"Don't Touch my Model!" Towards Managing Model History and Versions during Metamodel Evolution

Marcel Homolka
Johannes Kepler University Linz
Linz, Austria

Luciano Marchezan
Johannes Kepler University Linz
Linz, Austria

Wesley K. G. Assunção
North Carolina State University
Raleigh, USA

Alexander Egyed
Johannes Kepler University Linz
Linz, Austria

ABSTRACT

Metamodels, as any other software artifact, are expected to evolve. Consequently, the instances of those metamodels - *aka* the models - must evolve according to the changes made to the metamodels. This is commonly known as co-evolution and is a prominent research topic in Model Driven Engineering. However, co-evolution mostly adopts an all-or-nothing strategy and does not consider two important aspects, namely (i) recording the evolution history of a metamodel and (ii) allowing models to co-evolve at different times. We find that industrial co-evolution is commonly triggered by customer needs (the users of metamodels). For example, in the manufacturing domain, co-evolution tends to be tied to evolving hardware infrastructure. This implies that co-evolution is rarely dictated by the evolution of the metamodel but rather by the evolution needs of the models - and these evolution needs vary. In this paper, we propose an approach that allows engineers to record the history of a metamodel as versions and also create and maintain arbitrary models of those versioned metamodels, thus allowing engineers to co-evolve models at different times.

CCS CONCEPTS

• **Software and its engineering** → *Model-driven software engineering*.

KEYWORDS

metamodel evolution, versioning, recording metamodel history

ACM Reference Format:
Marcel Homolka, Luciano Marchezan, Wesley K. G. Assunção, and Alexander Egyed. 2024. "Don't Touch my Model!" Towards Managing Model History and Versions during Metamodel Evolution. In *New Ideas and Emerging Results (ICSE-NIER'24), April 14–20, 2024, Lisbon, Portugal.* ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/3639476.3639758

1 INTRODUCTION

Model Driven Engineering (MDE) is a software development paradigm that is widely used in different industries [1, 15]. MDE commonly describes complex domains using metamodels [30] that define an abstract description of a specific domain [4, 30]. Metamodels are used to define the structure of models, i.e., what properties a model should have and how it relates to other parts of the domain. The relationship between the metamodel and the model is similar to the relationship between a class and its instance, i.e., a model is an instance of a metamodel. A metamodel, similar to any other software artifact, is expected to change over time, as evidenced in empirical studies on domain-specific language (DSL) [3]. However, changing a metamodel typically leads to invalid models, which must be updated by applying co-evolution [2, 16, 18–21, 27, 29, 32].

Co-evolution is the process of changing a model according to the changes made to its metamodel [16], aiming to keep models consistent with metamodels over their lifespan. Current research on co-evolution focuses on two strategies: fully automate co-evolution [2, 19, 20] or assist users during the co-evolution [18, 21]. Both strategies still consider the previous versions of a metamodel as unimportant artifacts that are discarded after the co-evolution is complete. Such an all-or-nothing approach leads to two main problems: (i) *lack of history* - in existing approaches, where the previous versions are discarded, keeping track of the history of a metamodel is often seen as an afterthought [13]. Nonetheless, a recent survey with modeling engineers has shown that one of the most requested features for meta-modeling tools is the ability to create versions of models and metamodels that can be compared and merged [28]. This finding highlights the need for a systematic way of recording the metamodel evolution, that can be used for maintaining metamodels. (ii) *lack of co-existing versions* - co-evolution is usually applied to all models. However, there are situations where models cannot be co-evolved, e.g., due to constraints of specific customers or incompatibility issues with an older hardware or library. In such situations, it should be possible to delay or prevent the co-evolution of a subset of models and allow these old models to co-exist in the same environment with the evolved models. *We speak of co-existence when models of different versions of their metamodel exist within a single project*, e.g., consider a manufacturing domain where two alike machines exist, but only one is updated.

In this paper, we present an approach that addresses both of these issues by introducing the creation of metamodel versions. The changes between each version are recorded to allow engineers to create different model versions accordingly. The approach also enables the co-evolution of model subsets by having the metamodel versions co-exist in the same environment. For that, we introduce the idea of *co-existence*, which describes the ability to have different versioned metamodels and models existing together.

ICSE-NIER'24, April 14–20, 2024, Lisbon, Portugal

ACM ISBN 979-8-4007-0500-7/24/04.
https://doi.org/10.1145/3639476.3639758

2 BACKGROUND

Two common ways of recording artifact changes are (i) state-based and (ii) operation-based versioning [24]. On the one hand, state-based approaches like Git [12] store states of an element and derive the differences by comparing two different states, i.e., diffing of versions. On the other hand, operation-based versioning approaches like Eclipse Edapt [8] represent changes as transformation operations performed on a state to obtain a successor state. State-based approaches are typically insufficient to extract a complete history of an artifact [22, 26, 31]. Differently, operation-based versioning allows storing more detailed information about changes. Studies in the area of metamodel evolution use such operations by providing a refactoring catalog [2, 5] to assist engineers in the evolution of metamodels. For this reason, we adopt an operation-based infrastructure as the basis for our approach [14]. 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.

Now, we present a motivating example of a metamodel and its evolution to highlight the issues addressed in this paper. We use it to demonstrate the importance of a version history and delayed co-evolution with co-existence in the metamodel domain. The example is inspired by related work on the technical debt of model evolution [29]. Figure 1 shows a simplified metamodel, alongside its evolution over time, that is used to create services that are deployed on machines. The metamodel allows the creation of two model elements, namely `Service` and `Port`. While `Service` describes the service running on the machine, `Port` describes the ports required by the given service. In *Version 1*, the `Service` component has two properties called `inPort` and `outPort`. The `inPort` property contains all the ports where the service receives incoming messages, while `outPort` contains the ports used to send messages generated by the service. During an evolution, in *Version 2* engineers decided to merge the `inPort` and `outPort` properties into a new property called `ports` to simplify the `Service` properties. Also, a constraint was added to `ports`, to allow a `Service` to have only two `Ports`.

The importance of recording the history of a metamodel becomes apparent even with such a simple metamodel. To emphasize this, we look at how `inPort` and `outPort` have been merged into `ports`. This change already poses some issues for state-based version control systems and *diff* comparison tools such as EMF Compare [10]. EMF Compare is a tool to compare and merge two different models and metamodels created in the Eclipse Modeling Framework (EMF) [9]. While EMF Compare allows one to find simple changes in the metamodel, e.g., the creation of properties and changing of constraints, more complex changes are often mislabelled. For the merging of `outPort` and `inPort` into `ports`, it would detect that one of the properties, i.e., `outPort`, was deleted, while the other, i.e., `inPort`, was changed to match the definition of `ports`. This causes problems if one were to use this comparison as a basis for co-evolving models. The resulting models would probably all delete their `OutPorts` thus destroying the semantics of the original model. While the changes made in the motivated example are easy to detect, if several of these changes occur in a more complex metamodel, engineers would struggle to understand those changes without additional context. Thus, further demonstrating the need for a systematic way of recording the changes made to metamodels.

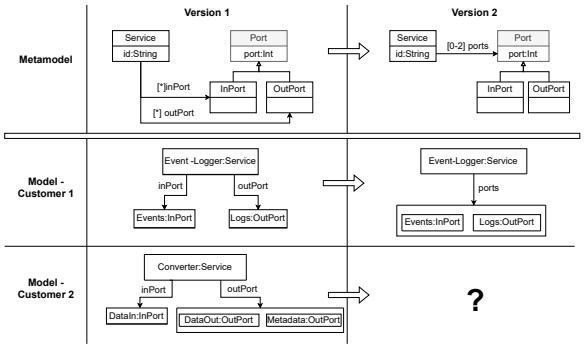

Figure 1: Evolution of a service metamodel and models

Existing co-evolution approaches aim to transform all models of a metamodel into newer versions. However, since these versions are often used to suit different customers and contexts, models should instead be evolved to support specific needs. The example from Figure 1 shows a company that hosts services for different customers. The first model is an Event-Logger for *Customer 1*, where the customer wants to upgrade to the latest version. Here, the model can be co-evolved easily without changing the semantics of this model (see *Versions 1* and *2* from *Customer 1* in Figure 1). The second service, which converts files and extracts metadata from them, i.e., the service for *Customer 2*, highlights the problem of forced co-evolution. Since this model has three ports (one incoming and two outgoing ports), it is not possible to transform this model into *Version 2* of the metamodel without changing its semantics, i.e., removing one of these ports and adapting it.

The need for a solution, in which these different versioned services can co-exist, becomes apparent with this simple example. Often engineers would try to work around this issue by either making changes to the metamodel backward compatible or having a machine that only hosts tools and models with a specific version of the metamodel, i.e., clone-and-own [11, 17, 23]. While the latter allows to manage multiple versions, it would become increasingly infeasible the more versions have to be supported [25]. Having the metamodel be backward compatible can solve this issue, however, it also creates the problem that it restricts the changes that could be made to a metamodel over its lifespan. With backward compatibility, the engineer would have to keep previous bad design decisions alive during new iterations to support older models, thus limiting the number of refactorings a metamodel can experience.

3 RELATED WORK

This section reviews related work on metamodel co-evolution, that address similar problems to our approach. The studies by Khelladi et al. [22] and Vermollen et al. [31] focus on reconstructing the history of a metamodel. Their approaches try to recover an operation-based history, by comparing two state-based versions of a model. The work from Benetti et al. [2], on the other hand, provides a refactoring catalog to cover the changes that can be made to the metamodel to assist during co-evolution. These three studies, however, do not allow different versions of metamodels to co-exist and only record the changes made to the metamodel instead of both model and metamodel.

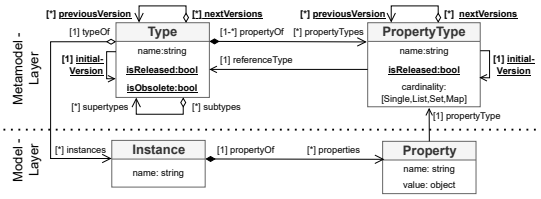

Figure 2: Meta-metamodel for co-existing metamodels

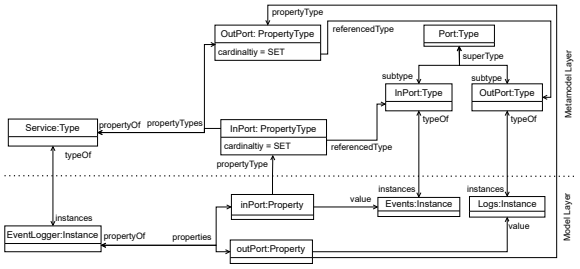

Figure 3: Metamodel and Model in the meta-metamodel

One recent work on co-evolution by Di Ruscio et al. [29] proposes a tool to minimize technical debt over the metamodel evolution by creating an in-between version of two metamodel versions. That 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 dept in the metamodel and allow all models to be co-evolved into the new version instead of having multiple versions co-exist. The work by Cicchetti et al. [6, 7] focuses on concurrent versioning and the problems that can occur when multiple engineers are working on the same version in the same environment. Their approach merges concurrent versions of metamodels and their models. They solve this issue by using a *difference metamodel*, which is created based on a given metamodel. This *difference metamodel* is then used to store the changes made to this metamodel, i.e., additions, deletions, and updates, which are then used to merge the concurrent versions.

4 PROPOSED APPROACH

In this section, we present our proposed approach for metamodel versioning and models co-existence. Our approach's underlying principle is to allow engineers to create versioned metamodels within the metamodel domain itself. For that, we use a meta-metamodel based on state-of-the-art concepts regarding MDE versioning infrastructures [14]. We adapted the given meta-metamodel to allow engineers to define metamodels, create versions of those metamodels, and instantiate models of those versioned metamodels. This enables to differentiate between the different versioned elements in the metamodel. For a better overview of the adapted meta-metamodel, we first show its core concepts, i.e., how those metamodels and models are defined (Section 4.1). This is achieved by creating the initial version of our service metamodel and the Event-Logger from the motivating example inside the proposed meta-metamodel. Next, we present the properties that were added to introduce versioning (Section 4.2). The versioning mechanism is then emphasized with the help of the service metamodel as an illustrative example. Finally, we highlight how our approach can be used to record the changes made to the metamodel (Section 4.3). This is shown by presenting the version graph of the `port` property, that was extracted from the meta-metamodel.

4.1 Meta-metamodel Core elements

The proposed meta-metamodel, illustrated in Figure 2, consists of two layers. The first layer is the *Metamodel Layer*, where each component is used to describe the metamodel. The other layer, called *Model Layer*, describes the models of metamodels that are defined in the previous layer. The *Metamodel Layer* consists of two components called `Type` and `PropertyType`. `Type` is used to define the structure of a model element, i.e., what properties the given model element will have, whereas `PropertyType` defines the structure of a given property. `PropertyType` has multiple fields that are used to define the property. The `cardinality` field describes whether the property is a *List*, *Set*, *Map* or a *Single* property, while `referenceType` is used to define the type of allowed elements in the field and `capacity` limits the number of entries that are allowed in the given property. The *Model Layer* also consists of two components, namely `Instance` and `Property`. An `Instance` describes a given model element, e.g., the Event-Logger `Service` from our motivating example, where its field `typeOf` points to the corresponding `Type` of a given model element. The component `Property` is used to describe the properties of an element, storing the given value of the property and referencing the corresponding `PropertyType` via `propertyType`. In the example of Figure 3, we can see how the initial version of the service metamodel and the Event-Logger model presented in the motivation (see the model from customer 1 in Figure 1) would look like. Figure 3 shows that on the *Metamodel Layer*, four types have been created, one for the `Service` and three types for the ports, i.e., `Port`, `InPort`, and `OutPort`. The `Service Type` has two `PropertyTypes` one for the `outPort` property and one for the `inPort` property from the metamodel. Both of these properties are *set* properties as defined by `cardinality`, and they only allow instances of type `InPort` or `OutPort` inside, i.e., see `referenceType`. The Event-Logger model can be seen within the *Model Layer* (bottom of Figure 3), in which an `Instance` called `EventLogger` was created. This `Instance` is of the type `Service` and has two properties called `inPort` and `outPort`, as seen in the field `properties`. The Property `inPort` is of the property type `InPort` and contains `Events` as a value, while the `outPort` has `OutPort` as its property type and contains `Logs`. Both the `Events` and `Logs` are `Instances` of their respective type, i.e., `InPort` and `OutPort`.

4.2 Versioning Concepts

For the versioning of metamodel components, i.e., `Type` and `PropertyType`, the underlined fields in Figure 2 were created. Specifically, the three properties `initialVersion`, `nextVersion` and `previousVersion` are used for modeling versions. The properties `previousVersion` and `nextVersion` point to the predecessor and successor versions of a given type, whereas the `initialVersion` is used to point 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

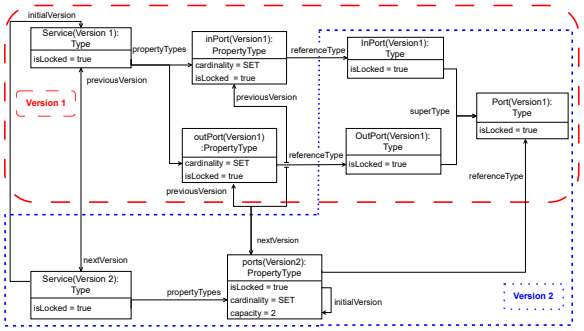

Figure 4: Two co-existing metamodel versions

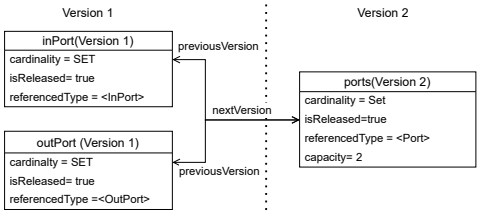

Figure 5: Version tree of the port PropertyTypes

null and initalVersion points to the newly created type. Now engineers can modify the given type and, when they are satisfied with the current state of the type, they can set this version to released by setting the property isReleased to true. With that, instances of this type can be created and, additionally, this version is locked and cannot be modified anymore. 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 cloned and used as the base for the new version. 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 cloned version, i.e., previousVersion. The engineer can now change this version, since it is a copy of the previous version. If the versioned component is a Type, it also reuses the PropertyTypes of its previousVersion since it is still referencing them via the field propertyTypes. Finally, there is also the property isObsolete which allows one to set a given type to obsolete. Then, no new instances of the given type can be created, allowing engineers to disable certain versions of Types. 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:** First, the initial version of the metamodel is created. This version is the same as the one shown in *Metamodel Layer* in Figure 3. Here, engineers would create the Types for the Service and Port components, i.e., Port, InPort and OutPort. As mentioned earlier, the field initialVersion points to the first version of a given Type or PropertyType and is used to help to distinguish between different types. Since all the created Types are new Types, i.e., the first version rather than a successor version of previous types, all of them are referencing themselves via the initialVersion field. Next, the PropertyTypes, i.e., inPort and outPort, are created and added to the Service. Again, the initialVersion is set for them. Afterwards, this version of the metamodel is set to released and all types are locked. **Creation of Version 2:** In *Version 2*, both inPort and outPort are merged into a new PropertyType called ports. Because ports was created by a merge, it has both inPort and outPort as its previousVersion. Additionally, due to the merge, ports is considered a new PropertyType, and thus it references itself via initialVersion instead of either inPort or outPort. Next, a new version of Service is created. Here the current state of the first version of the Service is cloned and set as the second version, still referencing the old PropertyTypes. This allows one to adapt the PropertyTypes accordingly 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.

4.3 Tracking the changes in the metamodel

The use of an operation-based infrastructure, allows us to track the changes made to metamodels and models through create, update, and delete operations. In addition, the proposed metamodel versioning mechanism records the changes made, i.e., the versions. This allows the history of a type to be extracted from the previousVersion and nextVersion references. Figure 5 shows the changes made to the property ports as a version graph extracted from the meta-metamodel. This version graph is a state-based graph that shows the state, i.e., the versions, of a given type over its lifetime. It allows one to extract the changes made to that type by comparing the states. For example, it allows engineers to find out if a property has been merged by comparing its previousVersions or nextVersions respectively. In the case of the example, it shows that ports has two previousVersions, i.e., inPort and outPort, which means that both of them have been merged into ports.

5 FUTURE PLANS

In this paper, we have presented a novel approach that supports recording the history of metamodels and allowing models of different metamodel versions to co-exist. This opens room for new research opportunities. For example, extending the operational semantics of the versioning infrastructure to provide more detailed information about metamodel refactorings. To achieve this, we designing an approach that enables engineers to group and label operations, which would allow us to introduce a refactoring catalog similar to the one described by recent studies [2, 5]. This enables us to introduce approaches to automate engineers in co-evolving models. Furthermore, the labeled operations add the possibility to differentiate between the changes made to models, i.e., co-evolution changes or modeling changes made by the engineer. To evaluate our approach we plan to mine Git repositories for EMF metamodels and commits made to these metamodels. We plan to use the generated dataset to show how metamodels evolve and how our approach supports different versioned metamodels coexisting.

ACKNOWLEDGMENTS

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

REFERENCES

[1] Deniz Akdur, Vahid Garousi, and Onur Demirörs. 2018. A survey on modeling and model-driven engineering practices in the embedded software industry. *Journal of Systems Architecture* 91 (2018), 62–82.

[2] Lorenzo Bettini, Davide Di Ruscio, Ludovico Iovino, and Alfonso Pierantonio. 2022. An executable metamodel refactoring catalog. *Software and Systems Modeling* 21 (10 2022), 1689–1709. Issue 5. https://doi.org/10.1007/s10270-022-01034-9

[3] Holger Stadel Borum and Christoph Seidl. 2022. Survey of established practices in the life cycle of domain-specific languages. In *25th International Conference on Model Driven Engineering Languages and Systems*. 266–277.

[4] Marco Brambilla, Jordi Cabot, and Manuel Wimmer. 2017. *Model-driven software engineering in practice*. Morgan & Claypool Publishers.

[5] Elyes Cherfa, Soraya Mesli-Kesraoui, Chouki Tibermacine, Salah Sadou, and Régis Fleurquin. 2021. 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, 24–34.

[6] Antonio Cicchetti, Federico Ciccozzi, and Thomas Leveque. 2012. A Solution for Concurrent Versioning of Metamodels and Models. *J. Object Technol.* 11 (2012), 1. Issue 3.

[7] Antonio Cicchetti, Federico Ciccozzi, Thomas Leveque, and Alfonso Pierantonio. 2011. On the concurrent versioning of metamodels and models: challenges and possible solutions. In *2nd International Workshop on Model Comparison in Practice*. 16–25.

[8] Eclipse-Foundation. 2023-12-06. *Eclipse-Edapt*. https://projects.eclipse.org/projects/modeling.emft.edapt

[9] Eclipse-Foundation. 2023-12-06. *Eclipse-EMF*. https://projects.eclipse.org/projects/modeling.emf.emf

[10] Eclipse-Foundation. 2023-12-06. *EMF-Compare*. https://eclipse.dev/emf/compare/

[11] Stefan Fischer, Lukas Linsbauer, Roberto Erick Lopez-Herrejon, and Alexander Egyed. 2014. Enhancing clone-and-own with systematic reuse for developing software variants. In *2014 IEEE International conference on software maintenance and evolution*. IEEE, 391–400.

[12] Git. 2023-12-06. *Git*. https://git-scm.com/

[13] Regina Hebig, Djamel Eddine Khelladi, and Reda Bendraou. 2017. Approaches to Co-Evolution of Metamodels and Models: A Survey. *IEEE Transactions on Software Engineering* 43 (5 2017), 396–414. Issue 5. https://doi.org/10.1109/TSE.2016.2610424

[14] Edvin Herac, Wesley K. G. Assunção, Luciano Marchezan, Rainer Haas, and Alexander Egyed. 2023. A flexible operation-based infrastructure for collaborative model-driven engineering. *Journal of Object Technology* 22, 2 (July 2023), 2:1–14. https://doi.org/10.5381/jot.2023.22.2.a5 The 19th European Conference on Modelling Foundations and Applications (ECMFA 2023).

[15] John Hutchinson, Jon Whittle, Mark Rouncefield, and Steinar Kristoffersen. 2011. Empirical assessment of MDE in industry. In *33rd international conference on software engineering*. 471–480.

[16] Ludovico Iovino, Amleto Di Salle, Davide Di Ruscio, and Alfonso Pierantonio. 2020. Metamodel deprecation to manage technical debt in model co-evolution. *23rd ACM/IEEE International Conference on Model Driven Engineering Languages and Systems: Companion Proceedings*, 1–10. https://doi.org/10.1145/3417990.3419625

[17] Cory J Kapser and Michael W Godfrey. 2008. "Cloning considered harmful" considered harmful: patterns of cloning in software. *Empirical Software Engineering* 13 (2008), 645–692.

[18] Wael Kessentini and Vahid Alizadeh. 2020. Interactive metamodel/model co-evolution using unsupervised learning and multi-objective search. *23rd ACM/IEEE International Conference on Model Driven Engineering Languages and Systems*, 68–78. https://doi.org/10.1145/3365438.3410966

[19] Wael Kessentini and Vahid Alizadeh. 2022. Semi-automated metamodel/model co-evolution: a multi-level interactive approach. *Software and Systems Modeling* 21 (10 2022), 1853–1876. Issue 5. https://doi.org/10.1007/s10270-022-00978-2

[20] Wael Kessentini, Houari Sahraoui, and Manuel Wimmer. 2019. Automated metamodel/model co-evolution: A search-based approach. *Information and Software Technology* 106 (2 2019), 49–67. https://doi.org/10.1016/j.infsof.2018.09.003

[21] Wael Kessentini, Manuel Wimmer, and Houari Sahraoui. 2018. 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*. 101–111.

[22] Djamel Eddine Khelladi, Regina Hebig, Reda Bendraou, Jacques Robin, and Marie-Pierre Gervais. 2015. Detecting complex changes during metamodel evolution. In *Advanced Information Systems Engineering: 27th International Conference, CAiSE 2015, Stockholm, Sweden, June 8-12, 2015, Proceedings 27*. Springer, 263–278.

[23] Miryung Kim, Vibha Sazawal, David Notkin, and Gail Murphy. 2005. An Empirical Study of Code Clone Genealogies. In *10th European Software Engineering Conference Held Jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering* (Lisbon, Portugal) *(ESEC/FSE-13)*. Association for Computing Machinery, New York, NY, USA, 187–196. https://doi.org/10.1145/1081706.1081737

[24] Maximilian Koegel, Markus Herrmannsdoerfer, Jonas Helming, and Yang Li. 2009. State-based vs. operation-based change tracking. In *Proceedings of MODELS*, Vol. 9.

[25] Jacob Krüger and Thorsten Berger. 2020. An empirical analysis of the costs of clone-and platform-oriented software reuse. In *28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 432–444.

[26] Feifei Niu, Wesley KG Assunçao, LiGuo Huang, Christoph Mayr-Dorn, Jidong Ge, Bin Luo, and Alexander Egyed. 2023. RAT: A Refactoring-Aware Traceability Model for Bug Localization. In *2023 45th International Conference on Software Engineering (ICSE). IEEE*.

[27] Manuel Ohrndorf, Christopher Pietsch, Udo Kelter, Lars Grunske, and Timo Kehrer. 2021. History-based model repair recommendations. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 30, 2 (2021), 1–46.

[28] Mert Ozkaya and Deniz Akdur. 2021. What do practitioners expect from the meta-modeling tools? A survey. *Journal of Computer Languages* 63 (2021), 101030.

[29] Davide Di Ruscio, Amleto Di Salle, Ludovico Iovino, and Alfonso Pierantonio. 2023. A modeling assistant to manage technical debt in coupled evolution. *Information and Software Technology* 156 (4 2023), 107146. https://doi.org/10.1016/j.infsof.2022.107146

[30] Douglas C. Schmidt. 2006. Model-driven engineering. *Computer* 39 (2006). Issue 2. https://doi.org/10.1109/MC.2006.58

[31] Sander D Vermolen, Guido Wachsmuth, and Eelco Visser. 2011. Reconstructing complex metamodel evolution. In *International Conference on Software Language Engineering*. Springer, 201–221.

[32] Guido Wachsmuth. 2007. Metamodel adaptation and model co-adaptation. In *European conference on object-oriented programming*. Springer, 600–624.