is worth mentioning that search-based approaches have extensively been used in model-driven engineering [39]. In particular, they were used for metamodel-related artifacts such as generating OCL constraints [40], [41], model selection for metamodel testing [42], [43], and transformation learning [44].

VIII. Conclusion

The coevolution of metamodels and OCL constraints is crucial for automating the creation and maintenance of model-based domain-specific languages. Existing approaches depend heavily on either explicit tracking or automated identification of high-level conceptual changes to metamodels and use predefined rules to produced new versions of OCL constraints. In this paper, we propose a two-step process to automatically coevolve metamodels and OCL constraints. First, we pose the coevolution as a multi-objective optimization problem and use a genetic algorithm to evolve a population of candidate solutions. This produces a (potentially large) set of possible candidate OCL constraints. Second, we recommend to the user a smaller set of candidate solutions. This allows users to get a better grasp of the solution space and identify the most desirable candidates. We evaluated our approach on three cases of metamodel and OCL coevolution. We found that our approach identifies and recommends correct candidate solutions with high statistical significance for most cases.

In the future, we aim to address the main limitations of our approach, which we outline below. The first limitation concerns our choice of the set of mutation operators used by the genetic algorithm in the first step of our approach. The creation of the set was driven by the examples of metamodel evolution available to us at the time, and is therefore not exhaustive or complete. In the future, we intend to extract update rules from published literature on OCL coevolution and implement them as mutation operators in our genetic framework, in order to cover all possible updates.

Another limitation of our approach is the generation of recommendations using the syntactic difference between candidates, captured by the Levenshtein distance between the textual representations of OCL constraints. In the future, we intend to investigate more sophisticated ways of generating recommendations, that also take into account the semantics of the OCL language.

Finally, while the three objective functions that we used performed reasonably, we observed that they do not have enough discriminatory power. Specifically, we found cases where two candidates that were ranked as equivalent were in fact not equally good solutions. In other words, our objective functions cannot fully discriminate between correct and partially correct solutions. In the future we aim to develop objective functions that also take into account the semantics of OCL constraints. To do this, we intend to leverage metamodel test cases, *i.e.,* instance models of the initial metamodel for which the initial OCL constraints have known verification results (cf. Section II-A). These test cases can be evolved to the new version of the metamodel [14] and can then be subsequently used to test candidate solutions.

REFERENCES

[1] M. Fowler, *Domain-specific languages*. Pearson Education, 2010.

[2] S. Kelly and J.-P. Tolvanen, *Domain-specific modeling: enabling full code generation*. John Wiley & Sons, 2008.

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

[4] E. Cariou, C. Ballagny, A. Feugas, and F. Barbier, "Contracts for model execution verification," in *Modelling Foundations and Applications: 7th European Conference, ECMFA 2011, Birmingham, UK, June 6 - 9, 2011 Proceedings*, 2011.

[5] M. Tisi, R. Douence, and D. Wagelaar, "Lazy Evaluation for OCL," in *Models 2015: 15th International Workshop on OCL and Textual Modeling*, 2015.

[6] A. Demuth, R. E. Lopez-Herrejon, and A. Egyed, "Supporting the co-evolution of metamodels and constraints through incremental constraint management," in *International Conference on Model Driven Engineering Languages and Systems*, 2013, pp. 287–303.

[7] J. Cabot and J. Conesa, "Automatic integrity constraint evolution due to model subtract operations," in *International Conference on Conceptual Modeling*, 2004, pp. 350–362.

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

[9] S. Marković and T. Baar, "Refactoring ocl annotated uml class diagrams," in *8th International Conference on Model Driven Engineering Languages and Systems*, 2005, pp. 280–294.

[10] A. Cicchetti, D. Di Ruscio, R. Eramo, and A. Pierantonio, "Automating co-evolution in model-driven engineering," in *Int. conf. on Enterprise Distributed Object Computing Conference*, 2008, pp. 222–231.

[11] Object Management Group: Meta Object Facility Core Specification, version 2.0 (2006).

[12] F. Budinsky, S. A. Brodsky, and E. Merks, *Eclipse Modeling Framework*, 2003.

[13] J. Warmer and A. Kleppe, *The Object Constraint Language: Getting Your Models Ready for MDA*, 2003.

[14] W. Kessentini, H. A. Sahraoui, and M. Wimmer, "Automated metamodel/model co-evolution using a multi-objective optimization approach," in *Modelling Foundations and Applications - 12th European Conference, ECMFA 2016, Held as Part of STAF 2016, Vienna, Austria*, 2016.

[15] K. Deb, S. Agrawal, A. Pratap, and T. Meyarivan, "A fast elitist non-dominated sorting genetic algorithm for multi-objective optimisation: NSGA-II," in *Int. Conf. on Parallel Problem Solving from Nature - PPSN*, 2000.

[16] M. Harman, S. A. Mansouri, and Y. Zhang, "Search-based software engineering: Trends, techniques and applications," *ACM Comput. Surv.*, pp. 11:1–11:61, 2012.

[17] D. Steinberg, F. Budinsky, E. Merks, and M. Paternostro, *EMF: Eclipse Modeling Framework*. Pearson Education, 2008.

[18] V. I. Levenshtein, "Binary codes capable of correcting deletions, insertions, and reversals," in *Soviet Physics Doklady*, 1966, pp. 707–710.

[19] S. Lloyd, "Least squares quantization in pcm," *IEEE Transactions on Information Theory*, pp. 129–137, 1982.

[20] M. Gogolla, A. Vallecillo, L. Burgueno, and F. Hilken, "Employing classifying terms for testing model transformations," in *Proc. of the Int. Conf. on Model-Driven Engineering Languages and Systems*, 2015, pp. 312–321.

[21] L. Burgueno, J. Troya, M. Wimmer, and A. Vallecillo, "Static fault localization in model transformations," *IEEE Transactions on Software Engineering*, vol. 41, no. 5, pp. 490–506, 2015.

[22] K. Hassam, S. Sadou, and R. 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)*, 2010, pp. 16–27.

[23] S. Sawilowsky, "New effect size rules of thumb." *Journal of Modern Applied Statistical Methods*, pp. 597–599, 2009.

[24] J. F. Roddick, "Schema evolution in database systems: An annotated bibliography," *SIGMOD Rec.*, 1992.

[25] J. Banerjee, W. Kim, H.-J. Kim, and H. F. Korth, "Semantics and Implementation of Schema Evolution in Object-Oriented Databases," in *Proc. of SIGMOD*, 1987.

[26] B. Meyers and H. Vangheluwe, "A framework for evolution of modelling languages," *Sci. Comput. Program.*, 2011.

[27] A. Demuth, R. E. Lopez-Herrejon, and A. Egyed, "Automatically generating and adapting model constraints to support co-evolution of design models," in *Automated Software Engineering (ASE)*, 2012, pp. 302–305.

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

[29] Q. Omg, "Meta object facility (mof) 2.0 query/view/transformation specification," *Final Adopted Specification (November 2005)*, 2008.

[30] S. Marković and T. Baar, "Refactoring ocl annotated uml class diagrams," in *International Conference On Model Driven Engineering Languages And Systems*, 2005, pp. 280–294.

[31] ——, "Refactoring ocl annotated uml class diagrams," *Software and Systems Modeling*, pp. 25–47, 2008.

[32] A. Kusel, J. Etzlstorfer, E. Kapsammer, W. Retschitzegger, J. Schoenboeck, W. Schwinger, and M. Wimmer, "Systematic co-evolution of ocl expressions," *11th APCCM*, 2015.

[33] M. Famelis and M. Chechik, "Managing design-time uncertainty," *Software & Systems Modeling*, pp. 1–36, 2017.

[34] R. Eramo, A. Pierantonio, and G. Rosa, "Managing uncertainty in bidirectional model transformations," in *Proceedings of the 2015 ACM SIGPLAN International Conference on Software Language Engineering*, 2015.

[35] L. Rose, M. Herrmannsdoerfer, S. Mazanek, P. Van Gorp, S. Buchwald, T. Horn, E. Kalnina, A. Koch, K. Lano, B. Schätz, and M. Wimmer, "Graph and model transformation tools for model migration," *SoSyM*, vol. 13, no. 1, 2014.

[36] G. Taentzer, F. Mantz, and Y. Lamo, "Co-transformation of graphs and type graphs with application to model co-evolution," in *6th International Conference on Graph Transformations*, 2012, pp. 326–340.

[37] B. Meyers, M. Wimmer, A. Cicchetti, and J. Sprinkle, "A generic in-place transformation-based approach to structured model co-evolution," in *Proc. of MPM Workshop*, 2010.

[38] S. Kruse, "On the use of operators for the co–evolution of metamodels and transformations," *ME 2011 – Models and Evolution, Workshop at ACM/IEEE 14th Int. Conf. on Model Driven Engineering Languages and Systems (2011)*, 2011.

[39] I. Boussaïd, P. Siarry, and M. Ahmed-Nacer, "A survey on search-based model-driven engineering," *Automated Software Engineering*, vol. 24, no. 2, pp. 233–294, 2017.

[40] S. Ali, T. Yue, X. Qiu, and H. Lu, "Generating boundary values from OCL constraints using constraints rewriting and search algorithms," in *IEEE Congress on Evolutionary Computation*, 2016, pp. 379–386.

[41] M. Faunes, J. Cadavid, B. Baudry, H. Sahraoui, and B. Combemale, "Automatically searching for metamodel well-formedness rules in examples and counter-examples," in *Proc. of the Int. Conf. on Model-Driven Engineering Languages and Systems*, 2013, pp. 187–202.

[42] J. J. Cadavid, B. Baudry, and H. A. Sahraoui, "Searching the boundaries of a modeling space to test metamodels," in *Proc. of the Int. Conf. on Software Testing Verification and Validation*, 2012, pp. 131–140.

[43] E. Batot and H. A. Sahraoui, "A generic framework for model-set selection for the unification of testing and learning MDE tasks," in *19th International Conference on Model Driven Engineering Languages and Systems*, 2016, pp. 374–384.

[44] I. Baki and H. Sahraoui, "Multi-step learning and adaptive search for learning complex model transformations from examples," *ACM Trans. on Soft. Eng. and Methodology*, p. 36, 2015.