“What Happened to my Models?” History-Aware Co-Existence and Co-Evolution of Metamodels and Models

Marcel Homolka
ISSE - Johannes Kepler University Linz
Linz, Austria
marcel.homolka@jku.at

Luciano Marchezan
ISSE - Johannes Kepler University Linz
Linz, Austria
lucianomarchp@gmail.com

Wesley K. G. Assunção
North Carolina State University
Raleigh, USA
wguezas@ncsu.edu

Alexander Egyed
ISSE - Johannes Kepler University Linz
Linz, Austria
alexander.egyed@jku.at

***Abstract*—Metamodels like many other software artifacts, are expected to evolve and exist in different versions. Consequently, the instances of these metamodels (models), become invalid and need fixing. The usual strategy for this is adapting models based on changes made to the metamodels, i.e., co-evolution. However, co-evolution usually adopts an all-or-nothing strategy that overlooks three important aspects: (a) preserving the history of the metamodel, (b) preserving the model's history before the co-evolution, and (c) supporting models to co-exist for different metamodel versions and delay their co-evolution. These aspects lead to problems for maintaining models in practice since often co-evolution gets driven by customer needs, e.g., the customer decides when to update. In this paper, we propose a novel approach that allows the creation of a metamodel version and records the history of metamodels and models by adopting an operation-based infrastructure. These metamodel versions allow engineers to delay the co-evolution of certain models by having co-existing metamodel versions. This notion of co-existence, in addition to the operation-based infrastructure, helps to preserve the complete history of metamodel and models, i.e., the history before and after co-evolving a model. To evaluate our approach, we conducted an empirical study, where we co-evolved models of varying domains. The results show that our approach correctly records the history of the metamodel and model. Furthermore, we measured the performance during co-evolution while having all versions co-existing in the same space. It shows that in the worst case, our approach required 887.91 seconds to co-evolve a model with more than 88,000 elements and 2,000,000 properties.**

***Index Terms*—metamodel versioning and maintenance, metamodel evolution, model co-evolution, recording history**

I. Introduction

Metamodels are software artifacts that are widely used in various industries and fields [1]–[3]. They are commonly employed in Model-Driven Engineering (MDE) to describe complex domains [4], [5]. These metamodels define the structure of models, i.e., the properties a model should have and how they relate to other parts of the domain. The relationship between a metamodel and the model is similar to the one between a class and its instance, where each model is an instance of a particular metamodel [5], [6].

Like any other software artifact, a metamodel changes over time, as shown in empirical studies on domain-specific languages (DSLs) [7]. However, changes made to a metamodel directly affect all its models. As a result, models that do not conform to the new metamodel become invalid and need to be updated, i.e., evolved. The usual strategy to fix models is by co-evolution [8]–[18]. Co-evolution is the process of adapting models according to the changes made to their metamodel to maintain a model's consistency to the metamodel over its lifespan [15]. In the field of co-evolution, most research focuses on two strategies. They either focus on (a) assisting engineers during the co-evolution process [12], [13] or (b) trying to solve the co-evolution by fully automating it [8]–[10], [17], [18]. The limitation both strategies have in common is that they usually do not take the previous version of the model and the metamodel into account. Existing co-evolution approaches often consider the previous versions of the metamodel and models as irrelevant or unimportant artifacts that get discarded after the co-evolution is complete. Consequently, the entire history of the models, e.g., changes over time, are lost.

The all-or-nothing strategy described above, however, leads to three main issues. First, *incomplete history of metamodels*. In existing approaches, recording the metamodel history is seen as an afterthought, as previous versions get just discarded [19]. However, a recent survey with modeling engineers has shown that one of the most requested features of modeling tools is to allow engineers to compare and create versions of models and metamodels [20]. This finding highlights the need for a systematic way of recording the metamodel evolution. Second, *incomplete history of models*. Similar to the previous issue, the history of models is usually lost after the co-evolution, since the old metamodel version gets discarded [3]. However, this leads to problems in which the model's history prior to the co-evolution gets lost, thus making it challenging to recover previous changes made to the model, which otherwise are used to maintain the model [21], [22]. Third, *forced co-evolution*. Co-evolution usually gets applied to all models.

©2024 IEEE. Personal use of this material is permitted. Permission from IEEE must be obtained for all other uses, in any current or future media, including reprinting/republishing this material for advertising or promotional purposes, creating new collective works, for resale or redistribution to servers or lists, or reuse of any copyrighted component of this work in other works. DOI: 10.1109/ICSME58944.2024.00027

However, there are instances where the co-evolution of models is not feasible or recommended, e.g., due to incompatibility with an older hardware/library or customer constraints [23]. In those cases, it should be possible to allow engineers to delay or even prevent the co-evolution of subsets of models by having them co-exist in the same environment as the evolved models [11].

To address the limitations of existing work, we present a novel approach that records the history of metamodels and models during their co-evolution process while enabling models of different metamodel versions to co-exist. To achieve this, changes made to the metamodels and models get recorded on an operation-based chain. The changes between each version get recorded to allow engineers to create different model versions. These changes get used by engineers to access the complete history of a metamodel and use this history to apply automatic co-evolution strategies to their models. Additionally, the operation chain stores the changes made to the model without losing information during the co-evolution. Finally, our approach enables the co-evolution of model subsets (partial co-evolution) by having the metamodel versions co-exist in the same environment, which is achieved by introducing a versioning notation on the meta-metamodel layer.

To evaluate our approach, we performed two evaluations. First, we focused on measuring the correctness of the recorded history of metamodels. We have performed common evolution changes on seven different metamodels. The results show that our approach can record the applied refactorings correctly on metamodels of varying domains and complexity. Second, our evaluation focused on co-evolving models in a co-existing system. Here, we measured whether the history of a model remained intact after a co-evolution, i.e., the changes got applied to the original model rather than a new model, and how the co-existing system affects the performance of our approach. We co-evolved 250 PlantUML and 1180 FHIR models with our approach while having models of different versions co-exist in the same environment. The result showed that during the performed co-evolutions, all the models kept their history intact, while overall, in the worst case, our approach required 887.91 seconds to co-evolve a model consisting of more than 80,000 elements and 2,000,000 properties.

The remainder of this paper is structured as follows: Section II provides the background concepts and a motivational example for our work. Section III describes our approach. The evaluation of our approach is shown in Section IV and discussion and implications are presented in Section V. Section VI highlights work that is related to our approach. Lastly, Section VII has the concluding remarks.

II. Background and Motivation

In this section, we introduce a motivating example of an evolving metamodel and a co-evolving model to highlight the issues addressed in this paper. The example was adapted from a related work on the technical debt of model evolution [11]. Figure 1 shows a metamodel (at the top) alongside its evolution over time. This metamodel is used to create services that are deployed on machines, allowing the creation of two model elements, namely `Service` and `Port`. While `Service` describes the running service on the machine, `Port` describes the required ports. A port can be either of type `InPort`, i.e., a port that handles incoming messages, or `OutPort`, i.e., a port that is used to send messages. In *Version 1* of the metamodel, the `Service` component uses these ports via `inPort` and `outPort` properties that contain the respective ports. However, later on, it was decided to further simplify the metamodel by merging the properties `inPort` and `outPort` into a new property called `ports`, thus creating *Version 2* of this metamodel.

The importance of recording the metamodel history, e.g., for using it to co-evolve models, becomes apparent even with simple metamodels, such as the one from Figure 1. Two common approaches for recording the history are (i) state-based and (ii) operation-based versioning [24]. State-based approaches such as Git [25] store the state of an artifact and derive the difference by comparing two different states. In contrast, operation-based approaches like Eclipse Edapt [26] represent changes as transformation operations performed on a state to obtain a successor state.

Typically, state-based approaches are insufficient to extract a complete history of an artifact, as demonstrated by a recent study on extracting code refactorings from Git commits [27] and studies on reconstructing metamodel changes [28], [29]. To emphasize this, we look at how `inPort` and `outPort` have been merged into `ports` as this simple change poses issues for state-based version control systems such as EMF Compare [30]. EMF Compare is a tool to compare and merge two different models and metamodels created in the Eclipse Modeling Framework (EMF) [31]. While such tools would often detect simple changes in the metamodel, such as the addition or deletion of properties, more complex changes, like the merging of properties, are often mislabeled. In the case of the merge, tools such as EMF Compare would either detect that both properties, i.e., `outPort` and `inPort`, were deleted while a new property called `ports` was added (see ① in Figure 1), or they would detect that one of the properties, i.e., `outPort`, was deleted, while the other, i.e., `inPort`, was modified to match the definition of `ports` (see ②). This incomplete and incorrect history introduces many problems in managing and maintaining the metamodel and models. In particular, it often results in an incorrect model co-evolution.

The other strategy for recording the history is using operation-based versioning. This strategy allows storing more detailed information about changes. Recent studies in the field of metamodel evolution are using such operations by providing a refactoring catalog [8], [32] to assist engineers in the evolution of metamodels. Some of these approaches even group multiple operations into so-called `Coupled Operations`, i.e., an operation that consists of multiple sub-operations, to allow a more granular recording of the history [8], [33], [34]. However, such approaches can still face limitations if the complete history is not preserved. For example, the bottom part of Figure 1 (Model section) illustrates a simple event logger service with two ports. One `InPort` called `Events`,

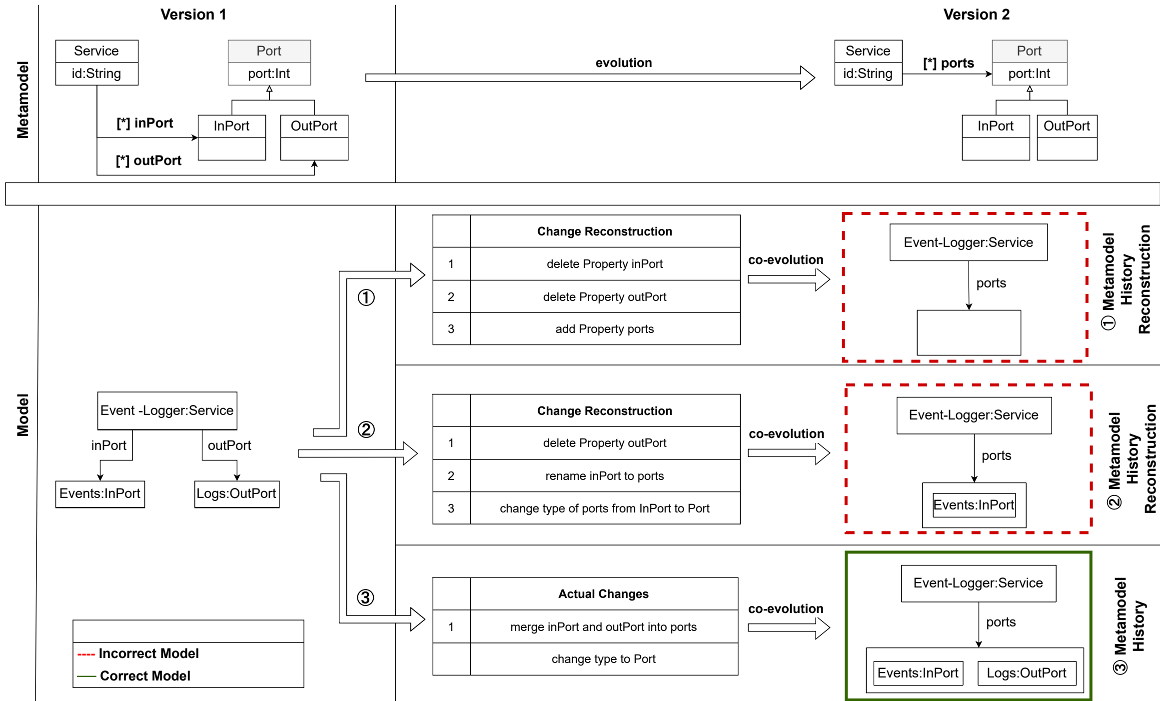

Fig. 1. Motivating Example of a Metamodel describing a Service Architecture

where it receives events from different services. These events get processed by the service and sent back as logs via the `OutPort` called `Logs`. If a developer now uses the recovered changes from ① as the basis for the co-evolution, both the elements in `inPort` and `outPort` would be deleted, while a new empty property called `ports` gets created (see co-evolution ① for the Event-Logger model). For the second case (see metamodel history ②), `inPort` would be transformed to `ports` still containing the `EventPort` and `outPort` would be deleted. While both of these co-evolved models are still syntactically correct regarding Version 2 of the metamodel, semantically they are different from before, thus impacting the design and implementation of the co-evolved models. This highlights the need for *using the complete history as a basis for co-evolving models* (**Limitation 1**). For instance, considering the complete history (see metamodel history ③ in Figure 1), both `inPort` and `outPort` are merged into the new property called `ports`, thus ensuring that the evolved model is semantically still the same.

Another issue with co-evolution approaches is their tendency to generate a new model based on the existing one, rather than adapting it. This results in the deletion of old models. Consequently, the model's history until that point is lost, making it difficult to restore the history. This also creates the challenge of *distinguishing between newly created models and those that have co-evolved from previous versions*, thus further deteriorating the model's history (**Limitation 2**). Another issue of existing work is the *all-or-nothing approach that forces all models to co-evolve once the metamodel evolves* (**Limitation 3**). There are situations, however, where models cannot be co-evolved, e.g., due to constraints of specific customers or incompatibility with an older hardware or library [23]. In such situations, it should be possible to delay the co-evolution for some models and allow these old models to co-exist in the same environment as the evolved models.

These three limitations emphasize the importance of recording a complete change history of the metamodel, using the metamodel's history to perform the co-evolution of models, and allowing delayed co-evolution of these models. In the next section, we present our approach, designed to address these limitations.

III. Proposed Approach

The underlying principle of our approach is to allow engineers to create metamodel versions, where the changes made to these metamodels, i.e., metamodel refactoring, are recorded. To achieve this, we based our approach on state-of-the-art concepts regarding operation-based MDE versioning infrastructures [35], [36]. In this infrastructure, artifacts are described as elements containing properties. Changes to these artifacts are then recorded via corresponding create, delete, and update operations on these elements and their properties. These operations can be grouped into `Coupled Operations` to provide a more comprehensive history of a given element, i.e., model and metamodel.

We then adapted this meta-metamodel to allow the creation of metamodel versions and instantiate models of those versioned metamodels, i.e., allowing co-existence between different versions. Furthermore, the operation architecture is defined with the notion of the `Coupled Operations`, i.e., multiple operations grouped, for recording a more granular

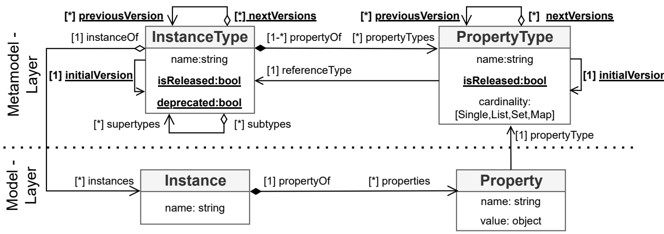

Fig. 2. Meta-metamodel for versioned metamodels

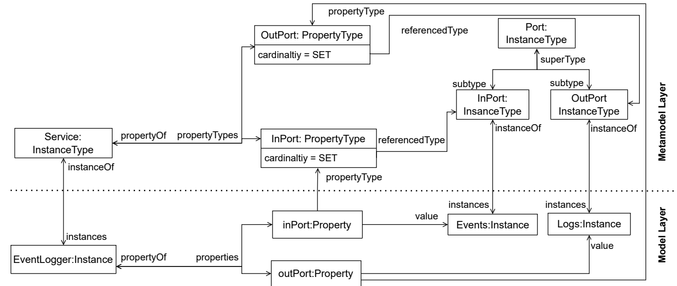

Fig. 3. Service metamodel and model created with our meta-metamodel

history, which can then be used to automate the co-evolution process. Details are presented next.

A. Our meta-metamodel

The proposed meta-metamodel, illustrated in Figure 2, consists of two layers. The first layer, called the *Metamodel Layer*, is used to describe metamodels, while the other layer, called *Model Layer*, describes the models that conform to the previously defined metamodel. The *Metamodel Layer* has two elements called `PropertyType` and `InstanceType`. In it, `InstanceType` describes the structure of a model element, i.e., what properties/features the given model element will have, whereas `PropertyType` is used to define the structure of a given property. A `PropertyType` contains multiple fields, e.g., the `cardinality` field defines whether the property is a *List*, *Set*, *Map* or a *Single* property. On the *Model Layer*, there are two elements called `Instance` and `Property`. Here, an `Instance` describes a given model element, e.g., the Event-Logger `Service` from our motivating example, where its field `instanceOf` points to the corresponding `InstanceType` of a given model element. A `Property`, on the other hand, describes the properties of an element and it is used to store the given value of the property. The `Property` references its corresponding `PropertyType` via `propertyType`.

Figure 3 illustrates how `Version 1` of the service metamodel and the Event-Logger model presented in our motivational example (Figure 1) get represented using our meta-metamodel. It shows that on the *Metamodel Layer*, four InstanceTypes were created, three for the ports, i.e., `Port`, `InPort`, and `OutPort`, and one type for the `Service`. The property `propertyTypes` of `Service` contains two PropertyTypes, one for the `inPort` and one for `outPort`. Both are *Set* properties, as defined by `cardinality`, that only allow instances of `InPort` or `OutPort`, i.e., see the association `referenceType` between PropertyType and InstanceType. On the *Model Layer*, we can see the Event-Logger (bottom of Figure 3). Here, an instance called `EventLogger` was created. It is of the InstanceType `Service` and has two properties called `outPort` and `inPort`, as seen in the field `properties`. The `outPort` is of the PropertyType `OutPort` and contains `Logs`, while the `inPort` is of type `InPort` and has `Events` inside. Both the `Events` and `Logs` are instances of their respective type, i.e., `InPort` and `OutPort`. It is important to note that this is only the internal representation for metamodel and models. Engineers would still see and use the simplified version of models and metamodels, similar to the one from the motivating example, rather than the complete representation shown here.

B. Versioning Principle

Figure 2 highlights the elements of our meta-metamodel that get used for the versioning of metamodels created with our approach, i.e., the underlined elements in `InstanceType` and `PropertyType`. The properties `previousVersion` and `nextVersion` point to the predecessor and successor versions of a given type, whereas the `initialVersion` points to the first version of a given type. The versioning mechanism works as follows. During the creation of a new type, the `previousVersion` and `nextVersion` are set to `null` and `initialVersion` points to the newly created type. Now engineers can modify the given type. When they are satisfied with the current state of the type, they can set this version to released by setting the boolean `isReleased` to true. Now, this version is locked and cannot be modified anymore, which allows the creation of instances of this type. When engineers want to adapt a type, i.e., evolve it, they need to create a new version from it. This means that the type's current state is used as the base for the new version. For `InstanceTypes` it would also mean that it still reuses the `PropertyTypes` from the old version to remove duplication of types. The newly created version is set as a successor, i.e., `nextVersion`, of the old version, and the old version is set as the predecessor of the new version, i.e., `previousVersion`. Finally, there is also the property `deprecated` which warns the user that no new instances of this type should be created. To demonstrate our approach, we show how the versioning mechanism is used for the service metamodel, as seen in Figure 4. **Creation of Version 1:** Here, the initial version of the service metamodel is created. This version is the same as shown in Figure 3. At first, an engineer would create the IntanceTypes for the `Service` and the different ports, i.e., `InPort`, `OutPort` and `Port`. As mentioned before, the field `initialVersion` points to the first version of a given type and is used to help to distinguish between different types. In this version, the created types would also be their `initialVersion`, since they are the first version created. Afterwards, all PropertyTypes, i.e., `outPort` and `inPort`, are created and added to the `Service`. Finally, this version is released and all the created

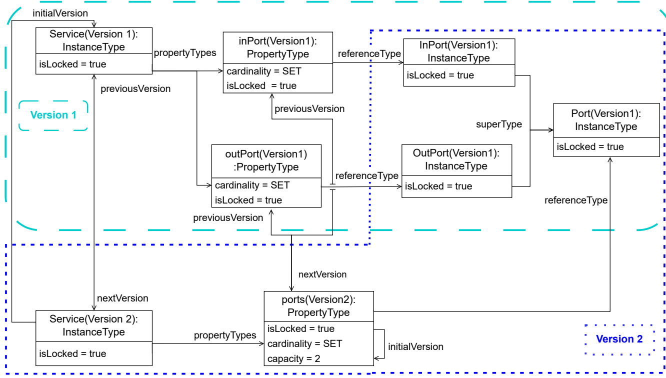

Fig. 4. Two co-existing metamodel versions

types are locked. **Creation of Version 2:** Here, `outPort` and `inPort` get merged into a new `PropertyType` named `ports`. Since `ports` is a type created by a merge, it has both `inPort` and `outPort` as its `previousVersions`. Next, a new version of `Service` is created. First, the current state of the previous version of `Service` is copied by our approach and set as the second version, still referencing the old PropertyTypes. This allows one to adapt the PropertyTypes by replacing the old types from the `propertyType` field and adding their new versions, i.e., replacing `inPort` and `outPort` with `ports`. Furthermore, this demonstrates how the approach reuses parts of the metamodel from the previous iterations to help reduce the complexity and size of the metamodel. Here, the current version reuses `InPort`, `OutPort` and `Port` from `Version 1` since no changes have been made to them (notice that `Version 1` and `Version 2` have overlapping elements in Figure 4).

C. Operation chain

To record the history of the metamodel and its models, we use an operation chain, i.e., a list of operations, that describes changes performed in the metamodels and models. Each operation from the chain represents a change that only affects a single property at a time. Additionally, these operations are also reversible, i.e., they allow engineers to restore previous states in the chain by inverting the operation. Each has a unique ID, a predecessor and a successor operation to allow easy traversal through the chain. All elements from the metamodels and models, i.e., `Instance`, `InstanceTypes` and `PropertyTypes`, are recorded as elements on the chain, where an element is described as an object that can have multiple properties. Figure 5 shows the currently supported operations that get recorded on the chain. These are `ElementCreate`, i.e., creation of a new element, `ElementDelete`, i.e., deletion of an element, and `ElementUpdate`, i.e., an element has been modified. `ElementUpdates` are operations that describe that `Properties` have been created (`PropertyCreate`), deleted (`PropertyDelete`) or modified (`PropertyUpdate`). A `Property` can be updated by either setting new values for it (`Update`), removing values

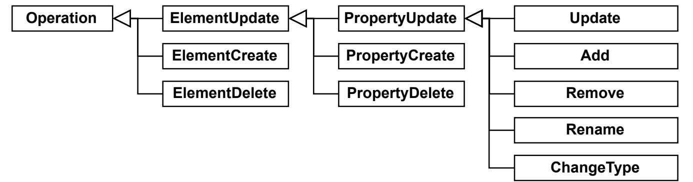

Fig. 5. List of supported Operations

(`Remove`), adding new values (`Add`), renaming the property (`Rename`), or changing its type (`ChangeType`).

Due to the atomic nature of operations, it is difficult to track complex changes, similar to state-based comparison tools, e.g., detecting the merging of properties. For that, we expanded the operation chain with the notion of `Coupled Operations`. They are defined as a group of operations that are grouped together. Those `Coupled Operations` can be labelled to allow the recording of a more complete change history, which in turn can then be used as a basis for co-evolving the corresponding models. This grouping and labeling of the operations can be performed either manually by an engineer or automatically by applying predefined refactorings, e.g., create property, merge property, etc. To show how this operation chain and `Coupled Operation` work, Figure 6 provides a simplified operation chain based on the changes made in the motivating example, i.e., the changes made to the `Service` element.

The operation chain starts with the creation of the first version of the `Service` type as shown in operations ① - ⑤ in Figure 6 and Table I. First, an InstanceType named `Service` is created (operation ①). Next, the PropertyType `inPort` is created (operation ②) and added to `Service` as a `propertyType` (operation ③). Similar to `inPort`, `outPort` is also created and added (operations ④ and ⑤). Those operations are then grouped together into `Coupled Operation` [1] (see Table II) and labeled as the creation of the `Service` type. Later, the initial version of `Service` is released (operation ⑥), and a second version of `Service` is created (operation ⑦). The modification to the second version of the `Service` type, i.e., merging `inPort` and `outPort` into `ports`, can be seen in Operations ⑩ - ⑮. They show that a property `ports` has been created and how `inPort` and `outPort` were removed. Those operations already show that just having a simple operation-based structure is not enough to record the history of a metamodel because they are prone to be interpreted differently without further context, e.g., see the motivating example. However, to mitigate it, these operations are grouped into a `Coupled operation` as a merge refactoring (see `Coupled Operation` [4]), which allows us to use the structured refactorings to automatically co-evolve the models according to the changes made to the metamodel.

D. Using Refactorings from the chain to co-evolve models

This section shows how `Coupled Operations` can automate the co-evolution process. For that, we look again

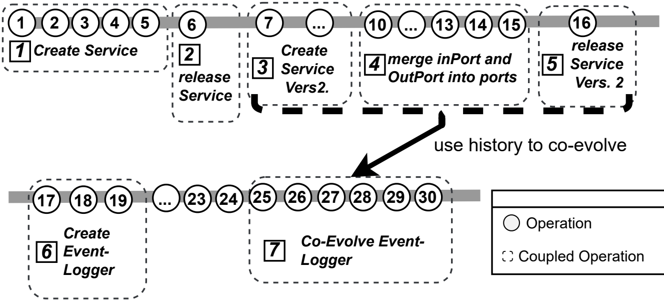

Fig. 6. Operation chain of types (metamodels) and instances (models)

TABLE I
OPERATIONS SHOWN IN FIGURE 6

ID	Operation*	Element	Name	Kind	Value
(1)	E. Create	InstanceType	Service		
(2)	E. Create	PropertyType	inPort		
(3)	P. Update	Service	propertyTypes	ADD	inPort
(4)	E. Create	PropertyType	outPort		
(5)	P. Update	Service	propertyTypes	ADD	outPort
(6)	P. Update	Service	isReleased	SET	true
(7)	E. Create	InstanceType	Service 2		
...					
(10)	E. Create	PropertyType	ports		
...					
(13)	P. Update	Service 2	propertyTypes	REM	inPort
(14)	P. Update	Service 2	propertyTypes	REM	outPort
(15)	P. Update	Service 2	propertyTypes	ADD	ports
(16)	P. Update	Service 2	isReleased	SET	true
(17)	E. Create	Service	Event-Logger		
(18)	P. Create	Event-Logger	inPort		
(19)	P. Create	Event-Logger	outPort		
...					
(23)	P. Update	Event-Logger	inPort	ADD	Events
(24)	P. Update	Event-Logger	outPort	ADD	Logs
(25)	P. Create	Event-Logger	ports		
(26)	P. Update	Event-Logger	ports	ADD	Logs
(27)	P. Update	Event-Logger	ports	ADD	Events
(28)	P. Delete	Event-Logger	inPort		
(29)	P. Delete	Event-Logger	outPort		
(30)	P. Update	Event-Logger	instanceOf	SET	Service 2

*E. = Element; P. = Property

at our example shown in Figure 6. First, we see that Coupled Operation [6] creates the Event-Logger Service from our motivating example. Then, the inPort and outPort of the event logger are set (operations (23) and (24)), and it gets decided to co-evolve the model into the second version of Service (Coupled Operation [7]).

For that, the function coEvolveInstanceTo gets called, (see Algorithm 1), by providing the model and the given type to which the model should be co-evolved, i.e., Service (2). The function coEvolveInstanceTo first checks if the provided type is a successor version from the model's type (line 2). Afterwards, all the TypeRefactorings, i.e., all the Coupled Operations of the given type are extracted and used to co-evolve the instance. This is done by iterating over all the type TypeRefactorings of instanceType and calling their perspective coEvolve method. In the case of a merge, coEvolve gets called. This function defines

TABLE II
COUPLED OPERATIONS SHOWN IN FIGURE 6

ID	Label	Operations
1	Create Service	(1) - (5)
2	Release Service	(6)
3	Create Service Version 2	(7) - ...
4	Service 2: merge inPort and outPort into ports	(10) - (15)
5	Release Service Version 2	(16)
6	Create Event-Logger	(17) - (19)
7	Co-evolve Event-Logger into Service Version 2	(25) - (30)

Algorithm 1 Co-Evolving an Instance

```
1: function COEVOLVEINSTANCETO(instance, instanceType)
2:     if instanceType.previousVersions.contains(instance.instanceType) then
3:         creation = instanceType.createOperation;  ▷ Start of refactorings of the type
4:         release = instanceType.releaseOperation;  ▷ End of refactorings of the type
5:         current = creation;
6:         while current != release do
7:             current.coEvolve(instance); ▷ Execute the co-evolution of the refactoring
8:             current = current.nextOperation;
9:         instance.instanceType = instanceType;
10: ...
11: function COEVOLVE(instance)                ▷ Example of the merge co-evolution
12:     Property a = instance.getProperty(typeA);
13:     Property b = instance.getProperty(typeB);
14:     Property merged = instance.createProperty(mergedType);
15:     for value in a.getValues() do
16:         merged.add(value);        ▷ add all values from a into the merged property
17:     for value in b.getValues() do
18:         merged.add(value);        ▷ add all values from b into the merged property
19:     a.delete();                              ▷ delete and remove property a
20:     b.delete();                              ▷ delete and remove property b
```

how a merge refactoring, is handled by our approach. First, a new property from the merged PropertyType is created (line 14). The values of both the previous properties, i.e., inPort and outPort, get extracted and added to the new property (operations (26) and (27) from Figure 6 and Table I), and then both previous properties are deleted and removed from the instance (operations (28) and (29)). Finally, the instanceType is set to the new version, (line 9 and operation (30)), and the previously generated operations are grouped together into a new Coupled Operation to show that these changes were triggered by a co-evolution from the user. Since the co-evolution of an element gets grouped as a Coupled Operation, this also allows us to record co-evolution changes on the operation chain.

IV. EVALUATION AND RESULTS

To evaluate how our proposed approach addresses the three limitations presented in Section II, we conducted a study guided by three research questions (RQ):

RQ1: Can the proposed approach record various types of metamodel changes to provide a complete history?

Over their lifespan metamodels are the target of various types of changes that can be difficult to detect, for example, the merging of two properties or splitting a property. It is, therefore, necessary to evaluate to what extent our approach can record and group different types of evolution changes together. To answer this, we rely on a catalog of well-

known metamodel refactorings from related works [8], [37]. These refactorings are applied to metamodels across various domains, where we extract the operations generated by the refactoring process to determine if our approach correctly groups the operations as intended.

RQ2: To what extent can the change history of models be preserved during co-evolution?

The history of a metamodel is an integral part of model co-evolution. However, those models have a history before the co-evolution, which is a necessary source of information for understanding their current state and maintaining them. In this RQ, we assess how our approach preserves the history of models when those undergo automatic co-evolution due to an evolving metamodel. For that, we perform automatic co-evolution of models from different domains and check whether the evolution is performed on the original model rather than on a copy of the model, i.e., no new models get created during the co-evolution.

RQ3: To what extent is the performance of our approach impacted by the co-existence of multiple metamodel versions and their corresponding models?

Our approach is based on an operation-based infrastructure, that captures and stores every fine-grained change as an operation. Having multiple metamodels and their models stored as operations might lead to bottlenecks, reducing the applicability of our approach in the industry. This RQ focuses on evaluating the runtime performance of our approach when dealing with many models from different metamodel versions.

To answer these research questions, the evaluation was conducted in two different layers (parts) corresponding to the layers in our meta-metamodel, namely the metamodel layer and the model layer. For the metamodel layer, we measured the general applicability of recording changes made to metamodels with our approach, i.e., RQ1. For the model layer, we evaluated the automatic co-evolution and the support for co-existence with our approach, i.e., RQ2 and RQ3. We performed the evaluation on an Ubuntu machine with 32GB RAM and an i7-13700F, where we limited the tool's RAM usage to 8GB. All data used for this evaluation, i.e., the used metamodels, models, the created operations and additional results are available in our online appendix [38].

A. Metamodel Layer - Evaluation

For the first evaluation, seven different metamodels of varying sizes, domains and complexity were chosen: one based on the CAEX standard [39], one for behavior engineering [40], one used to analyze process models created with Eclipse Sirius [41], one about process-oriented modeling concepts [42], one based on the EMF implementation of UML [43], a metamodel based on the PlantUML [44] notion of an activity diagram and one metamodel that is used in the medical field (FHIR) [45]. Both PlantUML and FHIR were also used for RQ2 and RQ3 respectively. The size and complexity of the used metamodels can be seen in Table III. It shows that the medical metamodel is the most complex consisting of 229 InstanceType and 1,318 PropertyTypes (457 are attributes and 861 are references), while the smallest metamodel is the

TABLE III
COMPLEXITY AND DOMAINS OF THE METAMODELS USED IN RQ1

Metamodels	Domain	#IT	#PT	#Attributes	#References
CAEX [39]	Factory	34	93	38	55
Textbt [40]	Psychology	30	40	23	17
SimQRi [41]	Risk Assement	32	60	40	20
Dbl [42]	SDL	165	148	24	124
EMF-UML [43]	Modeling	309	821	276	545
Activity Diagram* [44]	Modeling	17	18	3	15
FHIR* [45]	Medical	229	1318	457	861

IT. = Instance Types; PT. = Property Types
Attributes contain primitive values; References contain other instances
* Both of these metamodels were also used to evaluate **RQ2** and **RQ3**

TABLE IV
OPERATIONS CREATED BY PROPERTYTYPE REFACTORINGS (RQ1)

Change	**E.Create**	**P.Create**	**Update**	**Add**	**Remove**	**time[ms]**
Merge	0	0	0	2	4	0.06
Split	0	0	0	4	2	0.06
Move	0	0	0	2	2	0.07
Create	1	17	8	2	0	0.87
Remove	0	0	0	0	2	0.16
Rename	0	0	1	0	0	0.09
Change Cardinality	0	0	1	0	0	0.03
Change Type	0	0	1	0	0	0.65

E. = Element; P. = Property
All numbers reflect the minimum, average and maximum values
The time shows the worst case to process a refactoring

activity diagram based on PlantUML consisting of only 17 InstanceTypes and 18 PropertyTypes.

To evaluate RQ1, we created a refactoring catalog, based on the ones proposed by Bettini et al. [8] and Herrmannsdoerfer et al. [37]. We adapted those catalogs to fit the notation of our proposed meta-metamodel, by combining related refactorings, e.g., rename attribute and rename reference were merged into rename property. The refactorings for properties now consist of the following: create, delete, rename, merge, split, move, change cardinality and change type of `PropertyTypes`. We have applied these changes to the metamodels from Table III. For PlantUML and FHIR, we made more systematic changes to the metamodel (see evaluation of RQ2 and RQ3), while for the others we applied each refactoring twenty times to random parts of the metamodel. During this process, we recorded the changes, i.e., operations, applied to the metamodels and the time spent to perform these operations, to evaluate whether our approach can correctly group the operations as refactorings. To collect our results, we iterated over each of the generated refactorings, i.e., `Coupled Operations`, and counted the different operations, i.e., the operations from Figure 5. Afterwards, we grouped each `Coupled Operation` according to their type from the refactoring catalog, i.e., create, merge, split, etc. We then collected a summary for each of those groups and the time required to perform them.

RQ1: Recording the Metamodel History: The results can be seen in Table IV, showing the aggregated operations performed by each of the different refactorings from our catalog. The rows show the different types of refactorings for `PropertyTypes`, e.g., merging of two properties or moving a property, while the columns show the different oper-

ations performed by the refactorings, i.e., `ElementCreate`, `PropertyCreate` etc, and their performance. It is important to note that each of the columns for the refactoring shows the minimum, average and maximum number of operations found as one number since the standard deviation for each of those different operations is zero, i.e., all metrics are the same. Furthermore, we can see, how the different refactorings were applied. If we look at, for example, the merge and split, it shows that both are reversed. Here, the merge creates four `Remove` operations, i.e., two for each `PropertyType` that got removed and two adds, i.e., for the added merged `PropertyType`, while the split refactoring is the opposite. In addition to that, all of the applied refactorings took less than 0.9ms to get executed by our prototype, implying that grouping these operations takes little time. Considering these results, we can conclude that our approach was able to correctly record and group those operations generated by the different refactorings.

> **Answering RQ1**: Our approach can record all the applied refactorings correctly, as shown by the fact that the standard deviation over the different operations is zero for each type of refactoring applied. Furthermore, applying these refactorings does not consume much time, since the worst case execution of a refactoring only took 0.87ms.

B. Model Layer - Evaluation

For the second part of the evaluation, we used the PlantUML and FHIR metamodels and imported models that needed to be co-evolved. First, we created a metamodel based on PlantUML's specification of activity diagrams, to which we added changes from our catalog (see the provided online repository for a more detailed look at the applied changes [38]). For FHIR, we implemented the DSTU2 [46] specification and its successor version STU3 [47]. In addition, we also created a synthetic successor version of DSTU2 where we extended the STU3 version with changes from our catalog, that were absent during the evolution from DSTU2 and STU3, e.g., merging or splitting of properties. To retrieve PlantUML models (activity diagrams), we mined GitHub repositories using Pydriller [48]. Next, the models were streamlined by removing any duplicates and any model that was invalid. In total, we found 250 models of different sizes and with varied complexity(Table V). For instance, the average number of properties ranges from 30 to 690, and the number of references from 6 to 138. Afterwards, the models got imported into our approach, where they were co-evolved one after the other into the next version. For FHIR, we used synthetically generated patient data from the Synthea open-source project [49]. The downloaded models are in version DSTU2 of the FHIR format and consist of 1180 different models, i.e., see Table V. Here, the average number of properties per model ranged from 4,344 to 2,135,184 while the number of references ranged from 1,267 to 622,762. Similar to PlantUML, the models were imported and co-evolved into the STU3 version and our synthetic version.

RQ2: Recording the Model History during Co-Evolution: To assess RQ2, we evaluated if the history of models remained intact after the co-evolution. To do this, we retrieved the co-evolutions performed over each model and grouped them. Similar to the previous evaluation, the number of different operations performed by each co-evolution was collected. Here, we wanted to determine whether our approach applies the changes directly to the model and thus does not create any new elements, i.e., the previous history gets preserved. Table VI shows the aggregated results with the average number of operations, the standard deviation, and the minimum, average and maximum for `ElementCreate` and `ElementDelete` operations. Again, for both deletes and creates, the standard deviation and the average are zero, which shows that no new elements are created but that all old model elements were adapted into the newer version.

TABLE V
SIZE OF THE MODELS USED FOR RQS 2-3

	Avg.	Median	Min	Max	StD.
Plant UML (250 Models)					
Instances	31.09	19	6	138	26.28
Properties	155.44	95	30	690	131.4
Primitive Properties	124.35	76	24	552	105.12
Reference Properties	31.09	19	6	138	26.28
Elements in Reference*	31.08	19	6	138	26.33
FHIR (1180 Models)					
Instances	2,588	1,548	181	88,966	5,691
Properties	62,111	37,152	4,344	2,135,184	136,585
Primitive Properties	43,995	26,316	3,077	1,512,422	96,748
Reference Properties	18,115	10,836	1,267	622,762	39,837
Elements in Reference*	41,929	24,434	2,715	1,423,456	91,743

* The number of elements a model contained over all its reference properties

TABLE VI
CREATED OPERATIONS DURING A MODEL CO-EVOLUTION (RQ2)

Co-Evolution	Avg.	Max	StD.	E.Create*	E.Delete*
PlantUML - hybrid	282.24	1,173	241.48	0	0
FHIR DSTU2 - STU3	8,281.11	294,136	20,661.03	0	0
FHIR DSTU2 - synthetic	12,950.27	480,145	29,811.16	0	0

E. = Element; * Min, Average and Max number of operations

> **Answering RQ2**: The results show that our approach preserves the history of a model before its co-evolution by updating previous elements instead of creating new ones, i.e., `ElementCreate` operations.

RQ3: Performance of Co-existence: To assess the performance of co-existence, we measured the time required to co-evolve a model during the co-evolution. Additionally, while performing the evaluation we kept the last 100 imported and co-evolved models in memory to show that our approach can handle co-existing model versions. This was also done to observe if there is an impact on the performance when co-evolving model. We performed this experiment three times for each model and used the worst results for the evaluated time. Overall there was little spread between the different experiments. Table VII shows the extracted metrics for the co-evolution of the PlantUML and FHIR models. The results show that in the worst case, the co-evolution of a PlantUML model took around 13 seconds and generated 1,173 change

TABLE VII
RUNTIME (S) FOR THE CO-EVOLUTION OF AN IMPORTED MODEL (RQ3)

	Avg.	Median	Min	Max	StD.
PlantUML - hybrid	1.81	0.99	0.02	13.41	2.36
FHIR DSTU2 - STU3	6.36	2.56	0.14	552.00	23.31
FHIR DSTU2 - synthetic	9.47	3.61	0.18	887.91	36.75

operations, while for a FHIR model, it took around 887.91 seconds and generated 480,145 change operations.

Answering RQ3: The results show that it is possible to perform an automatic co-evolution, in a system with co-existing models within a reasonable time frame. While in the worst case, the co-evolution took 887.91 seconds for FHIR, it can be argued that this is still a reasonable time frame for large and complex models, i.e., models containing more than 80,000 instances and 2,000,0000 properties, because co-evolution is only triggered once per model for each new version of the metamodel.

C. Threats to Validity

This section discusses threats to validity and how we mitigated them based on [50]. The set of refactorings used is an internal threat since they might impact the evaluation results. For RQ1, we mitigated this threat by basing our catalog on well-known refactorings from related works [8], [37] that were empirically evaluated before. For RQs 2 and 3, we mitigate this threat by co-evolving models of different complexity multiple times and considering the worst cases. Furthermore, large and complex models, i.e., see Table V, were used. An external threat is related to the generalization of our results to other model domains. For RQ1, we tried to mitigate this threat by using seven metamodels of varying sizes, complexity and domains. In RQs 2 and 3, we used models from different domains, namely design modeling (PlantUML) and medical (FHIR) to mitigate this problem. Furthermore, the generalization for the `Model Layer` can also be inferred from the obtained results of RQ1 since these results showed that the refactorings applied, created the same operations for each metamodel. A conclusion threat is the refactorings applied to the metamodels since the changes were applied to random parts of the metamodel. We mitigate this threat by applying changes occurring between the DSTU2 and STU3 specifications of FHIR to have real-world refactorings used in metamodels during the evaluation.

V. DISCUSSION AND IMPLICATIONS

In this section, we discuss the implications of our work. **Extensibility of the approach**: Our approach allows easy extension to other modeling tools (e.g., EMF or Visio). The only requirement to achieve this is implementing a parser that transforms the original representation to the one supported by our meta-metamodel. For that, it should be possible to transform the original representation into a property-value schema (which is the basis for our meta-metamodel).

Decision making: By recording the complete history of models and metamodels, alongside the support of co-existing versions and automatic co-evolution, our approach can help engineers make informed decisions for the maintenance and evolution of models and metamodels. As evidenced by the evaluation results, our approach supports a large number of co-existing models (RQ3) while still preserving the history of the metamodel and its models (RQ1 and RQ2). This means that the approach allows engineers to delay the co-evolution of models, without having any significant performance impact (RQ3's results). The evaluation also evidences that it is possible to co-evolve a model using the metamodel's history while still preserving the model's history. However, currently, our approach can only be applied when the history of the metamodels is recorded from the beginning inside our prototype. Thus, our implementation of the approach is limited to scenarios where the history is recorded or reapplied (by an engineer) within our approach. In future work, we plan to address this issue by providing a way to reconstruct the history based on previous versions of metamodels and models. **Tool Performance**: As evidenced by the evaluation results, our tool managed to maintain multiple co-existing metamodel versions. The implemented prototype showed that it allows engineers to delay the co-existence of models, without having any significant performance impact. One current limitation is that we have not yet measured the maximum number of supported concurrent versions. However, since the number of concurrently supported metamodels is often much more limited than the number of loaded models, it should not be considered an issue. Furthermore, the goal of our tool was to show the principle usability of the proposed approach. In future work, we plan to further improve the prototype's performance by only loading the currently needed parts of the metamodel or models into memory, i.e., lazy loading.

VI. RELATED WORK

This section shows an overview of related work on metamodel evolution, model co-evolution and metamodel versioning. To the best of our knowledge, no related work exists, which tries to address all of the problems highlighted in this paper (see Table VIII for a summary of the comparison). However, some studies address related aspects:

One of the most commonly used software to store the history of artifacts is Git [25]. It allows one to store information about a model's history on a textual *diff*-based level. However, this recorded history is limited as shown by recent work [27]. Furthermore, it does not support automated co-evolution of artifacts. Another tool is Eclipse Edapt [26], which records changes made to an EMF metamodel. The changes can be used to automatically co-evolve models. However, Edapt does not allow co-existence and preserving the model history. Koegel et al. [33] presents an approach focused on recording the history of a model by storing EMF models and their history in an operation-based repository. For that, they created an operations-based change-tracking, that records the changes performed on the EMF-model-element level. Their

TABLE VIII
SUMMARY OF RELATED WORK

Approach	**Metamodel Changes**	**Model Changes**	**Co-evolution**	**Co-existence**	**Versioning Type**	**Representation**
Git VCS [25]	○	◓	○	◓	State-based	Textual
Eclipse Edapt [26]	●	○	●	○	Operation-based	EMF
EMF-Store [33]	○	●	○	○	Operation-based	EMF
Khelladi et al. [28]	◓	○	○	○	Operation-based*	EMF
Vermollen et al. [29]	◓	○	○	○	Operation-based*	EMF
Kehrer et al. [51]	○	◓	○	○	Operation-based	EMF
Refactoring Catalog [8]	●	○	●	○	Operation-based	Custom
In-Between Versions [11]	◓	○	●	◓	State-based	EMF
Concurrent Versions [52]	◓	○	●	◓	State-based	EMF
COPE** [34]	●	◓	●	○	Operation-based	EMF
Getir et al.** [53]	○	◓	◓	○	Operation-based	Custom
Herac et al. [35]	◓	◓	○	○	Operation-based	Custom
Mantz et al. [17]	○	○	●	○	State-based	Graph
Demuth et al. [18]	○	○	●	○	Operation-based	EMF
Our approach	●	●	●	●	Operation-based	Custom

○ Not covered; ◓ Partially covered; ● Fully covered. ;* reconstructs `Coupled Operations`; ** records only the co-evolution

work, however, only focuses on recording the model's history, overlooking the metamodel.

There are studies that focus on recovering the history of a metamodel. The work by Khelladi et al. [28] focuses on recovering complex changes, i.e., `Coupled Operations` from simple operations. The work by Vermollen et al. [29] recovers changes by comparing two state-based versions of a metamodel. Both studies, however, only focus on the metamodel's history. Moreover, as mentioned in both studies, this recovery is only an estimate of the correct history and should not be seen as the definitive recording of this history, due to a level of uncertainty in the recovered history. Similar to them, the work from Mantz et al. [17] proposes an approach that allows engineers to transform models and their corresponding metamodel in a graph-based structure, where they can adapt the metamodel changes manually into graph-based transformation rules. That would allow them to automatically co-evolve the corresponding models. However, similar to other approaches this one does not store the changes made to the model and does not support co-existing metamodel versions.

On the other hand, the work from Bettini et al. [8] provides an extensive and executable refactoring catalog. It allows engineers to adapt any metamodel using the refactorings from this catalog. These refactorings also work on an operation-based level and allow one to automatically co-evolve models. Their work, however, only focuses on recording the history of the metamodel and using this history to co-evolve models. They do not mention any support for having multiple versions co-existing or having a recorded history of the models. Nevertheless, their work provides a highly extensive refactoring catalog, on which we based the catalog used in our evaluation. Di Ruscio et al. [11] proposes a tool to minimize technical debt during the metamodel evolution by creating an in-between version of two metamodel versions. This in-between version can be considered a union of both versions. Here, properties are declared as deprecated, when they are deleted or removed during the next iteration to help engineers find technical debt in the metamodel. This allows all models to be co-evolved into the new version instead of having multiple versions co-exist. However, similar to previous work, it only focuses on co-evolving models and does not record their history.

The work of Cicchetti et al. [52], [54] focuses on concurrent versioning of metamodels and the problems that can occur when multiple engineers are working on the same version in the same environment. They solve this issue by using a *Difference metamodel* that gets created based on a given metamodel. This *Difference metamodel* is then used to store the changes made to this metamodel, which are then merged into the concurrent versions. Another study that focuses on model co-evolution and recording of the history is the work of Herrmannsdoerfer et al. [34]. Their work proposes to record the changes made to the metamodel and the model as `Coupled Operations` in a *History model*. The changes performed to the metamodel are recorded as a *History model*, which can then be used to co-evolve any given model. This *History model* is then also used to store the changes made to the model during the co-evolution. While their work attempts to record the changes made to the metamodel, they only focus on recording the changes made during the co-evolution, rather than over its entire lifespan. Similar to other related work, their work does not allow co-existing metamodel versions. Similar to that is also the work of Getir et al. [53], which proposes a tool that records the history of previous co-evolved models to assist the user during the model co-evolution. The study from Kehrer et al. [51] focuses on visualizing and grouping the changes made to models as user-level edit operations to give a better overview of those changes. Finally, the work from Demuth et. al. [18] focuses on automatically co-evolving models via consistent change propagation, where they immediately co-evolve the corresponding models after a change gets applied to a metamodel instead of bundling multiple changes together into a new version.

In summary, while these studies all address aspects of our work (or at least partially address them), none attempt to solve all of these problems at once. In addition, none of these papers have yet focused on co-existing metamodel versions.

Table VIII provides an overview of the related work presented in this section. It shows the supported aspects of each of the related works. The column `Metamodel Changes` shows the support for recording changes to metamodel, while `Model Changes` shows the support for recording the model history. `Co-Evolution` shows which approach supports the automatic co-evolution of a model from one version to another, while the column `Co-Existence` shows the related work that partially covers something similar to co-existence. Finally, there is `Versioning Type`, which shows how the changes are recorded and `Representation`, which tells how metamodels or models get represented in this approach. As shown in the comparison, our approach is novel since it is the first to completely support co-existence, as well as supporting all other features that are valuable for the field of metamodels and models maintenance and (co-)evolution.

VII. Conclusion and Future Work

In this paper, we present a novel approach that supports recording a complete and correct history of metamodels and models while allowing for different metamodel and model versions to co-exist. Furthermore, our approach allows models to be automatically co-evolved by using the history of the metamodel while keeping the model's history preserved. This is achieved by introducing a versioning notation on the meta-metamodel layer and using `CoupleOperations` to group multiple co-evolution operations, which opens the door for new research opportunities. The labelling of operations allows engineers to differentiate between automatic co-evolutions and manual co-evolutions performed by an engineer or general changes made to the model. In addition, co-existence allows engineers to concurrently work on multiple metamodel versions that are merged later or even evolve in parallel. Our evaluation showed that our approach supports co-existence of metamodel versions without impacting the performance. Additionally, our prototype allows us to store the changes made to metamodels and models correctly. For future research, we plan to extend our approach to allow reconstructing the history of a metamodel. We also plan to introduce a refactoring catalog on the model layer similar to the metamodel refactorings to allow a more granular recording of the model history.

VIII. Data Availability

The evaluation's artifacts and detailed results are available in an online repository [38].

IX. Acknowledgments

This research has been funded by the Austrian Science Fund (FWF, P31989-N31) and the FFG-COMET-K1 Center "Pro^{2}Future" (881844).

References

[1] N. Medvidovic, D. S. Rosenblum, D. F. Redmiles, and J. E. Robbins, "Modeling Software Architectures in the Unified Modeling Language," *ACM Trans. Softw. Eng. Methodol.*, vol. 11, no. 1, p. 2–57, jan 2002.

[2] J. Hutchinson, J. Whittle, M. Rouncefield, and S. Kristoffersen, "Empirical assessment of MDE in industry," in *33rd International Conference on Software Engineering (ICSE)*, 2011, pp. 471–480.

[3] D. Akdur, V. Garousi, and O. Demirörs, "A survey on modeling and model-driven engineering practices in the embedded software industry," *Journal of Systems Architecture*, vol. 91, pp. 62–82, 2018.

[4] D. C. Schmidt, "Model-driven engineering," *Computer*, vol. 39, 2006.

[5] M. Brambilla, J. Cabot, and M. Wimmer, *Model-driven software engineering in practice*. Morgan & Claypool Publishers, 2017.

[6] A. Wasowski and T. Berger, *Domain-Specific Languages: Effective modeling, automation, and reuse*. Springer, 2023.

[7] H. S. Borum and C. Seidl, "Survey of established practices in the life cycle of domain-specific languages," in *25th International Conference on Model Driven Engineering Languages and Systems (MODELS)*, 2022, pp. 266–277.

[8] L. Bettini, D. Di Ruscio, L. Iovino, and A. Pierantonio, "An executable metamodel refactoring catalog," *Software and Systems Modeling*, vol. 21, no. 5, pp. 1689–1709, 2022.

[9] W. Kessentini, H. Sahraoui, and M. Wimmer, "Automated metamodel/model co-evolution: A search-based approach," *Information and Software Technology*, vol. 106, pp. 49–67, 2 2019.

[10] W. Kessentini and V. Alizadeh, "Semi-automated metamodel/model co-evolution: a multi-level interactive approach," *Software and Systems Modeling*, vol. 21, pp. 1853–1876, 10 2022.

[11] D. D. Ruscio, A. D. Salle, L. Iovino, and A. Pierantonio, "A modeling assistant to manage technical debt in coupled evolution," *Information and Software Technology*, vol. 156, p. 107146, 4 2023.

[12] W. Kessentini and V. Alizadeh, "Interactive Metamodel/Model Co-Evolution Using Unsupervised Learning and Multi-Objective Search," in *23rd ACM/IEEE International Conference on Model Driven Engineering Languages and Systems (MODELS)*. ACM, 2020, p. 68–78.

[13] W. Kessentini, M. Wimmer, and H. Sahraoui, "Integrating the designer in-the-loop for metamodel/model co-evolution via interactive computational search," in *21th ACM/IEEE International Conference on Model Driven Engineering Languages and Systems (MODELS)*, 2018, pp. 101–111.

[14] G. Wachsmuth, "Metamodel adaptation and model co-adaptation," in *European conference on object-oriented programming*. Springer, 2007, pp. 600–624.

[15] L. Iovino, A. Di Salle, D. Di Ruscio, and A. Pierantonio, "Metamodel Deprecation to Manage Technical Debt in Model Co-Evolution," in *23rd ACM/IEEE International Conference on Model Driven Engineering Languages and Systems: Companion Proceedings*. ACM, 2020.

[16] M. Ohrndorf, C. Pietsch, U. Kelter, L. Grunske, and T. Kehrer, "History-based model repair recommendations," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 30, no. 2, pp. 1–46, 2021.

[17] F. Mantz, G. Taentzer, Y. Lamo, and U. Wolter, "Co-evolving metamodels and their instance models: A formal approach based on graph transformation," *Science of Computer Programming*, vol. 104, pp. 2–43, 2015.

[18] A. Demuth, R. E. Lopez-Herrejon, and A. Egyed, "Co-evolution of Metamodels and Models through Consistent Change Propagation." in *ME@ MoDELS*, 2013, pp. 14–21.

[19] R. Hebig, D. E. Khelladi, and R. Bendraou, "Approaches to Co-Evolution of Metamodels and Models: A Survey," *IEEE Transactions on Software Engineering*, vol. 43, pp. 396–414, 5 2017.

[20] M. Ozkaya and D. Akdur, "What do practitioners expect from the metamodeling tools? A survey," *Journal of Computer Languages*, vol. 63, p. 101030, 2021.

[21] M. Ohrndorf, C. Pietsch, U. Kelter, L. Grunske, and T. Kehrer, "History-Based Model Repair Recommendations," *ACM Trans. Softw. Eng. Methodol.*, vol. 30, 1 2021.

[22] L. Marchezan, W. K. Assuncao, R. Kretschmer, and A. Egyed, "Change-Oriented Repair Propagation," 2022.

[23] G. K. Michelon, W. K. G. Assunção, P. Grünbacher, and A. Egyed, "Analysis and Propagation of Feature Revisions in Preprocessor-based Software Product Lines," in *IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*, 2023, pp. 284–295.

[24] M. Koegel, M. Herrmannsdoerfer, J. Helming, and Y. Li, "State-based vs. operation-based change tracking," in *12th International Conference Model Driven Engineering Languages and Systems*, vol. 9, 2009.

[25] Git. Git. [Online]. Available: https://git-scm.com/

[26] Eclipse-Foundation. Eclipse-Edapt. [Online]. Available: https://projects.eclipse.org/projects/modeling.emft.edapt

[27] F. Niu, W. K. Assunçao, L. Huang, C. Mayr-Dorn, J. Ge, B. Luo, and A. Egyed, "RAT: A Refactoring-Aware Traceability Model for Bug Localization," in *2023 45th International Conference on Software Engineering (ICSE). IEEE*, 2023.

[28] D. E. Khelladi, R. Hebig, R. Bendraou, J. Robin, and M.-P. Gervais, “Detecting complex changes during metamodel evolution,” in *27th International Conference on Advanced Information Systems Engineering*. Springer, 2015, pp. 263–278.

[29] S. D. Vermolen, G. Wachsmuth, and E. Visser, “Reconstructing complex metamodel evolution,” in *International Conference on Software Language Engineering*. Springer, 2011, pp. 201–221.

[30] Eclipse-Foundation. EMF-Compare. [Online]. Available: https://eclipse.dev/emf/compare/

[31] ——. Eclipse-EMF. [Online]. Available: https://projects.eclipse.org/projects/modeling.emf.emf

[32] E. Cherfa, S. Mesli-Kesraoui, C. Tibermacine, S. Sadou, and R. Fleurquin, “Identifying Metamodel Inaccurate Structures During Metamodel/Constraint Co-Evolution,” in *2021 ACM/IEEE 24th International Conference on Model Driven Engineering Languages and Systems (MODELS)*. IEEE, 2021, pp. 24–34.

[33] M. Koegel and J. Helming, “EMFStore: a model repository for EMF models,” in *32nd ACM/IEEE International Conference on Software Engineering (ICSE): Volume 2*, 2010, pp. 307–308.

[34] M. Herrmannsdoerfer, “COPE–A Workbench for the coupled evolution of metamodels and models,” in *International conference on software language engineering*. Springer, 2010, pp. 286–295.

[35] E. Herac, W. K. G. Assunção, L. Marchezan, R. Haas, and A. Egyed, “A flexible operation-based infrastructure for collaborative model-driven engineering,” *J. Object Technol.*, vol. 22, no. 2, pp. 2:1–14, 2023.

[36] M. Homolka, L. Marchezan, W. K. Assunção, and A. Egyed, ““Don’t Touch my Model!” Towards Managing Model History and Versions during Metamodel Evolution,” in *Proceedings of the 2024 ACM/IEEE 44th International Conference on Software Engineering: New Ideas and Emerging Results*, 2024, pp. 77–81.

[37] M. Herrmannsdoerfer, S. D. Vermolen, and G. Wachsmuth, “An extensive catalog of operators for the coupled evolution of metamodels and models,” in *Software Language Engineering: Third International Conference, SLE 2010, Eindhoven, The Netherlands, October 12-13, 2010, Revised Selected Papers 3*. Springer, 2011, pp. 163–182.

[38] M. Homolka, L. Marchezan, W. K. G. Assunção, and A. Egyed, “ICSME 2024 Research Track: ”What Happened to my Models?” History-Aware Co-Existence and Co- Evolution of Metamodels and Models,” Jul. 2024. [Online]. Available: https://doi.org/10.5281/zenodo.10925952

[39] T. Mayerhofer, M. Wimmer, L. Berardinelli, and R. Drath, “A model-driven engineering workbench for CAEX supporting language customization and evolution,” *IEEE Transactions on Industrial Informatics*, vol. 14, no. 6, pp. 2770–2779, 2017.

[40] T. Myers, “TextBE: A textual editor for behavior engineering,” in *Proceedings of the 3rd Improving Systems and Software Engineering Conference (ISSEC)*, 2011.

[41] C. Ponsard, Q. Boucher, and G. Ospina, “SimQRi-A Query-oriented Tool for the Efficient Simulation and Analysis of Process Models.” in *2nd International Workshop on Executable Modeling, 19th ACM/IEEE International Conference on Model Driven Engineering Languages and Systems (MODELS)*, 2016, pp. 38–40.

[42] A. Blunk and J. Fischer, “Prototyping SDL Extensions,” in *8th International Conference on System Analysis and Modeling: Models and Reusability*. Springer, 2014, pp. 304–311.

[43] Eclipse-Foundation. Eclipse-MDT-UML2. [Online]. Available: https://projects.eclipse.org/projects/modeling.mdt.uml2

[44] PlanmtUML-Team. PlantUML. [Online]. Available: https://plantuml.com

[45] H. International. FHIR. [Online]. Available: https://hl7.org/fhir/

[46] ——. FHIR DSTU2 Summary. [Online]. Available: https://www.hl7.org/fhir/DSTU2/summary.html

[47] ——. FHIR STU3 Summary. [Online]. Available: https://www.hl7.org/fhir/STU3/summary.html

[48] D. Spadini, M. Aniche, and A. Bacchelli, “PyDriller: Python framework for mining software repositories,” in *26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ACM, 2018, pp. 908–911.

[49] Syhntea-Team. Syhntea. [Online]. Available: https://synthetichealth.github.io/synthea/

[50] C. Wohlin, P. Runeson, M. Höst, M. C. Ohlsson, B. Regnell, and A. Wesslén, *Experimentation in software engineering*. Springer Science & Business Media, 2012.

[51] T. Kehrer, U. Kelter, M. Ohrndorf, and T. Sollbach, “Understanding model evolution through semantically lifting model differences with SiLift,” in *2012 28th IEEE International Conference on Software Maintenance (ICSM)*. IEEE, 2012, pp. 638–641.

[52] A. Cicchetti, F. Ciccozzi, and T. Leveque, “A Solution for Concurrent Versioning of Metamodels and Models.” *J. Object Technol.*, vol. 11, p. 1, 2012.

[53] S. Getir, M. Rindt, and T. Kehrer, “A Generic Framework for Analyzing Model Co-Evolution.” in *ME@ MoDELS*. Citeseer, 2014, pp. 12–21.

[54] A. Cicchetti, F. Ciccozzi, T. Leveque, and A. Pierantonio, “On the concurrent versioning of metamodels and models: challenges and possible solutions,” in *2nd International Workshop on Model Comparison in Practice*, 2011, pp. 16–25.