2021 ACM/IEEE International Conference on Model Driven Engineering Languages and Systems Companion (MODELS-C) | 978-1-6654-2484-4/21/$31.00 ©2021 IEEE | DOI: 10.1109/MODELS-C53483.2021.00126

Metamodel Refactoring using Constraint Solving: a Quality-based Perspective

Oussama Ben Sghaier
Université de Montréal
Montréal, Canada
oussama.ben.sghaier@umontreal.ca

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

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

***Abstract*—The design of metamodels is a main task in model-driven engineering where modellers need to consider many quality factors. However, metamodels are subject to many changes during the software life cycle due to the evolution of requirements or for maintenance purposes. These changes may harm their quality by introducing bad smells that make the metamodels more complex and less understandable. Refactoring metamodels by removing bad smells is not an easy task due to their size, the need to achieve high standards of conflicting quality factors, and the many possible refactoring solutions. We propose a quality-driven approach to refactoring metamodels using constraint solving. We encode both the removal of bad smells and the quality criteria as a set of constraints. Then, we use a constraint solver to find a sequence of refactoring operations that satisfies both constraints. We illustrate the efficiency of our approach through a case study. The latter shows that the refactoring solution we obtain improves the time and correctness of performing understandability and extendibility tasks, as compared to other alternatives.**

***Index Terms*—constraint solving, software quality, refactoring, model-driven engineering.**

I. Introduction

Assisting software practitioners in their tasks is very important for the sake of delivering high software quality and in order to maintain an elevated level of software practitioner's efficiency and productivity [1]–[4]. In particular, the design of metamodels is an important task in the software generation workflow. Metamodels constitute a fundamental artifact in Model-Driven Engineering (MDE). They are the essence of many modelling activities such as language engineering, model transformation, code generation, consistency and conformance validation [5]. Therefore, they should be carefully designed by considering relevant quality factors.

Nevertheless, metamodels are subject to several and continuous modifications related to the evolution and maintenance requirements. These changes may harm the quality of the metamodels by increasing their complexity, decreasing their understandability and extendibility, etc. Since many MDE artifacts depend on metamodels, this eventually negatively impacts productivity, increases fault-proneness and maintenance costs [6]–[8].

To cope with this situation, metamodels need to be regularly maintained through refactoring to improve their quality and to avoid technical debt. An approach to refactor metamodels consists of smells detection and correction [9], [10]. The detection phase consists of identifying bad design decisions. The correction phase consists of removing bad smells using the appropriate refactoring operations. An example of a bad smell is "dead class" which consists of a class disconnected from the rest of the design [11]. The corresponding refactoring operation entails removing the dead class.

Metamodel maintenance is very important but also challenging, as it requires a lot of effort and should take care of many quality criteria, such as understandability, extendibility, and maintainability. Improving the metamodel quality is a complex task since quality factors are conflicting [12], [13]. For instance, improving the extendibility of the metamodel by introducing some super-classes containing the duplicated features may harm the understandability of the design. Therefore, refactoring metamodels should be driven by the improvement of quality and should find the best compromise between the different quality factors as well as the number of smells to be removed.

Many approaches were proposed for metamodels or models refactoring. These approaches are based on formal methods (e.g., [14]), model transformations (e.g., [15]), or a learning process from preexisting examples (e.g., [16], [17]). These proposed methods use different techniques to detect refactoring opportunities without worrying about the quality factor which is the main goal of refactoring. The refactoring operation should not be performed haphazardly but rather be based on well-defined objectives, such as improving certain quality criteria.

Bettini et al. presented in [11] a quality-driven framework for detecting and resolving metamodel smells. They define a static mapping between design smells and quality attributes. Then, refactoring a metamodel consists of removing all the bad smells that have an impact on the target quality attributes. Nevertheless, removing all design smells from a metamodel is not necessarily the best solution, as it might not lead to the best overall values of quality attributes. In fact, it is known that quality attributes are potentially conflicting [12], [13]. Improving one quality criterion could lead to the degradation of another. For example, introducing too many elements to a metamodel to improve its flexibility and extendibility may harm its understandability. Thus, a refactoring solution should find the best trade-off with respect to the target quality attributes.

978-1-6654-2484-4/21/$31.00 ©2021 IEEE
DOI 10.1109/MODELS-C53483.2021.00126

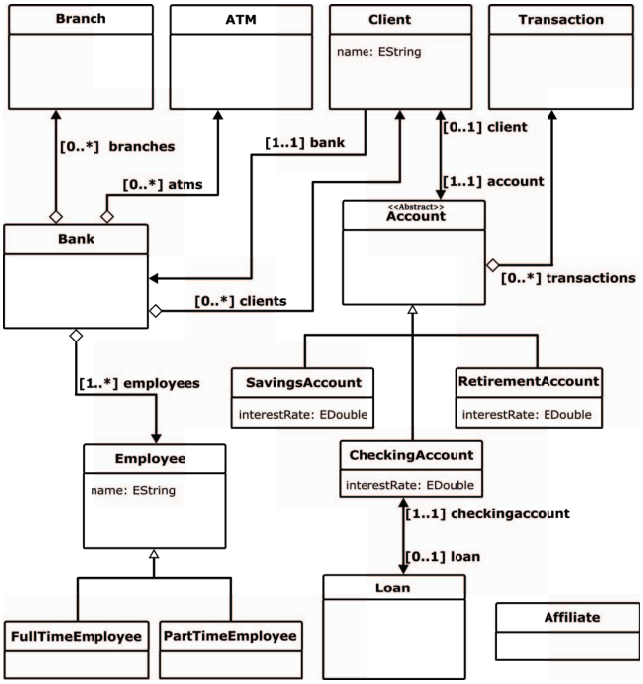

Fig. 1. Metamodel for bank management systems

In this paper, we propose a quality-driven approach based on constraint solving to refactor metamodels. Our approach recommends refactoring solutions obtained by a trade-off between quality factors and smell removal. We illustrate the efficiency of our approach with a case study of a Customer Relationship Management (CRM) metamodel. We measure the time required to perform understandability and extendibility tasks on this metamodel. We also evaluate the correctness of the task output. The results show that our approach improves the time and correctness of performing extendibility and understandability tasks and generates relevant refactoring solutions that improve these quality factors of the metamodel.

The remainder of this paper is organized as follows: We introduce a running example in Section II. Our approach is described in Section III. Section IV is dedicated to an illustrative case study. Finally, we outline the related work in Section V and conclude in Section VI.

II. Running example

We present the design of a simplified metamodel for bank management systems as a running example that we will use later to illustrate our approach.

As shown in Fig. 1, the `Bank` is the central class that is composed of automated teller machines (`ATMs`) and `Branches`. It employs `FullTime` or `PartTime Employees`. The bank has `Clients` who hold different types of `Accounts`: `Checking`, `Savings` or `Retirement`. An account logs a list of `transactions`. A checking account can have a `Loan` attached to it. Finally, the class `Affiliate` is used to store information about employees' affiliation to specific branches.

This metamodel contains many types of design smells [18], which are the result of poor design decisions. We briefly describe the following four smells.

Dead metaclass is a design smell identifying a class not related to any other class in the metamodel, like `Affiliate`. This is analogous to *dead code* which is defined as a fragment of code that is no longer used [19]. This design smell may have a negative impact on the quality of the metamodel because it reduces its understandability and makes it more complex by introducing unusable and useless elements in the metamodel. Refactoring this smell consists of simply removing the class.

Classification by enumeration or by hierarchy is a design smell that concerns cases where we should use enumeration instead of an inheritance hierarchy of classes [18]. For instance, an `Employee` can be classified as a `FullTimeEmployee` or `PartTimeEmployee`. Depending on the use case, this could be considered as a design smell since the two sub-metaclasses do not add any new features. Alternatively, it could be more appropriate to represent the employment type as an enumeration inside `Employee` with these two values.

Concrete abstract metaclass is a design smell where super-classes are concrete instead of abstract. For instance, `Employee` should not be instantiated because an employee must be one of the two subtypes. To refactor this smell, we make the class abstract [11].

Duplicated features is another common smell, where the same feature is duplicated in several classes [18]. For example, some of the attributes are repeated in more than one class (e.g., `name` in `Employee` and `Client`). This design smell can be resolved by applying a *pull-up attribute* refactoring, if these classes have a common super-class (e.g., `interestRate` in the different subclasses of `Account`); otherwise, we should *create a super-class* that unifies the duplicated features (e.g., for the attribute `name`, we could create a class `Person` and have `Client` and `Employee` inherit from it).

III. Proposed approach

A. Overview

We propose a formal approach for improving metamodel quality via refactoring that relies on constraint solving as the reasoning back-end. The main benefit of constraint solving is that it allows us to better model the different quality concerns as constraints, as well as to reason about the different compromises and trade-offs between the constraints. To explore the feasibility of such an approach, we use Alloy [20] as a prototyping specification language to express as constraints: (a) the quality criteria and (b) metamodel smells. Alloy is a formal constraint specification language based on first-order relational logic. It provides a lightweight modelling tool to express and check properties of system specifications by performing model checking within a finite bound. This makes it well suited for rapid prototype development and allows us to explore the feasibility of our constraint solving based approach.

Having modelled quality criteria and smells as Alloy constraints, we use Alloy's model finding capabilities to try to

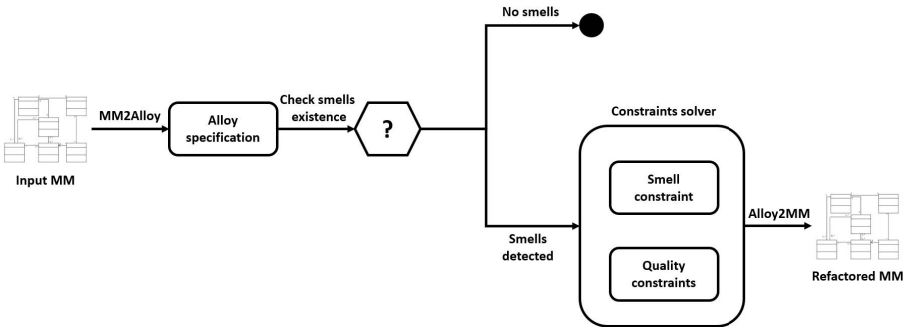

Fig. 2. Constraint solving approach for quality-driven refactoring of metamodels.

produce refactored versions of the given metamodel that satisfy these constraints. Alloy returns a set of candidate solutions which we interpret as recommendations for improving the quality of the input metamodel. We show schematically our approach in Fig. 2. It has two parts: (1) a detection phase in which we check for the presence of design smells (2) a refactoring phase where we remove the bad smells while satisfying quality constraints.

First, we translate the input metamodel to an Alloy specification using a model transformation. We then encode the absence of design smells as constraints, which we check with the Alloy Analyzer. If smells exist in the metamodel, we execute the refactoring phase. We encode the removal of smells and the quality criteria as additional logical constraints, which we combine with the encoding of the input metamodel. We then use Alloy to generate refactored versions of the metamodel. By construction, these satisfy the encoded quality and absence-of-smells constraints. In the following, we present our approach in detail.

```
1  package CD2Alloy;
2  classdiagram Bank_CD {

4    class Branch;
5    class ATM;
6    class Client {string name;}
7    class Transaction;
8    class Bank;
9    abstract class Account;
10   class SavingsAccount extends Account {
11     double interestRate;}
12   class CheckingAccount extends Account {
13     double interestRate;}
14   class RetirementAccount extends Account {
15     double interestRate;}
16                          ...
17   composition composition1
18     [1] Bank (bank) -> (branches) Branch [*];
19   composition composition2
20     [1] Bank (bank) -> (atms) ATM [*];
21   association association1
22     [1] Client (client) -> (bank) Bank [1];
23   association association2
24     [0..1] Client (client) -- (account) Account [1];
25                          ...
26 }
```

Listing 1. An excerpt from the class diagram representation of the Bank metamodel

B. MM2Alloy transformation

The initial phase is to transform the input metamodel class diagram into an Alloy specification. We adapted the *CD2Alloy* transformation by Maoz et al. [21], which translates UML class diagrams to Alloy and provides tool support as an eclipse plugin. Our variant, *MM2Alloy*, explicitly encodes metamodel concepts such as the extensions, associations, and their multiplicity and type.

The *CD2Alloy* plugin requires encoding the input class diagram in a textual format. We show an excerpt of the encoding of the bank management system metamodel (Fig. 1) in this format in Listing 1. We also show the Alloy encoding that is generated by *MM2Alloy* for this input in Listing 2.

A metamodel is represented as a class diagram (*CD*) object (lines 33-38). It is defined by its *classes*, *features* (i.e., attributes), *associations* and *extensions* (i.e., inheritance relations). We translate classes to Alloy signatures that inherit a common top-level signature named *Obj* (line 8). For example, in line 17 of Listing 2, the class `Branch` is defined as a signature extending `Obj`. The names of *attributes* and *associations* inherit from a common signature *FName* (lines 4 and 11). Primitive types of attributes inherit from the signature *Val* (line 6).

In line 35, *associations* are defined as a set of binary relations between *associationEnd* objects, defined in lines 25-31. An *associationEnd* is defined by its class, type (e.g., *composition*, *unidirectional* or *bidirectional*) and its arity represented by its lower and upper bounds. Individual associations are encoded using the *buildAsso* predicate (lines 40-47). For example, line 58 shows the encoding of the composition association between the classes `Bank` and `Branch`, along with the relevant bounds.

Inheritance relationships (*Extensions*) are defined in line 36 as a set of binary relationships between classes. In other words, $\langle class1, class2 \rangle \in CD.extensions$ means that *class1* inherits *class2*. For example, in line 64, we specify that `SavingsAccount` inherits from `Account`.

Attributes are represented as a ternary relationship between classes, names and types (line 37). Individual attributes are encoded using the *buildFeature* predicate (lines 48-50). For example, in line 67, the class `Client` is as-

```
1  module Bank_CD

3  //Names of fields/associations in classes of the model
4  abstract sig FName {}
5  //Types of fields
6  abstract sig Val {}
7  //Parent of all classes
8  abstract sig Obj {}

10 //Names of fields/associations in cd
11 one sig name extends FName {}
12 one sig interestRate extends FName {}
13 //Types in model cd
14 one sig string extends Val {}
15 one sig double extends Val {}
16 //Classes in model cd
17 one sig Branch extends Obj {}
18 one sig ATM extends Obj {}
19 one sig Client extends Obj {}
20 one sig Transaction extends Obj {}
21 one sig Bank extends Obj {}
22 one sig Account extends Obj {}
23 one sig SavingsAccount extends Obj {}

25 sig associationEnd{
26   class: Obj,
27   type: String,
28   label: FName,
29   lowerBound: Int ,
30   upperBound: Int ,
31 }
32 //Metamodel representation
33 sig CD{
34   classes: set Obj,
35   associations: set associationEnd -> associationEnd,
36   extensions: classes -> classes,
37   features: Obj -> FName -> Val
38 }

40 pred buildAsso[cd:CD, obj1:Obj, label1:FName, _type:
       ↪ String, obj2:Obj, label2:FName, l1:Int, u1:Int,
       ↪ l2:Int, u2:Int]{
41   one ae1,ae2:associationEnd |{
42     ae1≠ae2
43     ae1.class =obj1 and ae1.type=_type and ae1.label=
         ↪ label1 and ae1.lowerBound=l1 and ae1.
         ↪ upperBound=u1
44     ae2.class =obj2 and ae2.type=_type and ae2.label=
         ↪ label2 and ae2.lowerBound=l2 and ae2.
         ↪ upperBound=u2
45     ae1->ae2 in cd.associations
46   }
47 }
48 pred buildFeature[cd:CD, c: Obj, fn: FName, v: Val]{
49   c->fn->v in cd.features
50 }

52 //Build the input metamodel
53 pred initialConditions[cd:CD]{
54   //Classes
55   cd.classes ={Branch + ATM + Client + Transaction +
       ↪ Bank + Account + SavingsAccount + . . .}
56   //Associations
57   #cd.associations=8
58   buildAsso[cd,Bank,bank,"composition",Branch,branches
       ↪ ,1,1,0,-1]
59   buildAsso[cd,Client,client,"unidirectional",Bank,bank
       ↪ ,1,1,1,1]
60   buildAsso[cd,Client,client,"bidirectional",Account,
       ↪ account,0,1,1,1]
61     . . .
62   //Extensions
63   #cd.extensions=5
64   cd.extensions ={SavingsAccount->Account +
       ↪ CheckingAccount->Account + . . .}
65   //Features
66   #cd.features=5
67   buildFeature[cd, Client, name, string]
68   buildFeature[cd, SavingsAccount, interestRate, double
       ↪ ]
69   . . .
70 }
```

Listing 2. Specification of the Bank metamodel in Alloy

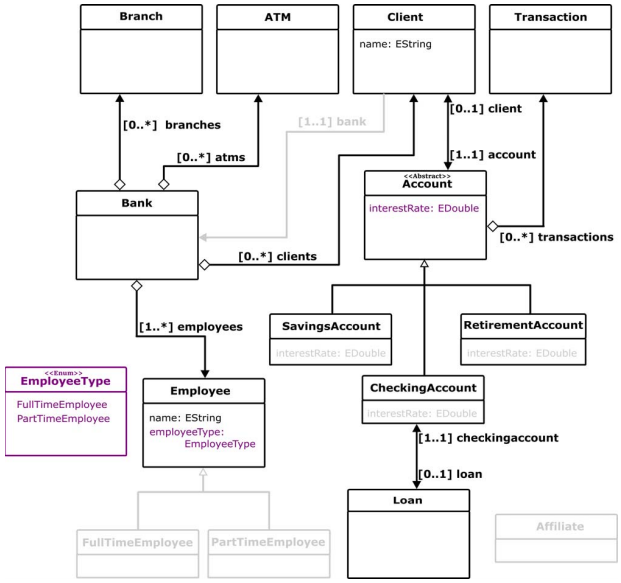

Fig. 3. Refactoring solution generated using our approach for the bank management system metamodel. Added elements are colored in purple and deleted elements are in grey. Four refactorings were applied on the initial version of the metamodel (Fig. 1): (1) pull-up feature *interestRate* to *Account* class (2) remove dead metaclass *Affiliate* (3) transform the classification by hierarchy to a classification by enumeration for the sub-classes of *Employee*, a new enumeration were introduced *EmployeeType* (4) Remove the unidirectional association between *Client* and *Bank* as it is implicitly contained in the composition relation.

signed the attribute `name:String`. In Alloy, this means that $\langle Client, name, string\rangle \in CD.features$.

A given input metamodel is defined using the predicate *initialConditions* (lines 53-70). This predicate initializes all the concrete elements of the input metamodel by encoding them in the Alloy representation described above: classes (line 55), associations (lines 57-61), inheritance relationships (lines 62-64), and attributes (lines 65-69).

Alloy performs bounded scope analysis by checking the encoded specification over a finite number of instances. To visualize the specification in Listing 2, we run the predicate *initialConditions* for a scope of 1 atom for the signature *CD*, 15 atoms for *Obj*, and 9 atoms for all other signatures.

C. Detection of bad smells

In this phase, we aim to identify bad smells, which can be indicators of poor design decisions. Our prototype supports the detection of the 4 bad smells described in Section II: Dead metaclass, Classificiation by enumeration or by hierarchy, Concrete abstract metaclass, and Duplicated features. We encode each smell as a predicate in Alloy. For example, Listing 3 shows our encoding of the Dead metaclass smell. A "dead" metaclass is one that is disconnected from the rest of the model, i.e., it does not have associations or inheritance relations. Our encoding of the smell therefore has two parts: in lines 6-12, we define how to check each individual class; in lines 1-4, we define that for the smell to exist in a

```
//check if there is a dead metaclass in the class diagram
pred deadClass[cd: CD] {
  some c: cd.classes |isDeadClass[cd, c]
}
//check if a metaclass is a 'dead metaclass'
pred isDeadClass[cd: CD, o:Obj] {
   //check if the class o does not have associations
  all x,y :associationEnd |(x->y in cd.associations)
       implies (not (o in x.class) and (not (o in y.
       class)))
   //check if the class o does not have inheritance
       relations
  all p:cd.classes |not o->p in cd.extensions and not p
       ->o in cd.extensions
}
```

Listing 3. Detection of dead metaclass in Alloy

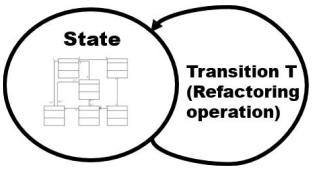

Fig. 4. Refactoring process: An ordered finite state machine

metamodel, there must exist at least one class for which the check succeeds. In case a smell exists, Alloy produces an example that allows to localize it in the metamodel. For example, running the predicate for the metamodel of the running example, will result in the Alloy Analyzer adding to the class `Affiliate` the annotation `$deadClass_c` allowing us to localize the smell.

D. Quality-driven metamodel refactoring

Each design smell is associated with corresponding refactoring operations that remove it. Our prototype supports the refactoring operations presented in Section II: removing a dead metaclass, introducing an enumeration instead of a hierarchy, making a concrete superclass abstract, pulling-up duplicated features, and introducing superclassess for duplicated features.

If we detect some bad smells in the first phase of our approach, in the second phase, we refactor the metamodel to improve its quality. In our approach, the refactoring process is driven by some quality criteria (i.e., extendibility, understandability, maintainability.) according to the interests of the modeller.

We represent the refactoring process as an ordered finite state machine, as shown in Fig. 4 and Fig. 5. Each state represents a version of the metamodel, where the initial state is the input metamodel and the last state is the output refactored metamodel. Each transition represents the application of a refactoring operation at the metamodel of the transition's source state which results in the metamodel of the transition's target state.

Our approach is to use a constraint solver to find a sequence of transitions (i.e., refactoring operations) where the last state will contain the refactored metamodel that satisfies the quality constraints given as input. We will then output the resulting

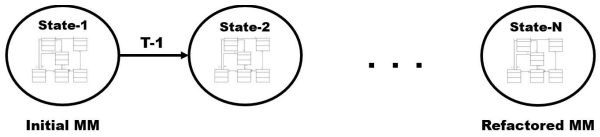

Fig. 5. Flattened view of the refactoring process

```
1  open util/ordering[State]
2
3  sig State {
4    cd: one CD
5  }
6  fact {
7    all s: State, s': s.next {
8       s' in State implies
9         refactoringOperation [s.cd, s'.cd]
10 }}
11 pred refactorOperation[cd, cd': CD] {
12   removeDeadClass[cd, cd'] or
13   pullUpField[cd, cd'] or
14   ExtractSuperClass[cd, cd'] or
15   . . .
16 }
17 pred removeDeadClass[cd, cd': CD]{
18   one x: cd.classes |
19      isDeadClass[cd,x]
20      cd'.classes =cd.classes - x
21      cd'.associations =cd.associations
22      cd'.extensions =cd.extensions
23      all fn: FName, v: Val, c: Obj |
24         c=x implies
25            not (c->fn->v in cd'.features)
26       else c->fn->v in cd.features implies
27          c->fn->v in cd'.features
28       else not c->fn->v in cd'.features
29 }
```

Listing 4. Refactoring specification in Alloy

metamodel as a recommendation to the modeller. The modeller can accept it or request additional recommendations, i.e., the additional candidate solutions produced by the constraint solver.

We show our prototype implementation with Alloy in Listing 4. It defines the refactoring process as an ordered finite state machine, using the Alloy module `util/ordering` to impose an ordering on the *State* signature. Defined in lines 3-5, a *State* contains a single *CD* object, i.e., a metamodel class diagram. In lines 7-19, we mandate that the metamodels of two consecutive states should be related by a *refactorOperation*, which can be any of the known refactorings. As an example, we show the implementation of the *removeDeadClass* refactoring in lines 21-33. It consists of removing the dead metaclass from the metamodel in the next state (line 23), while also removing its attributes (lines 27-33). All other associations and inheritance relations are preserved (lines 24-26) in the resulting metamodel.

Next, we encode the requirement that the quality of the resulting metamodel should be improved with respect to the input as additional quality constraints. In our Alloy prototype, we used three quality characteristics: *maintainability*, *understandability*, and *extendibility*. Based on prior work on software quality [22]–[24], we operationalize these quality characteristics using the design metrics listed in Table I.

TABLE I
DESIGN METRICS USED TO DEFINE QUALITY ATTRIBUTES

Metric	Description
NC	Number of classes
NA	Number of attributes
NR	Number of references
DIT_{max}	Maximum hierarchical level i.e., depth of the inheritance tree
$Fanout_{max}$	Maximum Fanout, i.e., maximum number of referenced classes by a class
$PRED_c$	Number of predecessors in the inheritance hierarchy of a given class c
INHF	Number of inherited features
NTF	Total number of features

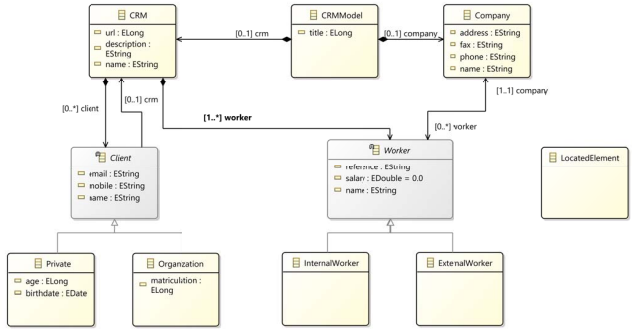

Fig. 6. Initial version of the CRM metamodel

Genero and Piattini [22] define maintainability as the average of low level design metrics (Table I) as follows:

$$\frac{NC + NA + NR + DIT_{max} + Fanout_{max}}{5} \quad (1)$$

Lower values indicate better maintainability.

Sheldon and Chung [23] define understandability as the average number of predecessors in the hierarchy of inheritance:

$$\frac{\sum_{c=1}^{NC} PRED_c + 1}{NC} \quad (2)$$

Higher values indicate a more understandable design.

Arendt et al. [24] introduced the *Attribute Inheritance Factor* (*AIF*) as a measure of extendibility, defined as:

$$\frac{INHF}{NTF} \quad (3)$$

Higher values of this factor indicate a higher degree of extendibility.

We show our prototype Alloy implementation for computing quality attributes in Listing 5. We first calculate the low level design metrics from Table I (lines 1-32). These are then used to compute a value for the three quality attributes: *understandability*, *extendibility* and *maintainability* (lines 34-43).

The goal of our approach is to improve the quality attributes of the input metamodel. We encode the concept of quality improvement using the predicate *QA* in lines 46-53 of Listing 5. We then use this predicate to require that the output metamodel has better quality than the input (line 62).

Finally, we use the predicate *smellsConstraint* (lines 55-57) to define a minimum number of smells that the solver should try to remove. This is easily encoded as the minimum number of *State* atoms that Alloy should try to instantiate.

In lines 59-64, we show an example execution setup using the *executionExample* predicate. In this example, we require that at least 4 smells are removed, while improving the *Extendibility* and *Maintainability*.

Executing this example in the given scope (line 64) for the running example results in Alloy producing some refactoring solutions that satisfy the specified constraints. Among these solutions, it produces a refactored metamodel in four transitions, as shown in Fig. 3, corresponding to the following four refactoring operations: remove dead metaclass *Affiliate*, pull-up duplicated feature *interestRate*, classification by enumeration for the sub-classes of *Employee* and remove the unidirectional association between *Client* and *Bank*. This refactored metamodel can be presented as a recommendation to the modeller. The Alloy Analyzer can also produce additional recommendations as additional instances.

The complete Alloy listing for this example is available online[1] for replication.

IV. ILLUSTRATIVE CASE STUDY

In this section, we illustrate the usefulness of our approach through a case study on the *Customer Relationship Management* (*CRM*) metamodel shown in Fig. 6. We perform a user study to assess the effectiveness of our approach when considering two quality factors (extendibility and understandability) in the refactoring process.

A. Research questions

To evaluate our approach, we compared the metamodel version obtained by our approach (trade-off between smell removal and quality factors) with two alternative approaches as baselines: (a) the initial metamodel, and (b) the metamodel obtained by applying the approach developed by Bettini et al. [11]. This approach is designed to produce a metamodel where all smells are removed, regardless of quality considerations. We formulate two research questions:

- **RQ1:** *How does our approach perform in practice compared to the baseline alternatives with respect to independent user assessment of understandability?*
- **RQ2:** *How does our approach perform in practice compared to the baseline alternatives with respect to independent user assessment of extendibility?*

B. Setup

1) Studied metamodel: In our study, we used the *CRM* metamodel shown in Fig. 6, which has a number of bad smells. Specifically: (a) `LocatedElement` is a *dead metaclass*. (b) `InternalWorker` and `ExternalWorker` are two classes

[1]https://github.com/OussamaSghaier/SAM2021/

```
// number of classes
fun NC[cd: CD]: one Int {
   {ans:Int |ans =#cd.classes}
}
// number of references
fun NR[cd: CD]: one Int {
   {ans:Int |ans =#cd.associations}
}
// number of composition relations
fun NCR[cd: CD]: one Int {
   {ans:Int |ans =#{l,r: associationEnd |l->r in
      cd.associations and
      (l.type ="composition" or
         r.type ="composition")
}}}
// number of unidirectional relations
fun NUR[cd: CD]: one Int {
   {ans:Int |ans =#{l,r: associationEnd |
      l->r in cd.associations and
      ( l.type="unidirectional" or
         r.type="unidirectional")
}}}
// number of attributes
fun NA[cd: CD]: one Int {
   {ans:Int |ans =#cd.features}
}
// number of generalizations
fun NGenH[cd: CD]: one Int{
   {ans:Int |ans =#cd.extensions}
}

...

fun Maintainability[cd: CD]: one Int{{
   ans:Int |
   ans=div[sum{NC[cd]+NA[cd]+NR[cd]+DITmax[cd]+
        ↪ FANOUTmax[cd]},5]
}}

fun Understandability[cd: CD]: one Int{{
     ans:Int |ans =div[int PRED[cd] + int 1 , NC[cd] ]
}}

...

// Quality assurance predicate
pred QA[cd, cd': CD, q: set String]{
  "Maintainability" in q implies
     gte[Maintainability[cd],Maintainability[cd']]
  "Understandability" in q implies
     lte[Understandability[cd],Understandability[cd']]
  "Extendibility" in q implies
     lte[Extendibility[cd],Extendibility[cd']]
}
// predicate for the number of removed smells
pred smellsConstraint[threshold: Int]{
  #State ≥int threshold + 1
}
// example of execution
pred executionExample{
  initialConditions[first.cd]
  smellsConstraint[4]
  QA[first.cd,last.cd,{"Extendibility"+"Maintainability
       ↪ "}]
}
run executionExample for 9 but 15 Obj
```

Listing 5. Excerpt from Alloy code for computing quality

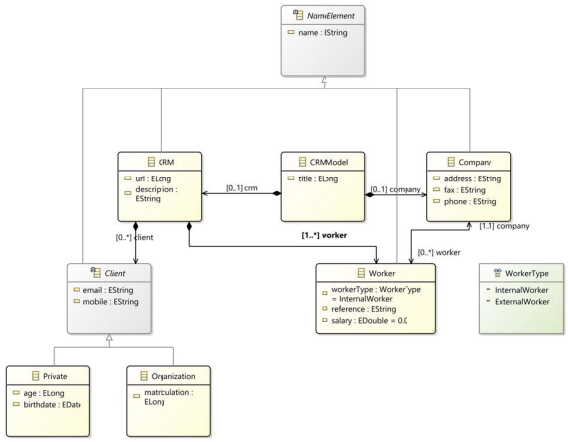

Fig. 7. CRM metamodel with all smells removed

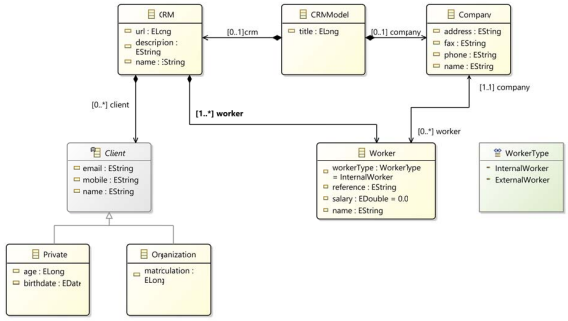

Fig. 8. CRM metamodel generated using our approach

that inherit `Worker` but do not add any new features. (c) `Name` is a *duplicated feature* in many classes: `Client`, `CRM`, `Company`, `Worker`. (d) There is a redundant unidirectional relation between `Client` and `CRM` that is implicitly contained in the composition relation.

We show the initial version of the CRM metamodel, that includes all the smells in Fig. 6. Fig. 7 shows the CRM metamodel with all smells removed, following the approach by Bettini et al. [11]. Fig. 8 shows a solution generated using our quality-driven approach where we focus on improving *understandability* and *extendibility* while removing at least two bad smells.

2) User study: The metamodels produced by our approach are the result of a trade-off between (a) removing a maximum number of smells and (b) maintaining or improving the values of a given set of quality attributes. The encoding of quality constraints is done according to a theoretical quality model based on published scientific literature, as described in Section III. To assess in practice the quality of the produced metamodel, we conducted a user study with nine participants familiar with MDE concepts and metamodelling in particular.

We consider three versions of this metamodel: (1) "*Initial*", i.e., without smell removal, (2) "*ALL*", i.e., with all smells removed, and (3) "*CS*", i.e., using our constraint-solving

TABLE II
QUESTIONS USED IN THE USER STUDY

Quality attribute	Question
Understandability	1/ Who are the users of the CRM system (A Customer Relationship Management system allows managing company's relationships and interactions with customers and potential customers.)? 2/ What is the profile information of each user?
Extendibility	1/ We want to include temporary workers to the CRM metamodel. What are the necessary modifications? 2/ We want to extend the metamodel so that workers could supply services to clients who are able to book appointments with workers. An appointment has a time and a description. What are the required changes?

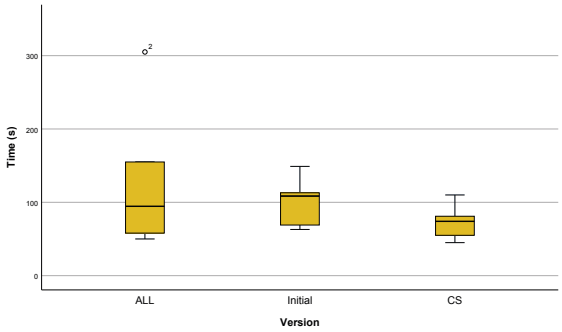

Fig. 9. Response time in terms of understandability

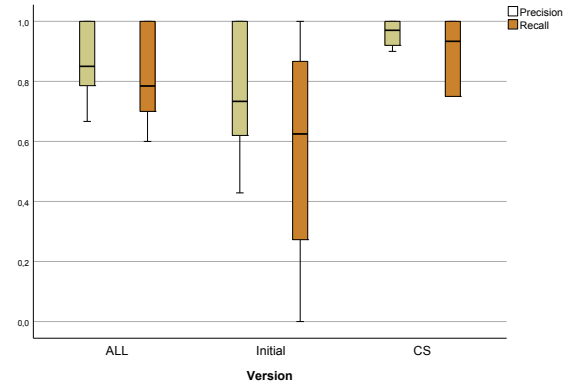

Fig. 10. Correctness of the results in terms of understandability

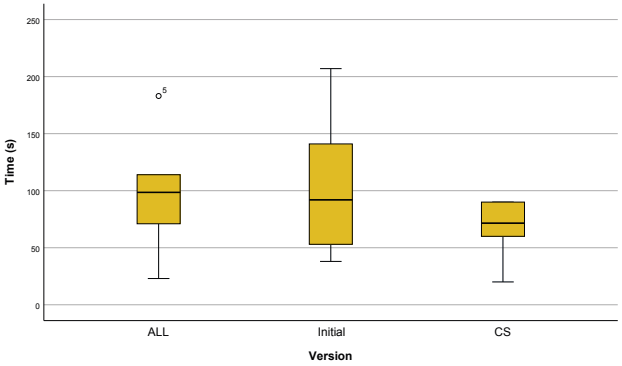

Fig. 11. Response time in terms of extendibility

approach. We divided the participants into 3 groups of 3, and assigned to each group a different metamodel version.

We designed two questions per quality attribute with different levels of difficulty. For **RQ1**, we asked the participants to answer the questions using information encoded in the metamodel to evaluate their comprehension level of the metamodel. For **RQ2**, we asked the participants, through questions, to extend the metamodel by performing the necessary changes to satisfy some given requirements. Table II shows the questions used in this user study.

Each evaluation session was done as a 30 minute Zoom meeting with one participant at a time. During a session, we presented to each participant a version of the metamodel; then we asked the questions. For each question, we recorded the time the participant took to answer, as well as their response.

C. Results

RQ1: In Fig. 9 we show the time that participants took to answer questions related to understandability. They took less time to answer while using the metamodel generated by our approach (CS) compared to the other versions. As a baseline, the median of the time taken by participants to answer understandability-related questions is 109 seconds for the *Initial*, 95 seconds for *ALL* and 74 seconds for the *CS* variant generated using our approach. There is also less dispersion in the values of time for the CS case as the three participants have close time values. An interesting observation is that removing all smells resulted in more effort from the participants to understand the metamodel, because of the addition of the abstract class `NameElement` and the inheritance lookup to perform by the participants.

We show the correctness results of the participants' responses to understandability-related questions in Fig. 10. Participants tended to answer correctly more often for the metamodel generated with our approach (CS) ($precision = 0.97$ and $recall = 0.93$) compared to the *ALL* ($precision = 0.85$ and $recall = 0.78$) and *Initial* ($precision = 0.73$ and $recall = 0.62$) metamodel variants. This is evident in the better precision and recall in terms of median and the fact that there is less variability in values.

Overall, participants had a higher comprehension level of the *CRM* metamodel when shown the refactored version produced using our approach, compared to other alternatives. However, they had some difficulties answering correctly the understandability-related questions on the initial metamodel due to the presence of smells.

RQ2: We show time measurements for performing extendibility tasks in Fig. 11. Overall, participants took less time to extend the *CRM* metamodel when using our version (*CS*)

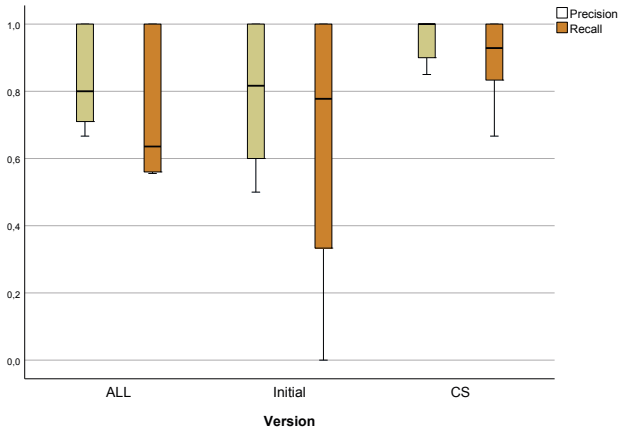

Fig. 12. Correctness of the results in terms of extendibility

compared to the *ALL* and *Initial* versions. Participants took 92 seconds on performing extendibility tasks when using the initial metamodel, 99 seconds for the *ALL* metamodel variant and 72 seconds for the metamodel generated using our approach.

We show the precision and recall on performing extendibility-related tasks on the different versions of the *CRM* metamodel in Fig. 10. Using the refactored metamodel generated by our approach allows to improve the correctness of performing extendibility tasks ($precision = 1$ and $recall = 0.92$) compared to other alternatives ($precision = 0.77$, $recall = 0.81$ for *Initial* and $precision = 0.80$, $recall = 0.63$ for *ALL*). Removing all smells (i.e., ALL) has worsened the extendibility of the *CRM* metamodel in terms of median, compared to *Initial* version, but it reduced the variability of the correctness measures.

Our approach was able to improve the time and correctness (i.e., precision and recall) of performing extendibility and understandability tasks, on the *CRM* metamodel, compared to other variants (i.e., *Initial* and *ALL*).

D. Threats to validity

The case study presented here is not intended to rigorously validate our approach, but to illustrate its efficiency on a concrete example. Still, the results obtained must be considered in the light of some decisions that may limit their validity.

Construct validity involves the quality model (formulas) used by our approach. The results obtained are dependent on these quality formulas. Other, more sophisticated, quality models may be used. Our approach can be adapted to support such quality models as the involved quality factors can be encoded as constraints.

An internal threat to the validity of the case study relates to the knowledge variance between subjects, as we used a Between-subjects design, i.e., each participant is assigned to a unique version of the metamodel. To mitigate this, we tried to balance the groups in terms of level of knowledge and experience when assigning participants to metamodel versions.

Finally, an external threat to the validity concerns the generalizability of our findings to other cases. To fully validate our approach, we are designing a more comprehensive study involving more subjects and a large sample of metamodels.

V. Related work

In the literature, many works have investigated the automation of model and metamodel refactoring. These approaches are based on different techniques such as formal methods (e.g., [14]), model transformations (e.g., [15]), or learning process from preexisting examples (e.g., [25]).

The first family of work targets the refactoring by means of smell detection and correction. A representative example is EMF Refactor tool [26], [27]. It supports metrics reporting, smells detection and resolution for models based on Eclipse Modeling Framework (EMF) [28].

Other approaches consist of deriving endogenous and in-place model transformations. Reimann et al. [29] present a generic refactoring framework based on EMF to model refactorings for different modelling and meta-modelling languages. These generic refactorings can be reused for different languages only by providing a mapping. Based on the defined mapping, a generic transformation is executed to restructure the models. In [30], [31], the authors derive model transformations by analyzing user editing actions when refactoring models.

Several approaches were proposed to learn model transformations from examples for, among others, refactoring. In [17], the authors use a search-based approach to generate the best sequence of refactorings based on textual and structural similarity with a set of provided examples. In [32], the authors propose a process to learn complex model transformations by considering three common requirements: element context and state dependencies and complex value derivation. A similar work was proposed in [16] that relies on genetic programming to learn model transformation rules guided by the conformance with the provided examples. Kessentini et al. [25] use a set of transformation examples to derive a target model from a source model. The authors explore different transformation possibilities evaluated based on their conformance with the examples at hand.

Related to our work, Gheyi et al. [14] present a constraint-based approach to refactor models. The semantics of Unified Modeling Language (UML) were defined as well-formedness rules to guide the refactoring process for a set of connected models. The authors check these well-formedness rules on transformed models. If not satisfied, they solve the failed constraints by computing additional model changes required to have a valid and semantic-preserving refactoring transformation.

The above-mentioned research contributions do not consider explicitly the quality as a main concern to define the refactoring.

Bettini et al. [11] propose a quality-driven framework for detecting and resolving metamodel smells. The authors define static mapping between bad smells and quality attributes

by associating each smell to the set of quality attributes it affects. Based on quality requirements, the associated smells are identified and removed using refactoring operations. In this work, the quality is considered independently for each smell. Conversely, in our work, we seek to reach a trade-off between the quality constraints expressed by the developer and the smell removal.

VI. Conclusion

We proposed a constraint solving approach to refactoring metamodels. It is driven by the improvement of metamodel quality while maximizing the number of removed smells. We illustrated its effectiveness with a case study on the *CRM* metamodel. The results show that our approach improves the extendibility and understandability of the metamodel compared to other alternatives.

Our Alloy prototype demonstrated the feasibility of using a constraint solver to reason about quality improvement and refactoring. However, encoding quality metrics as constraints in the default relational first-order logic engine used by Alloy is not efficient as it provides only rudimentary arithmetic capabilities. In the future, we will investigate using recent advances in combining Alloy with SMT-based reasoning [33]. We intend also to perform an extensive evaluation with more metamodels, subjects and quality attributes. This would allow us to confirm our findings on the studied case.

References

[1] A. Hernández-López, R. Colomo-Palacios, and Á. García-Crespo, "Productivity in software engineering: A study of its meanings for practitioners: Understanding the concept under their standpoint," in *7th Iberian Conference on Information Systems and Technologies (CISTI 2012)*. IEEE, 2012, pp. 1–6.

[2] B. Kitchenham and E. Mendes, "Software productivity measurement using multiple size measures," *IEEE Transactions on Software Engineering*, vol. 30, no. 12, pp. 1023–1035, 2004.

[3] R. C. King, W. Xia, J. C. Quick, and V. Sethi, "Socialization and organizational outcomes of information technology professionals," *Career Development International*, 2005.

[4] A. Hernández-López, R. Colomo-Palacios, Á. García-Crespo, and F. Cabezas-Isla, "Software engineering productivity: Concepts, issues and challenges," *International Journal of Information Technology Project Management (IJITPM)*, vol. 2, no. 1, pp. 37–47, 2011.

[5] M. Brambilla, J. Cabot, and M. Wimmer, "Model-driven software engineering in practice," *Synthesis lectures on software engineering*, vol. 3, no. 1, pp. 1–207, 2017.

[6] F. Khomh, M. Di Penta, Y.-G. Guéhéneuc, and G. Antoniol, "An exploratory study of the impact of antipatterns on class change-and fault-proneness," *Empirical Software Engineering*, vol. 17, no. 3, pp. 243–275, 2012.

[7] M. Abbes, F. Khomh, Y.-G. Gueheneuc, and G. Antoniol, "An empirical study of the impact of two antipatterns, blob and spaghetti code, on program comprehension," in *2011 15Th european conference on software maintenance and reengineering*. IEEE, 2011, pp. 181–190.

[8] A. Yamashita and L. Moonen, "Exploring the impact of inter-smell relations on software maintainability: An empirical study," in *2013 35th International Conference on Software Engineering (ICSE)*. IEEE, 2013, pp. 682–691.

[9] M. Mohamed, M. Romdhani, and K. Ghédira, "Classification of model refactoring approaches," *Journal of Object Technology*, vol. 8, no. 6, pp. 121–126, 2009.

[10] H. Mumtaz, M. Alshayeb, S. Mahmood, and M. Niazi, "A survey on uml model smells detection techniques for software refactoring," *Journal of Software: Evolution and Process*, vol. 31, no. 3, p. e2154, 2019.

[11] L. Bettini, D. Di Ruscio, L. Iovino, and A. Pierantonio, "Quality-driven detection and resolution of metamodel smells," *IEEE Access*, vol. 7, pp. 16 364–16 376, 2019.

[12] B. Andreopoulos, "Satisficing the conflicting software qualities of maintainability and performance at the source code level." in *WER*. Citeseer, 2004, pp. 176–188.

[13] N. B. Harrison and P. Avgeriou, "Leveraging architecture patterns to satisfy quality attributes," in *European conference on software architecture*. Springer, 2007, pp. 263–270.

[14] R. Gheyi, T. Massoni, and P. Borba, "A rigorous approach for proving model refactorings," in *Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering*, 2005, pp. 372–375.

[15] J. Zhang, Y. Lin, and J. Gray, "Generic and domain-specific model refactoring using a model transformation engine," in *Model-driven Software Development*. Springer, 2005, pp. 199–217.

[16] C. e. Mokaddem, H. Sahraoui, and E. Syriani, "Recommending model refactoring rules from refactoring examples," in *Proceedings of the 21th ACM/IEEE International Conference on Model Driven Engineering Languages and Systems*, 2018, pp. 257–266.

[17] A. Ghannem, M. Kessentini, M. S. Hamdi, and G. El Boussaidi, "Model refactoring by example: A multi-objective search based software engineering approach," *Journal of Software: Evolution and Process*, vol. 30, no. 4, p. e1916, 2018.

[18] M. Strittmatter, G. Hinkel, M. Langhammer, R. Jung, and R. Heinrich, "Challenges in the evolution of metamodels: Smells and anti-patterns of a historically-grown metamodel," 2016.

[19] S. K. Debray, W. Evans, R. Muth, and B. De Sutter, "Compiler techniques for code compaction," *ACM Transactions on Programming languages and Systems (TOPLAS)*, vol. 22, no. 2, pp. 378–415, 2000.

[20] D. Jackson, *Software Abstractions: logic, language, and analysis*. MIT press, 2012.

[21] S. Maoz, J. O. Ringert, and B. Rumpe, "Cd2alloy: Class diagrams analysis using alloy revisited," in *International Conference on Model Driven Engineering Languages and Systems*. Springer, 2011, pp. 592–607.

[22] M. Genero and M. Piattini, "Empirical validation of measures for class diagram structural complexity through controlled experiments," in *Proceedings of the 2002 International Symposium on Empirical Software Engineering*. Citeseer, 2001.

[23] F. T. Sheldon and H. Chung, "Measuring the complexity of class diagrams in reverse engineering," *Journal of Software Maintenance and Evolution: Research and Practice*, vol. 18, no. 5, pp. 333–350, 2006.

[24] T. Arendt, F. Mantz, and G. Taentzer, "Uml model quality assurance techniques," *Philipps-Univ. Marburg, Marburg, Germany, Tech. Rep*, 2009.

[25] M. Kessentini, H. Sahraoui, M. Boukadoum, and O. B. Omar, "Search-based model transformation by example," *Software & Systems Modeling*, vol. 11, no. 2, pp. 209–226, 2012.

[26] T. Arendt and G. Taentzer, "Integration of smells and refactorings within the eclipse modeling framework," in *Proceedings of the Fifth Workshop on Refactoring Tools*. ACM, 2012, pp. 8–15.

[27] ——, "A tool environment for quality assurance based on the eclipse modeling framework," *Automated Software Engineering*, vol. 20, no. 2, pp. 141–184, 2013.

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

[29] J. Reimann, M. Seifert, and U. Aßmann, "Role-based generic model refactoring," in *International Conference on Model Driven Engineering Languages and Systems*. Springer, 2010, pp. 78–92.

[30] Y. Sun, J. Gray, and J. White, "Mt-scribe: an end-user approach to automate software model evolution," in *2011 33rd international conference on software engineering (ICSE)*. IEEE, 2011, pp. 980–982.

[31] P. Broschy, P. Langer, M. Seidl, and M. Wimmer, "Towards end-user adaptable model versioning: The by-example operation recorder," in *2009 ICSE Workshop on Comparison and Versioning of Software Models*. IEEE, 2009, pp. 55–60.

[32] I. Baki and H. Sahraoui, "Multi-step learning and adaptive search for learning complex model transformations from examples," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 25, no. 3, pp. 1–37, 2016.

[33] K. Tariq, "Linking alloy with smt-based finite model finding," Master's thesis, University of Waterloo, 2021.