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 co-evolution 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 co-evolution. 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 renders 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.

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

$$\text{context Transition inv trigger\_event:}$$
$$\text{Transition.allInstances() \rightarrow forAll \{ t | t.source = self.source and t.trigger = self.trigger implies self = t \}}$$

\[ C_1: \]

\[ \text{context Transition inv trigger\_event:} \]
\[ \text{Transition.allInstances() \rightarrow forAll \{ t | t.source = self.source and t.trigger = self.trigger implies self = t \}} \]
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:

```plaintext
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:

```plaintext
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 an atomic 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 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.
We illustrate this in Fig. 3(b), where we show that there exist constraints to a new set $S$ simply contain the element $k$ that $d \in K$, also the reverse scenario: the metamodel $M$ model in $M'$ and $M$ contains just the $K$.

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 naively posed as identifying set of OCL constraints $S'$ 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, the coevolved versions $d', e'$ of the models both simply contain 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 in Fig. 3(b), where we show that there exist multiple possible partitions of $ins(M')$.

The choice of appropriate partition of $ins(M')$ 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

We introduce first the concept related to multi-objective optimization then we describe one of the widely used algorithms, NSGA-II [15]. An optimization problem consists in finding optimal or near-optimal solutions with respect to some goals expressed in quality functions to maximize or to minimize called objective functions.

We first give some background definitions related to multi-objective optimization.

**Definition 1 (MOP).** A multi-objective optimization problem (MOP) consists in 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 $\Omega$. 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 $f(u) \preceq f(v)$) if and only if $f(u)$ 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 $P^* = \{x \in \Omega | \exists x^* \in \Omega, f(x^*) \preceq f(x)\}$.

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

In this paper we use 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
In this way, most crowded solutions are the least likely to be included but not all those of 5th one. Some solutions of the population. Fig. 5 shows an example of the selection process for two objectives. The solutions of the four first layers are included iteratively from the Pareto front to the lowest 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 aims, later, at favoring 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 than 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 these changes are “small enough”, i.e., 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 changed between the two versions. Computing atomic differences does not require the identification of conceptually high-level metamodel changes and can be done statically from the two 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
solutions reproduce with each other at each iteration, (c) how the fitness of a solution is measured, and (d) the stopping criterion.

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 $N$ ordered chromosomes (individual constraints). We only consider constraints affected by the evolution, others are evicted. 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 $G_{i+1}$.

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 thus 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), and all other $M'$ elements (weight 1). This strategy was used to evolve the constraint $C_1$ by renaming trigger to event.

- **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 $e = \text{self.trigger}$ in the constraint $C_1$. Its first Boolean ancestor in the AST of $C_1$ is the structure: "self.trigger = self.trigger". Applying pruning will therefore result in replacing this structure with true.

- **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. The evolution of $C_1$ 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 chromosomal 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 feature $f$ deleted during evolution, and (b) a feature $f'$ 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 $f'$, 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 ($l$).

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 co-evolution 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 the centroid $M$ of each solution we record the number of lost elements. The algorithm thus favours solutions with less information loss.
that initial states do not have incoming transitions in Section I have Levenshtein distance 18.

Then, we use the \(k\)-means algorithm \cite{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 mentioned in the Introduction, exiting approaches are based on detecting high-level changes in the metamodel and use fixed OCL updating strategies for each change type. Our validation aims to show that without considering the high-level changes and the fixed updating strategies, our approach is able to recommend accurate OCL updating alternatives. More specifically, we investigate the following questions:

\begin{itemize}
  \item \textbf{RQ0:} Are the results of our approach attributable to the search strategy or to the number of explored solutions? We answer this question by exploring the same amount of solutions with our algorithm and with a random search, and compare the best solutions from both strategies.
  \item \textbf{RQ1:} To which extent our approach finds the expected solution? To answer this question, we check if the expected solution is in the Pareto set.
  \item \textbf{RQ2:} To which extent our approach recommend the expected solution? We answer this question by checking if the expected solution is in the Pareto subset that is recommended.
\end{itemize}

A. Experimental Setting

1) Coevolution Cases: We selected three coevolution cases that are concerned with different levels of change complexity and that require diverse OCL updating alternatives.

\textbf{Family Structure:} Various versions of the metamodel involved in the case, which describes the family structure, were used as illustrative examples in many MDE publications. We use this case to experiment with metamodel changes that do not require sophisticated updating of the OCL constraints. Over the 9 OCL constraints defined on the initial version of the metamodel, 4 were affected by the changes. The required updating operations consist in 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.

\textbf{State Machine:} We describe the metamodel evolution of this case in Section I. 7 OCL constraints complement the initial version of the metamodel \footnote{https://github.com/atlanmod/LazyOcl_StateMachineExample} and 5 were affected by its evolution \footnote{http://ecariou.perso.univ-pau.fr/contracts/exec-contract.html}. In addition to some basic updating operations, some constraints required to change the way the object types are tested.

\textbf{Project Management:} This case, used in \cite{20}, describes a project management structure. It involves 8 OCL constraints from which 6 were impacted by the evolution. We included this case, because the evolution consisted, in part, of 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.

2) Algorithm Parameters: For NSGA-II, we experimented with 300 iterations, a population of 100 solutions per iteration, a crossover and mutation probabilities of respectively 0.9 and 0.6. For the weight average recommendation, we use the following weights: 40\% respectively for the syntax errors and the information loss, and 20\% for the number of changes. The rational behind these weights is that we consider the syntactic correctness and the information loss as primary objectives whereas, the number of changes is considered on to do not diverge much from the initial constraints. Finally, we varied the number of recommended solutions from 3 to 20.

3) Validation Method: Our algorithm is probabilistic by nature. Thus, to answer \textbf{RQ0}, we performed 30 executions. 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 one). Then, we generated 15050 random solutions in the same way as we do for the creation of the initial population. A Pareto set is then created for the random search using the dominance relationship as defined in Section II.

For each alternative (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 \(S\). For a constraint \(C_i\) of a metamodel \(M\) to be updated and its correct equivalent \(C'_i\) in \(M'\), we say that \(C_i\) is correctly fixed if \(\exists C'_{ij} \in S \mid C_{ij} = C'_{ij}\). Two OCL constraints are equal if the Levenshtein distance is 0 (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 \textbf{RQ1}, we look at the distribution of the number of correctly updated constraints over the 30 executions of our algorithm, inside the Pareto sets. The more often our algorithm is able to find a high number of correctly-updated constraints, the more its results can be used in a recommending process.

To answer \textbf{RQ2}, we will compare two recommendation strategies on their ability to suggest the expected updating solution within a limited number of recommendations. In particular, we will observe on the execution sample the number
of recommendations influences the possibility to obtain a number of correctly fixed constraints equivalent to one if we consider the whole Pareto set (as in RQ1).

B. Results and Interpretation

1) RQ0: is our approach better than a random search?: Table I shows that for the three validation cases, our approach allows to find far more correct solutions than the random search, e.g., for the State Machine case, out of five constraints impacted by evolution, random search finds on average 1.17 correctly-fixed constraints, when our approach correctly update on average 4.53. Similar differences are observed for the two other cases. All differences are statistically significant with a p-value < 0.001. Moreover, 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 [21], an effect greater than 2 is consider as very large.

2) RQ1: how efficient is our approach in finding coevolved OCL constraints?: Table I shows that we were able to find on average four or more correct constraints for the three considered cases (out of four to six impacted constraints). In particular, as indicated in Fig. 10a, we update correctly in the Family case all the constraints in the majority of the executions (25 out of 30), and for the remaining ones, only one constraint is missing. On the second case, [State Machine], all the constraints were correctly updated in 17 out of 30 executions. In 12 executions one constraint was missing, and in one execution two, out of five. In the third case, Project Management, all the executions were able to fix four out of six constraints. That third case is peculiar because in all the executions the same two constraints were missing, whereas in the two first two cases, all the constraints were found in at least one execution. It seems that the two missing constraints are at the edge of our potential and the complexity of the changes involved in the evolution could not be overpassed by our approach. In further investigation, we envision the conception and implementation of new mutation operators which could potentially fix these constraints.

3) RQ2: how efficient are the proposed recommendation technics?: Curves in Fig. 11 show that for both recommendation strategies, the number of correctly updated constraints grows with the number of recommendations. However, we observe that the gain given by a new recommendation decreases considerably after more or less 7 recommendations for the three cases. This gain is almost null after 15 recommendations.

C. Threats to Validity

The goal of our evaluation is to show that without using many inputs required by the state-of-the-art approaches, we are able to obtain good results. However, these results have to be considered in the light of the following threats that may affect their validity.

The first threat is related to the selection of the experimental data. We did our best to select cases that cover a wide range of metamodel changes and OCL updating possibilities. However, more (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 circumvent this threat, we performed 30 executions to compare our algorithm with the random search and to discuss the correctness of the produced results.

We use a syntactic comparison to determine if a candidate constraint is the same as the expected one. This is a threat to the validity of our results as two solutions can be different syntactically, but equivalent semantically. To prevent this threat, we examined a sample of couples of constraints with different Levenshtein distances to ensure that they are actually different. For example, we found a case for the State Machine metamodel where a candidate constraint, with one character difference with the expected one, was not considered as good. When checking this constraint, it appeared that it has the same expression as the expected one with the difference of using 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. This is explained by the fact 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 the existing approach, but rather to obtain comparable results with less effort to produce the inputs.

VII. RELATED WORK

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

Co-evolution has been subject for research since several decades in the database community [22], especially in the field of object-oriented databases [23]. 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 [24]. 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 co-evolution for each change during the metamodel evolution,
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.

Fig. 11: Plots of comparison between syntactical clustering using Levenshtein distance and ranking with Euclidean distance between solutions and optimal on three different metamodel. X-axis shows the number of options given to user, Y-axis the number of constraints fixed.

whereas offline approaches wait after the metamodel has been evolved to perform co-evolution of the OCL constraints.

For online approaches, Demuth et al. [25] [6] provide templates that define a fixed structure for OCL constraint that are then instantiated to update the constraints. However, they are limited to a 11 templates that cannot cover all changes at metamodel level. Hassam et al. [26] [20] 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 [27]. Similarly, Markovic et al. [28] [29] 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. [30] 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] [26] 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.

About automation, Markovic et al. [28] [29], Cabot et al. [7], Demuth et al. [25] [6] are fully automated approaches. Hassam et al. [26] [20], Khelladi et al. [8] and Kusel et al. [30] 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, automate the generation of solution sets, but let 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 co-evolution 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 [31] and to non-deterministic bidirectional transformations [32].

In addition to OCL co-evolution, other research contributions addressed the problem of co-evolving artifacts after metamodel changes. The most important body of work targets metamodel-model co-evolution [33]. Different approaches were proposed such as exploiting the formal relationship between models and their respective metamodels [34], defining coupled operations for metamodels and models [35], or using a multi-objective optimization strategy, similar to the one of this paper [14]. The co-evolution of transformation was also considered, alone [36] or coupled with one-of models [30].

Finally, it is worth mentioning that search-based approaches were extensively used in model-driven engineering [37]. In particular, they were used for metamodel-related artifacts such as generating OCL constraints [38], [39], model selection for metamodel testing [40], [41], and transformation learning [42].

VIII. CONCLUSION

The co-evolution 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 co-evolve metamodels and OCL constraints. First, we pose the co-evolution 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 co-evolution. 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 co-evolution 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

evolution of metamodels and constraints through incremental constraint management,” in International Conference on Model Driven Engineer-
ing Languages and Systems, 2013, pp. 287–303.
nance of ocl constraints,” in International Conference on Software Reuse, 2016, pp. 333–349.


