Automated Metamodel Augmentation for Seamless Model Evolution Tracking and Planning

Michael Nieke
m.nieke@tu-bs.de
TU Braunschweig
Braunschweig, Germany

Adrian Hoff
a.hoff@tu-bs.de
TU Braunschweig
Braunschweig, Germany

Christoph Seidl
c.seidl@tu-bs.de
TU Braunschweig
Braunschweig, Germany

Abstract

In model-based software engineering, models are central artifacts used for management, design and implementation. To meet new requirements, engineers need to plan and perform model evolution. So far, model evolution histories are captured using version control systems, e.g., Git. However, these systems are unsuitable for planning model evolution as they do not have a notion of future changes. Furthermore, formally assigning responsibilities to engineers for performing evolution of model parts is achieved by using additional tools for access control. To remedy these shortcomings, we provide a method to generate evolution-aware modeling notations by augmenting existing metamodels with concepts for capturing past and planned evolution as first-class entity. Our method enables engineers to seamlessly plan future model evolution while actively developing the current model state, both using a centralized access point for evolution. In our evaluation, we provide an implementation of our method in the tool *TemporalRegulator3000*, show applicability for real-world metamodels, and capture the entire evolution time line of corresponding models.

CCS Concepts • **Software and its engineering → System modeling languages**; **Software evolution**; *Domain specific languages*; *Software implementation planning*.

Keywords Model Based Software Engineering, Metamodel, Model, Evolution, Planning, History, Timeline

ACM Reference Format:
Michael Nieke, Adrian Hoff, and Christoph Seidl. 2019. Automated Metamodel Augmentation for Seamless Model Evolution Tracking and Planning. In *Proceedings of the 18th ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences (GPCE '19), October 21–22, 2019, Athens, Greece.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3357765.3359526

GPCE '19, October 21–22, 2019, Athens, Greece

ACM ISBN 978-1-4503-6980-0/19/10…$15.00
https://doi.org/10.1145/3357765.3359526

1 Introduction

In model-based software engineering, models play a pivotal role throughout the lifecycle of a project, e.g., for management, design and implementation. *Conceptual models* capture high-level concerns of a project and may be used for planning, e.g., *feature models* [30] or *process models* [23]. *Implementation models* have an operational semantics and may be the basis for code generation, e.g., state machines. To remain relevant over time, models have to evolve to meet new requirements. For this purpose, engineers perform changes on models and commit the new models to a version control system (VCS), e.g., Git [31], which aids in storing and retrieving previous model versions. We identified two major shortcomings of this practice, which we address in this paper:

First, especially for larger systems that constitute a core strategic investment for a company, planning evolution of the respective models is crucial in defining milestones for development, monitoring evolution progress and performing preliminary analyses for the planned state after evolution. In particular, conceptual models or abstract parts of implementation models would lend themselves to this planning by sketching future changes without having to specify all implementation details. However, the current practice of using VCSs for evolution does not support evolution planning and can emulate it only via workarounds, e.g., by maintaining an additional branch with a planned state of a model that has to be kept in sync by repeated merging, which may require manual resolution of resulting conflicts.

Second, in larger models, different model parts are maintained by different engineers depending on their expertise and competence. To avoid inadvertent or even malicious access, changes to model parts for which an engineer does not have the responsibility must be prevented. However, with state of practice methods, models have to be decomposed into multiple files and access to these files has to be regulated using external tools. This severely limits the granularity of access control as a model cannot generally be decomposed along areas of expertise, e.g., in tightly integrated interdisciplinary projects.

To address these shortcoming of current model evolution practices, in this paper, we provide a method to enable arbitrary modeling notations for integrated storing, tracking, planning, and access control of model evolution. We provide both the conceptual basis and a practical implementation to extend an existing notation's metamodel with first-class concepts for evolution support in their respective modeling notations. This procedure is fully automated and, while it yields a new metamodel, we also generate facilities that ensure seamless integration with an existing modeling infrastructure, e.g., editors. In the resulting evolution-aware model, the entire evolution-time line is stored in one sole artifact as direct connections between already performed evolution as well as planned future evolution and its later enactment. In the generated model evolution facilities, we provide a centralized access point to perform the actual changes. With this access point, we lay the foundation for element-based access control in terms of restricting changes to model parts without the need of an external rights management tool.

In summary, we make the following contributions:

- We provide a method to augment arbitrary metamodels with integrated support for model evolution.
- We provide means to generate a centralized evolution gateway, which serves as single point of access to evolution information while hiding the complex intricacies of model evolution.
- We enable seamlessly using evolution-aware models with existing tools by providing a set of adapters that transparently track evolution but appear similar to the original model elements regarding their interface.
- We fully automate our augmentation method, make it a generative process that is available for practical use and show its feasibility by providing an implementation within our tool *TemporalRegulator3000*.
- In our evaluation, we show that our method is applicable to real-world metamodels and that we are able to represent evolution time lines of large-scale models.

Through our augmentation mechanism for metamodels, we are able to provide evolution support for arbitrary models while striking a balance between applicability to a wide range of modeling notations, ease of use and compatibility with an existing modeling infrastructure. As result, we enable tracking evolution, active development, future evolution planning, and centralized access control in one coherent notation without the need for additional tools. Even though we present a technical solution that also copes with specifics of EMF Ecore as metamodeling notation, the concepts we provide can equally be applied to other MOF-based metamodeling notations.

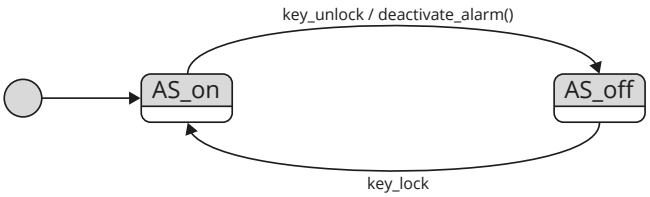

Figure 1. Exemplary State Machine for the Alarm System.

2 Background and Motivating Example

As motivating example, we use the automotive Body Comfort System (BCS) [22, 24] with slightly simplified notations and evolution history.

2.1 Example Scenario

A state machine models system behavior in terms of states and transitions [12]. In addition to common states, further state types exist: an *initial* state defines the start of execution; if an *end* state is reached, the system stops; *composed* states consist of sub states, i.e., they are state machines themselves. A transition may be labeled with *triggers*, *guards*, and *actions*. A trigger defines an event the transition reacts to. A guard defines a condition (usually a Boolean expression) under which a triggered transition may be activated. An action is executed when navigating to the target state.

Figure 1 shows an exemplary state machine for the alarm system of the BCS case study, which activates if an intrusion is detected. It consists of three states: an initial state, a state `AS_on`, which represents that the alarm system is active, and a state `AS_off`, which is active if the alarm system is deactivated. The system starts with the alarm system turned on. If the alarm system is activated and the car is opened with the key, the event `key_unlock` is fired, which activates the transition to `AS_off`. Additionally, if an alarm is active, it is deactivated if the car is opened. If the car is locked again, the event `key_lock` is fired and the alarm system is activated again. In an early version of the system, only the high-level behavior of the alarm system is specified to define interfaces with other systems, i.e., locking and unlocking of the car. The behavior for intrusion detection is not specified at this point.

2.2 Metamodel

In model-based software engineering, *models* play a pivotal role throughout the entire lifecycle of a project, e.g., for management, design, implementation, or, in the case of state machines, system behavior specification. A *metamodel* specifies a particular modeling notation in a structured way [11], e.g., in state machines, that different types of states exist that may be connected via transitions. The Meta Object Facility (MOF) is a standard for defining metamodels in a common notation. A metamodel consists of classes, attributes and

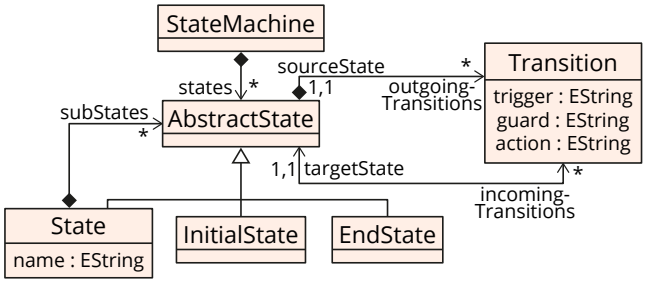

Figure 2. Metamodel of State Machines.

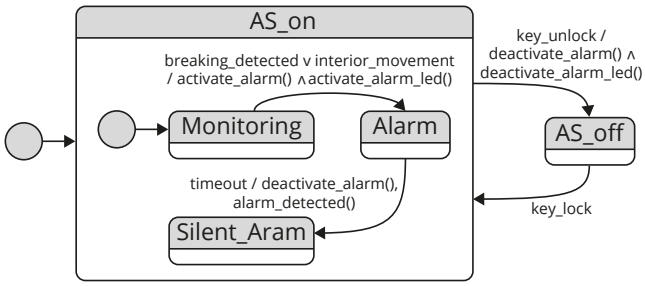

Figure 3. Planned Evolution for the Alarm System State Machine.

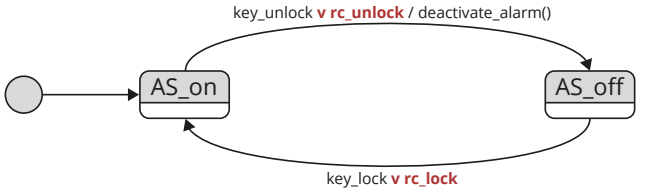

Figure 4. Intermediate Development of the Alarm System State Machine.

references between classes. A model is an instance of a metamodel, i.e., a concrete artifact defined using the notation specified in the metamodel.

Figure 2 shows an exemplary metamodel for state machines [12]. To use the modeling notation defined in a metamodel, one has to create instances of classes, which we refer to as *objects*. The states of a `StateMachine` are defined using the containment reference `states` to the abstract class `AbstractState`. A containment reference is a special type of reference, which denotes that the containing class is owning the contained class, i.e., objects of the contained class only exist within the lifespan of objects of the containing class. A reference has a multiplicity denoting the lower and upper bounds of referenced objects, i.e., the number of objects that can be referenced. The three types of states `State`, `InitialState`, and `EndState` inherit from `AbstractState`. The class `State` additionally has a `name` attribute and if it is a composed state, its sub states are defined using the containment reference `subStates`. An attribute is similar to a reference but instead of referencing objects of another class, it stores values of a primitive type. A `Transition` has three attributes, which define its `triggers`, `guards`, and `actions` as Strings. The target and source state of a transition are defined using respective references to `AbstractState` and the set of outgoing and incoming transitions of an `AbstractState` are defined as opposites to the `Transition`'s references.

2.3 Motivating Example

To meet new requirements or to fix bugs, system behavior needs to change and, thus, the state machines that specify this behavior need to evolve. In the motivating example's first version, the actual intrusion detection is not specified. To integrate this functionality, the state machine has to be extended. However, for large changes, this requires thorough planning and engineers should only be allowed to change parts for which they are responsible.

Before implementing system behavior based on these state machines, it must be ensured that the specified behavior satisfies the requirements to avoid incorrect implementation. Figure 3 shows the planned state machine evolution, which contains intrusion detection and alarm activation logic as sub states of the `AS_on` state. During `Monitoring`, the system reacts to the events `breaking_detected` and `interior_movement` by activating the alarm and its LED. After a timeout, the system stops the alarm and logs the intrusion via the `alarm_detected()` action. The system then remains in the `Silent_Alarm` state where the LED is still active. Additionally, the behavior when unlocking the car is planned to be adapted as the LED has to be deactivated when opening the car.

Enacting planned evolution of a complex system is a large endeavor and may not be performed in an ad-hoc manner. For state machines, the implementation of abstract system behavior can generally be generated but details have to be implemented manually. In the first evolution iteration of the example scenario, the `Monitoring` state with the intrusion detection is implemented first. If this works reliably, the other new states for activating the alarm with the timeout are implemented. Thus, the planned evolution history is successively enacted with each new revision being regarded as present version of the state machine.

In contrast to planned evolution, fixing critical bugs or implementing requirement on short notice results in ad-hoc changes. In the exemplary scenario, a remote control key to lock and unlock the car is introduced shortly before the release of the system. Consequently, another trigger for the transitions between the activated and deactivated alarm system state is added. Figure 4 shows the changed state machine. It must be ensured that these changes do not lead to inconsistencies with the planned changes of Figure 3.

When using a VCS, branches for enacted changes and planned evolution have to be maintained and merged manually. Conflicts between the current state and future plan become apparent only when evolution is being enacted and branches are being merged. Furthermore, short-term changes have to be made compatible with long-term plans immediately upon committing even though more sophisticated changed may have to be planned to still maintain the original plan. With an evolution-aware notation that is capable of planning future evolution, consistency analyses can be performed directly when planning changes and short-term changes may temporarily diverge from long-term plans.

When changing state machines, multiple engineers with different competencies are involved. In the exemplary scenario, engineers who are responsible for locking and unlocking the car implement the triggers for remote control and engineers who are responsible for the alarm system provide the sub states for the AS_on state. To ensure that engineers do not interfere with each other, e.g., in safety or security critical systems, it is important to define responsibilities for different elements of the state machine. With existing techniques that work on file basis, it is not possible to enforce that only the alarm system engineers change the AS_on state. Consequently, intentional or unintentional changes to this state by engineers who do not have the competency to change this system cannot be prevented. With a notation that has a centralized access point to model changes, access control could be implemented without the need of external tools and even on granularity of model elements.

3 Augmenting Metamodels

As highlighted in the motivating example, a model evolution plan is a living artifact where the planned evolution is continuously enacted and consequently becomes the present state. With current methods, e.g., VCSs, no support for future evolution exists, which requires to store planned model evolution as branches or snapshots. This requires to keep planned evolution and active development in sync manually.

We provide a method to precisely store model evolution as continuous time line, which explicitly enables planning of future evolution. We augment a metamodel with concepts for evolution as first-class entity. As basis, we use *temporal elements* [26] with a temporal validity $\vartheta = [\vartheta_{since}, \vartheta_{until})$ - a right-open interval that defines a time span in which the element is temporally valid, i.e., the time span from when it is first conceived to when it is decommissioned. Planning future evolution can be achieved by setting the temporal validity to dates beyond the present. To enact planned evolution, the introduction date of single elements can successively be set to the current point in time. As already performed and planned evolution are stored within the same model, the connection to other evolution steps is captured directly and introducing intermediate changes, such as short-term changes, maintains this relation. This enables to reason on the entire model evolution time line (i.e., past, present, and future) which could be used, e.g., to ensure consistency of short-term changes with planned evolution.

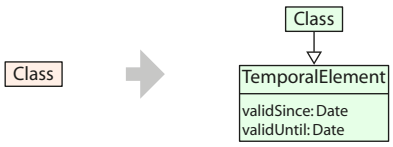

Figure 5. Transformation rule for augmenting metamodels with class evolution.

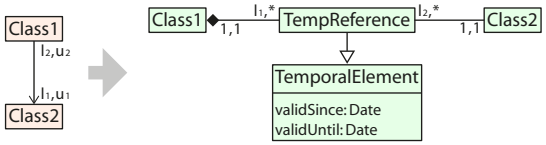

Figure 6. Transformation rule for augmenting metamodels with unordered class-reference evolution.

To augment an arbitrary metamodel with evolution concepts, each element of the metamodel whose instances may be subject to change is extended to resemble a temporal element. This comprises classes, attributes and also references between classes. For this purpose, in the following, we define a set of generic transformation rules to create an evolution-aware metamodel. To ensure practical applicability, we provide transformation rules for all elements of an EMF Ecore metamodel but our concepts are applicable to other MOF metamodel languages [11].

3.1 Augmentation of Classes

To plan the introduction or removal of objects (i.e., instances of classes), it is necessary to be able to define the point in time of their creation or decommission. In the motivating example, the state Monitoring is planned to be created (cf. Figure 3). The respective points in time can be modeled directly using the temporal validity of a temporal element. Thus, a class that should become evolution-aware needs to become a temporal element. Figure 5 shows the transformation rule for classes. After transformation, the class of the original metamodel inherits from the type TemporalElement. Thus, it is possible to express that, during evolution, an object is created by setting its ϑ_{since} or decommissioned by setting its ϑ_{until} respectively. For the metamodel of state machines (cf. Figure 2), this means that the evolution-aware AbstractState inherits from TemporalElement to support state evolution.

3.2 Augmentation of References

References capture the relation between multiple objects of a model. However, these relations may change and new relations are added. In the running example, the new states Monitoring and Alarm are added along with a transition

between them. To connect the states using this transition on model-level, respective values for the metamodel's reference between state and transition is required.

A references in a metamodel is not a separate type and, thus, it is not directly possible to model it as temporal element through inheritance. As remedy, conceptually, we create an association class to capture evolution information. Practically, we create an individual class that wraps the original reference but becomes evolution-aware by inheriting from `TemporalElement`. Figure 6 shows the transformation rule for augmenting a metamodel with reference evolution.

To capture the evolution of this reference, the newly create association class `TempReference` inherits from `TemporalElement`. As, over the course of time, multiple references may have been added or removed, the upper bounds u_1 and u_2 of the original reference have to be relaxed. Consequently, a `Class1` object may contain an arbitrary number of `TempReference` objects and a `Class2` object may be referenced by an arbitrary number of `TempReference` objects.

In the metamodel of the motivating example, references between `Transition` and `AbstractState` specify the source and target state of a transition. Thus, to define an evolution-aware metamodel that enables capturing the evolution of the source state of a transition, a new class `TempSourceState` is introduced that inherits from `TemporalElement`. Additionally, a containment reference between `AbstractState` and `TempSourceState` as well as a reference between `TempSourceState` and `Transition` are added. The original reference between `AbstractState` and `Transition` is removed.

Performing and planning reference evolution is possible by adding a new object of the `TempReference` class with respective dates as temporal validity. For instance, to change the source state of a transition at a *date*, a new `TempSourceState` object is created and its temporal validity is set to $\vartheta = [date, \infty)$. The new object holds a reference to the respective transition and a reference to the new source state. If the transition had a source state before, the temporal validity of the respective `TempSourceState` is set to $\vartheta = [\vartheta_{since}, date)$. To retrieve the source state of a transition for a particular *date*, all references to `TempSourceState` objects are retrieved and they are filtered by removing those objects that are not valid at that respective date, i.e., only keep objects for which $date \in \vartheta$ holds.

3.3 Augmentation of Attributes

In the running example, the trigger for the transition from `AS_on` to `AS_off`, which constitutes an attribute value, is modified as part of the evolution (cf. Figure 4). Augmenting the metamodel with attribute value evolution works analogously to references.

Figure 7 shows the transformation rule for capturing the evolution of values of the attribute `attr`. In contrast to the

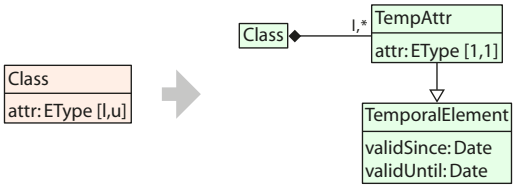

Figure 7. Transformation rule for augmenting metamodels with attribute evolution.

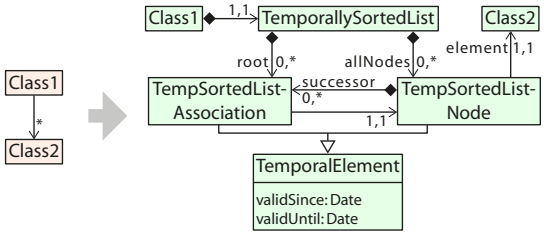

Figure 8. Transformation rule for augmenting metamodels with ordered object-reference evolution.

transformation rule for references, the new association class `TempAttr` does not have a reference to a second class but has an own attribute. The type of the new attribute matches the type of the original metamodel attribute. Each `TempAttr` object represents an (evolved) attribute value. Consequently, the multiplicity of that new attribute is 1. The multiplicity of the original attribute is captured by the multiplicity of the reference to the association class, whereas the upper bound is relaxed again. In the exemplary metamodel for state machines, triggers, guards, and actions of transitions are modeled as attributes of the class `Transition`. To model the evolution of triggers, a new association class `TempTrigger` is created in the evolution-aware metamodel. To capture the evolution mentioned above, a new object of the class `TempTrigger` is created and added in the respective reference of the transition. By relaxing the multiplicity, it is possible to store the trigger before evolution and after evolution in one model. Querying and performing evolution works analogously as for the temporal references.

3.4 Augmentation of Ordered Attributes and References

In the default case, values of multi-valued references and attributes are perceived as sets without a specific order. However, if the order of values is relevant, evolution-aware models need to capture and preserve that order over the course of evolution. To this end, we provide extended transformation rules to make *ordered* attributes and references evolution-aware. For each point in time, we need to have *one* particular order of the list. We realized this on implementation by using linked lists. The list has to be linked for each point in time but the links may change during evolution. Thus, the

list has a set of root elements and each element has a set of successor elements. However, for each single point in time, only one temporally valid root per list and one temporally valid successor element per list node is possible.

Figure 8 shows the transformation rule for ordered reference evolution. For multi-valued attributes, this works analogously. The elements of a `TemporallySortedList` are represented by `TempSortedListNodes` and the links are represented by `TempSortedListAssociations`. As elements may be added to or removed from such a list, the `TempSortedListNode` inherits from `TemporalElement`. When adding or removing elements, new links have to be established and, thus, the `TempSortedListAssociation` inherits from `TemporalElement` as well. As the list order is subject to evolution as well, elements may have different successors at different points in time. Thus, each `TempSortedListNode` has a set of successors represented by the reference to `TempSortedListAssociation`. The same applies to the root element of the list. Note that we realized the `TemporallySortedList` using generic types, i.e., the type for a list's entries is determined by setting the generic type.

3.5 Non-Augmented Elements

For different reasons, some elements of the original metamodel do not need specific augmentation. Attributes and references may be marked with additional properties, some of which preclude augmentation: In EMF Ecore, the value of non-changeable elements cannot be altered, that of volatile and transient elements is not saved, and that of derived elements is calculated only on demand but has no own identity. In consequence, the respective values are not subjected to evolution in a way that necessitates storing an evolution history. Furthermore, operations may conceptually be subjected to evolution. However, we provide an engineering solution that makes it superfluous to augment individual operations, which we elaborate on in Subsection 4.2.

4 Automatic Generation of Access Layers

In Section 3, we introduced transformation rules to augment a metamodel for a derived notation capable of storing evolution information. However, for a practical application of our method, we have identified three problems that need to be addressed: First, if transformation rules had to applied manually, the procedure to derive an evolution-aware metamodel would be very tedious. Second, while technically possible to alter and query evolution information directly within evolution-aware models (e.g., for analyses or tools), accessing this information is cumbersome due to the complex structure of the augmented metamodel. Third, as our procedure creates a new metamodel for the evolution-aware notation, even though essential, compatibility with an existing tool landscape of the original notation is threatened.

We provide three measures to address these problems: First, we automate the application of the aforementioned transformation rules as part of a generative procedure to augment arbitrary metamodels, which lowers the adoption barrier of our procedure. Second, we demonstrate how to automatically create a centralized access point for evolution, which hides complex intricacies of evolution-aware models both for performing evolution and querying evolution information. Third, we devise a method to automatically generate an adapter infrastructure, which preserves compatibility with the original metamodel and automatically tracks performed changes as evolution. With these contributions, we lower the barrier for both adoption and usage of evolution-aware models. Figure 9 shows an overview of the generated structure, which we use in the following to elaborate on the three measures.

4.1 Generating Evolution-Aware Metamodels

In Section 3, we deliberately formulated the transformation rules in a way that makes them suitable for automated application. In consequence, augmenting a metamodel in its entirety is a matter of repeatedly applying the transformation rules suitable for each respective element.

However, it is not necessarily the case that all model elements defined in a metamodel are subject to evolution. Naturally, both the procedure of augmenting a metamodel and the creation/maintenance of evolution-aware models entail a cost in the form of increased runtime and memory usage. To reduce this cost for model elements that will not be subjected to evolution, we enable tailoring the generation procedure by deselecting elements of the provided metamodel. For each of the selected elements, the respective transformation rules are applied automatically to augment them. For each of the deselected elements, the original, non evolution-aware, elements are used in the augmented metamodel. The result of our automated generative procedure is an evolution-aware metamodel which enables to track, plan and analyze arbitrary evolution integrated with its respective models. Figure 9 shows the generated structure of an exemplary metamodel ① with one class (`OMMElement`) and one attribute (`name`). The right side shows the generated code of the evolution-aware metamodel after applying our transformation rules ②.

4.2 Seamless Usage of Evolution-Aware Models

Planning and performing evolution can be done by accessing the evolution-aware models directly. However, this is cumbersome due to the complex nature of the evolution-aware metamodel. For instance, to access the name of an `EvAMMElement` of Figure 9 at a particular date, users have to retrieve the set of all related evolution-aware `TANames` and iterate over it to determine the name that is temporally valid at that date (cf Section 3). This procedure is complicated even for a small example but becomes even more problematic when considering that users may be familiar with the

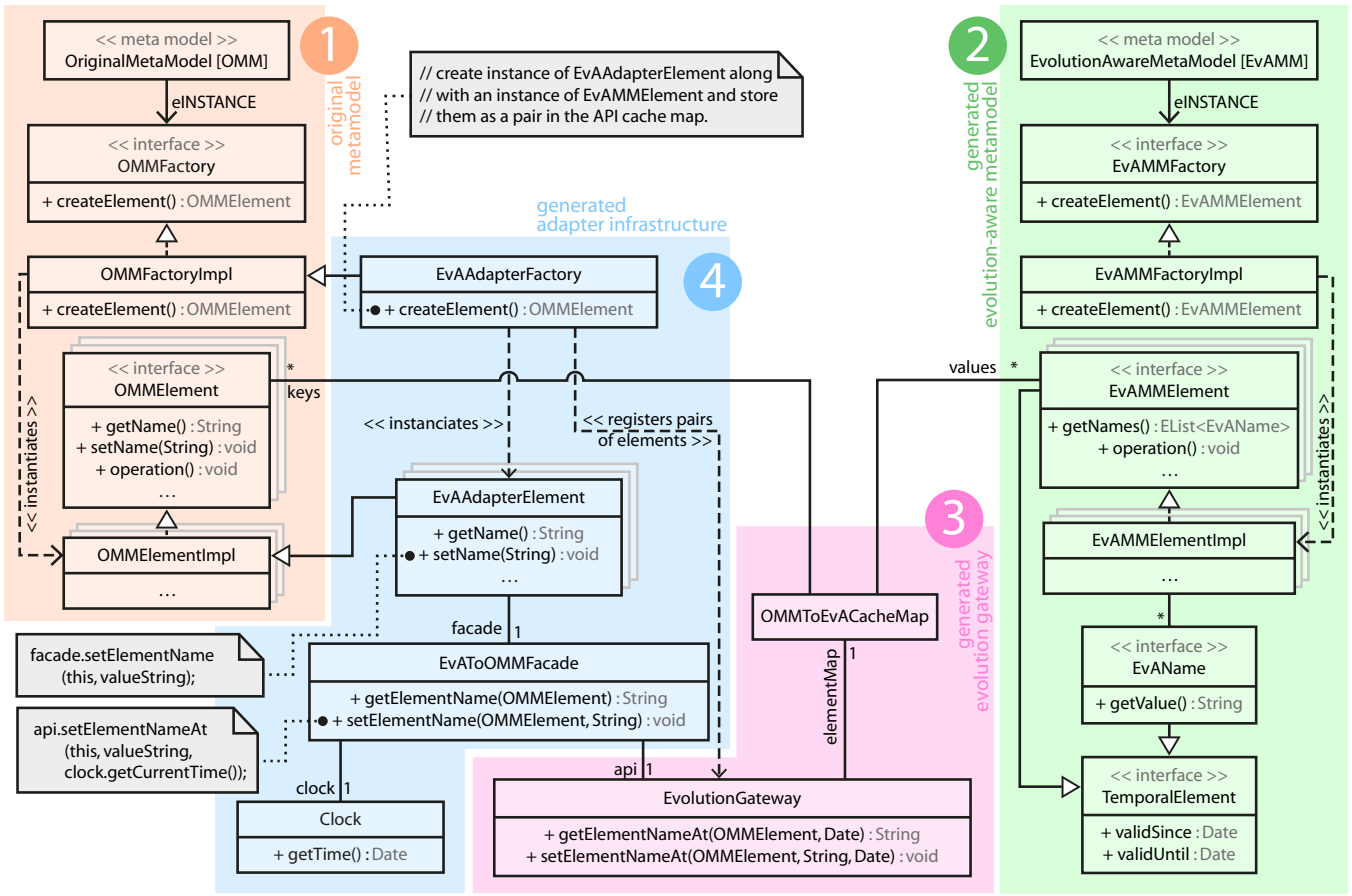

Figure 9. Generated code from original ① and respective evolution-aware metamodel ②. Exemplary for one class and one attribute. Evolution gateway for centralized access to evolution-aware models ③. Adapter infrastructure for transparent access using original metamodel interfaces ④.

structure of the original metamodel but not the augmented metamodel. Additionally, as our augmentation procedure yields a new metamodel, compatibility problems may arise with the infrastructure of the original metamodel, e.g., editors. Consequently, existing tools could not be used as-is to plan and perform model evolution. This results in high adaptation effort and, even worse, adaptation may not be possible if third-party tools are used.

To remedy these problems and to increase acceptance of evolution-aware models, we have devised a procedure that can create compatibility between the augmented and the original metamodel by automatically generating a suitable *adapter infrastructure*. To further ensure that this compatibility allows seamlessly using evolution-aware models instead of the original models even in existing implementations, it is paramount that interfaces of the original model classes are maintained. Figure 9 ④ shows an exemplary adapter infrastructure we generate. For each class of the original metamodel (`OMMElementImpl`), we generate an adapter class (`EvAAdapterElement`) that serves as proxy for accessing the data stored in the evolution-aware model. This class inherits from the original metamodel class but overrides each method for reading or writing data, i.e., getter and setter methods.

For planning and performing evolution, it is necessary to specify a date for which changes are to be performed. As this general procedure is similar for all evolution-aware classes of a metamodel, we introduce a facade [9] `EvAToOMMFacade` that hides these details from the adapters [9]. The facade provides mirrored methods for each method defined in the adapter classes, i.e., setters and getters for each attribute or relation. In the adapter classes, the method calls are delegated to this facade. The facade retrieves the date for applying the changes from a central `Clock`. As default, we provide a clock that uses the system's current time. We provide an override mechanism using the generation gap pattern [33], which enables supplying custom clocks, e.g., a date picker. The actual execution of changes is delegated to the `EvolutionGateway` which is introduced in the next section.

When creating a model of the original metamodel, e.g., using an editor, we need to ensure that instead of creating

original model classes, we create adapter classes. To this end, we make use of the generation gap pattern [33] and override the existing factory (`OMMFactoryImpl`), which is responsible for creating entities, with a factory (`EvAAdapterFactory`) that creates adapter entities instead. Thus, with the adapters and the overridden factory, users can transparently use the evolution-aware model without changing anything in the implementation. As a result, changes performed to the model are automatically tracked as evolution history. Additionally, providing support for planning evolution operations is simple as only the clock has to be overridden, e.g., by using a graphical element to pick a date.

4.3 Accessing Evolution-Aware Models

While the facade of the adapter infrastructure (cf. Subsection 4.2 and ③) enriches the method calls of the adapters with the date of the clock, the actual execution of the evolution operations still needs to be performed. To this end, we provide a method to generate a layer that connects the adapter infrastructure that is only aware of the original metamodel with the data structure of the evolution-aware metamodel. We denote this connecting layer `EvolutionGateway` (see ④). The `EvolutionGateway` is implemented using the facade pattern [9] and provides methods to access evolution information and perform evolution operations on each model element. To this end, a getter and a setter method for each attribute or model element relation is defined, similar to the facade (cf. Subsection 4.2), and the respective methods are performed for a given date. As the adapter infrastructure is only aware of the original metamodel, the parameters in the methods are elements from the original metamodel. For instance, in Figure 9, the illustrated method `setElementNameAt` sets the name for the given element at the given date. Note that the element provided to that method has the type `OMMElement`, i.e., an element matching the original metamodel. In our case, this is an object of type `EvAAdapterElement`.

Conceptually, the view for users of the generated system consists of elements resembling instances of the original metamodel (see ①). Technically, the view of the system utilizes instances of the augmented metamodel (see Figure 9 ②). Hence, operations of `EvolutionGateway` are called via methods that utilize types of the original metamodel for their parameters but are executed upon elements of the evolution-aware model. To translate between these two representations, the `EvolutionGateway` holds and maintains a map (`OMMToEvACacheMap`) that relates adapter entities matching the original metamodel to entities of the evolution-aware model, i.e., the entities that store the actual evolution information. For each method call, the `EvolutionGateway` initially retrieves the evolution-aware model element `EvAMMElement` that is mapped to the original model element `OMMElement`.

For retrieving evolution information, the `EvolutionGateway` provides getter methods that retrieve the values that are temporally valid at the given date. If the respective attribute or relation in the original metamodel was a single value, i.e., the upper bound is 1, the `EvolutionGateway` also returns a single value. Similarly, for multi-valued attributes and references, the `EvolutionGateway` returns a collection holding elements of the respective type. In both cases, the method is performed in three steps. First, the evolution-aware model element that is mapped in the `OMMToEvACacheMap` to the given original model element is retrieved. Second, the set of all value elements is queried from the evolution-aware model element, which comprises all value elements that have ever been temporally valid or that will ever be temporally valid. Third, the set of value elements is filtered based on the elements' temporal validity regarding the given date. For instance, in Figure 9, the method `getElementNameAt` retrieves the `EvAMMElement` that is mapped to the given `OMMElement`. Then, all `EvANames` related to the `EvAMMElement` are retrieved. Finally, it is checked which of the `EvANames` is temporally valid at the given date and the result of the valid `EvAName`'s `getValue` is returned.

Planning and performing evolution operations can be achieved using the setter methods of the `EvolutionGateway`. This is a four-step process: First, the evolution-aware model element is retrieved. Second, the value element that is valid at the given date is retrieved and the end of its temporal validity is set to the given date, i.e., ϑ_{until} = `date`. Third, a new value element with the given value is created, its temporal validity is set to begin at the given date and end at the date at which the element of the second step originally ended. Fourth, the new value element is added to the set of all value elements. For instance, in the method `setElementNameAt`, the currently valid `EvAName` of the `EvAMMElementImpl` is retrieved. The end of this name's temporal validity is set to `date`. Then, a new `EvAName` is created and its value is set to the given `String` parameter. Additionally, the beginning of the temporal validity of the new `EvAName` is set to `date`. Finally, the new `EvAName` is added to the set of names of the `EvAMMElementImpl`. Note that, in Ecore, an entity is deleted using a utility method which removes the respective entity from all references. Due to this, it is undecidable whether an entity is temporarily removed from all references, e.g., to move it in the model, or whether it is deleted. Thus, the `EvolutionGateway` provides a distinguished method to delete elements, i.e., setting the end of their temporal validity.

Changes to multi-valued attributes and relations are performed on the collections that are returned using getter methods. However, the `EvolutionGateway` returns collections with elements of the original metamodel and, thus, without information on evolution. Consequently, changes to that collection would not be transferred to the evolution-aware model and inconsistencies would

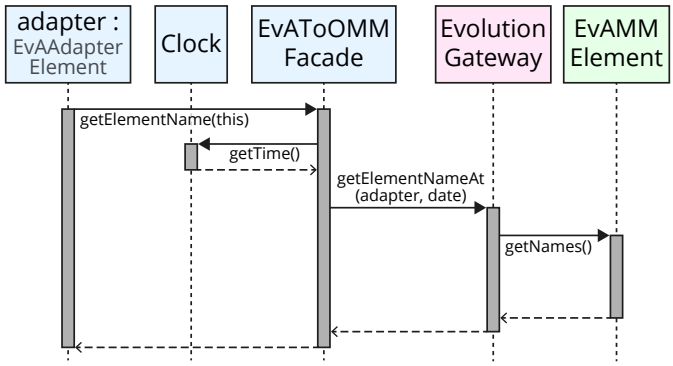

Figure 10. Sequence of exemplary method call on adapter class with usage of facade, evolution gateway, and evolution-aware metamodel.

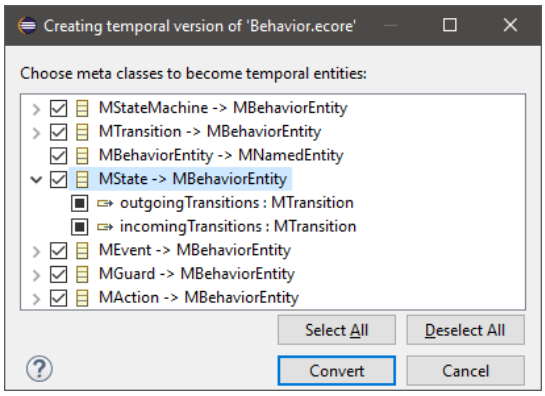

Figure 11. Wizard of the *TemporalRegulator3000* for a state machine metamodel.

arise. To address this problem, the `EvolutionGateway` enables users performing evolution on these sets by providing explicit methods for modifying multi-valued attributes and references, which results in changes to the data of the evolution-aware model. For instance, for an augmented metamodel for state machines, adding an outgoing transitions to a state would work using the following method `addOutgoingTransitionsAt(State state, List<Transition> transitionsToAdd, Date date)`. Note that the gateway also retrieves and maintains the correct order of lists using the `TemporallySortedLists` if necessary (cf. Subsection 3.4) and automatically maintains opposite references. However, to ensure that these methods are called and to prevent the aforementioned inconsistencies, the `EvolutionGateway` returns immutable collections and the adapter infrastructure wraps these collections with an observer pattern, which is used to delegate modification operations via the facade to the `EvolutionGateway`.

Figure 10 shows an exemplary chain of method calls from the adapter infrastructure to the evolution-aware model for the method `getName` on the `EvAAdapterElement`. First, the adapter delegates the call to the facade. The facade retrieves the date from the `Clock` and delegates the method call to the `EvolutionGateway` using that date. Finally, the `EvolutionGateway` accesses the evolution-aware model.

With the evolution gateway, we provide a centralized access point to evolution information without the need for handling the complex structure of the evolution-aware metamodel. The means for access resemble those of the original metamodel, as we use similar signatures of methods. With the additional information of the date, we are able to perform evolution and retrieve information pertaining evolution. Thus, we provide an easy entry to use evolution-aware modeling notations. Additionally, a centralized access point for performing evolution opens up new possibilities, e.g., access control for performing evolution could be integrated to allow accessing selected model elements only to specific persons.

5 Evaluation

A complex software system is comprised of a multitude of different notations, e.g., UML class diagrams for modeling the system and Java source code for its implementation. Furthermore, safety-critical systems commonly use declarative notations to aid the certification process, e.g., Software Fault Trees (SFTs) [21], and, if the system is developed in the sense of a software product line [28], a variability model may describe its configuration options, e.g., as a feature model [16]. To show applicability of our method, we evaluate different aspects of metamodel augmentation on model-based representations of all these notations. In particular, we show that it is possible to capture and access model evolution timelines while preserving compatibility with existing tools. Additionally, we show that our method provides a basis for restricting model access. As a first step, we present the implementation of our augmentation method and then we present the four research questions that guide our evaluation.

5.1 Implementation

The *TemporalRegulator3000* is the implementation of our augmentation method. It can be found in our online repository.[1] The tool generates all parts described in this paper, i.e., the augmented metamodel and the different access layers. As input, it uses arbitrary EMF Ecore metamodels and it guides users with a visual wizard through the generation of the augmented metamodel. As the augmentation can lead to significantly larger (meta)models (cf. **RQ4**), users may choose which elements of the original metamodel should be augmented to capture their evolution. After selecting the elements for augmentation, the *TemporalRegulator3000* generates the respective augmented metamodel by applying the transformation rules presented in Section 3. For instance, Figure 11 shows the wizard for an exemplary state machine metamodel together with its classes and some references which are selected for augmentation.

[1] https://gitlab.com/Adomat/temporalregulator3000

By running the *TemporalRegulator3000*, a new Eclipse plug-in containing the augmented metamodel and its generated source code is created. At runtime, this code overrides the original metamodel code. Consequently, all tools using the original metamodel use the new code instead – without the need for any changes (cf. **RQ2**).

5.2 Research Questions

With our method, we provide means to store the entire evolution timeline of a model. To evaluate, whether the metamodel augmentation is powerful enough to address our previously mentioned challenges, we pose four research questions.

RQ1 Is the augmented notation able to track, plan, and analyze model evolution in one artifact?

RQ2 Is transparent use of the evolution-aware model possible while preserving compatibility with existing tools?

RQ3 Are we able to restrict model access without external tools?

RQ4 Is the metamodel augmentation applicable to large-scale metamodels?

To answer the research questions, we employ the *TemporalRegulator3000* to create augmented metamodels of the above mentioned notations. For each research question, we use a different notation to show the flexibility of our method. In the following, we describe the experiments and results.

RQ1 We utilize the evolution history of real-world feature models [17] of a financial company.[2] The feature model's evolution history comprises ten versions and the feature model has between 557 (version 1) and 774 (version 10) features. To capture the entire evolution timeline of the feature model, we extended the tool for importing multiple model evolution snapshots into one augmented model.

We successfully imported all versions into one single evolution-aware model. By setting the `Clock`'s date to a future point in time, we imported multiple versions as planned evolution steps. While performing this evaluation, we extended the *TemporalRegulator3000* by a generic function to import multiple versions / snapshots of a model into one evolution-aware model which only requires to implement a comparator for all metamodel elements. We verified that we correctly imported all versions by reproducing each version from the evolution-aware model. In particular, we set the `Clock` to the date for which we imported the original version and then compared both models. In summary, we were able to track and plan model evolution in one artifact.

Regarding the analyses of evolution-aware models, we investigated which feature groups changed frequently and which groups had many features added. The features in these groups seem to be the less stable ones and potentially need a lot of testing. By using the information stored in the evolution-aware model, this analysis was straight-forward to realize. For each group, we checked the association elements for the relation to its child features (cf. Figure 6) and their temporal validity. As we only want to show feasibility, we present numbers for the most relevant groups. We identified two groups to which 36 features were added in two evolution steps and three groups were changed in 6 different evolution steps. Thus, evolution-aware models provide ample data that can be used for sophisticated analyses regarding evolution.

As the augmented metamodel introduces additional complexity, modeling a single feature version based on the augmented metamodel results in significantly increased model size. However, when considering multiple versions, the number of model elements is even reduced for the evolution-aware model. This is due to the fact that elements which are available in more than one version are part of each model, i.e., they exist in multiple snapshots. For instance, a feature that exists in all ten versions also exists in each model snapshot. In the evolution-aware model, this feature only exists once with a temporal validity indicating for which timespan it exists. However, by using `TemporallySortedLists` (cf. Figure 8), the number of objects increases again.

To give an estimate, how evolution-aware models grow compared to the orignal models, we compared both sizes for the feature model timeline. The sum of all objects in the feature model versions is 8,835, the sum of all reference values is 17,640, and the sum of attribute values is 24,837. For the evolution-aware model, the number of objects is 14,456 (164%), the number of reference values is 20,111 (114%), and the number of attribute values is 18,503 (75%). Especially, the usage of `TemporallySortedLists` leads to a high increase of objects. However, we assume that the evolution-aware model will become smaller than the versions if a long evolution timeline is captured.

RQ2 Backwards compatibility with the original metamodel notation is pivotal for acceptance of the augmented metamodels. Changes performed with existing tools, such as editors, need to be stored as evolution in the augmented model. To this end, we generate the adapter infrastructure, which serves as proxy for accessing data of the evolution-aware model (cf. Subsection 4.2). To evaluate the adapters' compatibility, we generate an augmented notation and investigate whether existing tooling still works. In particular, we utilize a metamodel for Software Fault Trees (SFTs) and verify whether the existing editor still works. To set the date for the central clock, we provide a simple UI of a date picker.

The existing editor for SFTs still worked without any adaptations. Using the implemented UI interface, we set the the clock's date to May 22, 2019. We created a simple SFT with two faults (`RF1`, `F1`) and one gate (`G`). Figure 12 shows the editor with the SFT together with the simple UI to set the clock's date. After that, we performed model evolution by

[2] https://github.com/FeatureIDE/FeatureIDE/tree/develop/plugins/de.ovgu.featureide.examples/featureide_examples/FeatureModels/FinancialServices01

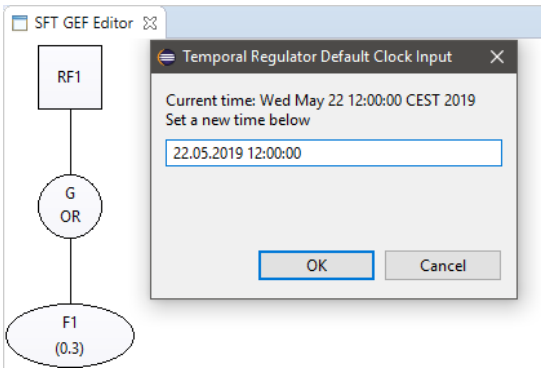

Figure 12. Screenshot of the SFT editor before evolution.

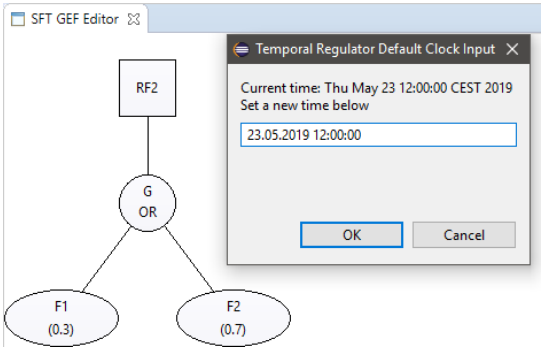

Figure 13. Screenshot of the SFT editor after evolution.

setting the clock's date to May 23, 2019. We added a new fault (F2) under the already existing gate. Figure 13 shows the editor and the clock UI for this evolved model. When changing the clock back to May 22, 2019, the editor showed the model before evolution again. Consequently, we are able to seamlessly integrate an augmented metamodel with existing tooling. Additionally, the evolution timeline is also stored using existing editors which enables performing analyses regarding evolution without the need for new editors.

RQ3 As motivated earlier, access control can only be achieved on file basis and by using third-party tools. With our method, it is possible to provide fine-grained access control on basis of individual model elements. To evaluate this, we utilize the augmented metamodel for SFTs, which we also used to answer **RQ2**. To realize such an access control, the centralized access in the `EvolutionGateway` (cf. Subsection 4.3) is modified to prohibit certain evolution operations. In particular, we disallow evolution operations modifying faults with a name containing "_critical".

Using the editor shown in Figure 12, we created a fault with name `F3_critical`. Then, we activated the access control mentioned above. Using the editor, we tried to change the name of that respective fault. In the `EvolutionGateway`, we restricted this by dropping change requests to such faults. Consequently, our actions using the editor did not result in any modifications to the SFT. Adding new faults and renaming them still worked as intended. Thus, we provide the basis for implementing a fine-grained access control on element level. Using this as basis, a more sophisticated rights management system for users and error reporting could be realized.

Table 1. Metamodel elements before and after augmentation.

		EClasses	EReferences	EAttributes
JaMoPP	Original	237	105	15
	Augmented	303	270	14
UML	Original	243	510	115
	Augmented	710	1167	109

RQ4 After we evaluated the functionality of our method, we verify whether the augmentation works for large-scale real-world metamodels. To this end, we utilize two metamodels of languages used in industry. In particular, we generate augmented metamodels for the official UML2 metamodel from Ecore and for the JaMoPP metamodel for Java [13].

We were able to augment the real-world metamodels for UML2 and JaMoPP. When augmenting metamodels to capture the evolution timeline, they significantly grow. Table 1 shows the size of the original metamodel compared to the augmented metamodel for UML2 and JaMoPP. The number of classes grows as augmented references and attributes are transformed into association classes (cf. Figures 6, 7). Additionally, for each new association class, new references to the source and target class of the original reference are necessary. Consequently, the number of references increases as well.

Note that the increase in number of classes is not the same as the sum of references and attributes of the original metamodel. Moreover, the number of attributes even decreased. This is due to the fact that we use `TemporallySortedLists` with generic types to augment ordered references and attributes (cf. Figure 8). Thus, for all ordered references and attributes of the same type, we only need to add one new `TemporallySortedList` with the respective generic type.

To give an estimate how large an augmented metamodel grows, we assume the worst case, i.e., no `TemporallySortedLists` can be used. Consequently, for each reference, a proxy class is introduced together with two references to the source and target class of the original reference. For each attribute, an association class and a reference to that class from the original class is required. As in the augmented metamodel, the attribute is moved from the original class to the association class, the number of attributes does not change. Consequently, the increase in number of elements (classes and references) is $O(3r+2a)$ with r being the number of references in the original metamodel (opposite references not included) and a being the number of attributes in the original metamodel. Note that the number of classes in the original metamodel does not have any impact on the number of elements in the augmented metamodel.

6 Related Work

There are multiple approaches that create evolution histories via a sequence of applied operations: Operation-based VCSs [20, 25] store evolution in version-control system by capturing the modifications that were performed instead of the state resulting from those modifications. Change-oriented programming [6] uses change objects synthesized from applied operations to represent evolutionary modifications on implementation artifacts. Engels et al. [7] and Kehrer et al. [18] capture model evolution by means of predefined operations or transformation scripts. The goal is to preserve model consistency between each evolution step.

Unlike the approaches based on specific operations, we store model evolution internal to implementation artifacts as temporal validities, which does not limit users to a constrained set of potential operations. Additionally, our method seamlessly integrates with existing tool infrastructure and, in addition, we are able to perform planning of future evolution, which can be enacted seamlessly. Nonetheless, a combination of methods to preserve model consistency [7, 18] with our method can be sensible.

Degueule et al. provide a method to enable model polymorphism [5]. Their language workbench *Melange* generates an adapter infrastructure for Ecore models that enables engineers to provide different interfaces (i.e., DSLs) to manipulate a single model. Similarly, we also enable model polymorphism using a generated adapter infrastructure but we focus on storing model evolution timelines.

Bousse et al. introduce trace metamodels which enable capturing of model execution traces which are similar to model evolution [1]. The generated domain-specific trace metamodels resemble the evolution-aware metamodels that are generated in this paper. In contrast to our method, Bousse et al. explicitly aim for efficient storage of these traces but do not consider seamless integration with tooling for existing metamodels. Concepts of this method could improve storage and handling of model evolution information.

Many researchers addressed the problem of metamodel-model co-evolution [4, 10, 14, 19, 29, 34]. Similar to the above mentioned methods, these methods define operations to modify models. The focus is to preserve compatibility of existing models with evolved metamodels. However, with these approaches, it is neither possible to capture model evolution history nor plan its future evolution.

Hermannsdörfer et al. propose a generic operation recorder for model evolution [15]. Similar to our method, the goal is to capture model evolution without the need to adapt existing tools. To this end, the operation recorder utilizes the observer mechanism of the EMF environment. Changes to EMF models are then captured in a separate generic operation model. To retrieve a certain model version, these operations need to be applied. Thus, existing tools like analyses or viewers cannot seamlessly access specific model versions.

In other fields of research, such as differencing and merging of models versions [2, 3, 8, 32] and reverse engineering of model changes [35, 36], differences between multiple model (versions) are retroactively computed. These methods work with existing VCSs, but computing the differences is expensive and often only an approximation. Additionally, seamless integration with existing tool infrastructure is not considered and planning model evolution is out of scope.

To the best of our knowledge, no method exists which considers access control on model element level. Additionally, most of the previously mentioned methods (except for the operation recorder [15]) do not seamlessly integrate with existing tool infrastructure.

7 Conclusion and Future Work

We presented a generic method for integrated storage of past history and planned evolution of models by augmenting metamodels. Our augmentation rules can be applied to arbitrary metamodels and we provide concepts for generating an infrastructure for accessing and writing evolution-aware models. Thus, we are able to store an entire model evolution timeline in one artifact while seamlessly integrating with existing tool infrastructure. Additionally, we provide a centralized access point to model evolution which can be used to realize a fine-grained access control on model element level without the need of third-party tools.

The results we present in this paper form the basis for planning of future model evolution. However, planning long-term goals for a model still requires to insert intermediate evolution steps. As a result, inconsistencies with the already planned evolution may arise (aka *evolution paradoxes*) [27]. With this work, we do not solve the problem of evolution paradoxes but provide the first step towards consistent model evolution planning. Additionally, evolution-aware constraints may be a relevant field of research, i.e., constraints that define how a model may or must evolve.

As shown in the evaluation, evolution-aware models significantly grow in size. To address this issue, in our future work, we will investigate the slicing of a model timeline into chunks. This requires to keep all relevant information in each chunk so that each chunk is consistent and would help in maintaining especially long evolution histories.

Acknowledgments

This work was supported by the Federal Ministry of Education and Research of Germany within CrESt (01IS16043S) and by the DFG (German Research Foundation) under SPP1593: Design For Future — Managed Software Evolution.

References

[1] Erwan Bousse, Tanja Mayerhofer, Benoit Combemale, and Benoit Baudry. 2019. Advanced and efficient execution trace management for executable domain-specific modeling languages. *Software & Systems Modeling* 18, 1 (01 Feb 2019), 385–421.

[2] Petra Brosch, Gerti Kappel, Philip Langer, Martina Seidl, Konrad Wieland, and Manuel Wimmer. 2012. An Introduction to Model Versioning. In *Proceedings of the 12th Intl. Conference on Formal Methods for the Design of Computer, Communication, and Software Systems: Formal Methods for Model-driven Engineering*. Springer-Verlag, Berlin, Heidelberg, 336–398.

[3] Cédric Brun and Alfonso Pierantonio. 2008. Model differences in the eclipse modeling framework. *UPGRADE, The European Journal for the Informatics Professional* 9, 2 (2008), 29–34.

[4] A. Cicchetti, D. D. Ruscio, R. Eramo, and A. Pierantonio. 2008. Automating Co-evolution in Model-Driven Engineering. In *2008 12th Intl. IEEE Enterprise Distributed Object Computing Conference*. 222–231.

[5] Thomas Degueule, Benoit Combemale, Arnaud Blouin, Olivier Barais, and Jean-Marc Jézéquel. 2017. Safe model polymorphism for flexible modeling. *Computer Languages, Systems & Structures* 49 (2017), 176–195.

[6] Peter Ebraert, Jorge Vallejos, Pascal Costanza, Ellen Van Paesschen, and Theo D'Hondt. 2007. Change-Oriented Software Engineering. In *Proceedings of the Intl. Conference on Dynamic Languages*. ACM.

[7] Gregor Engels, Reiko Heckel, Jochen M. Küster, and Luuk Groenewegen. 2002. Consistency-Preserving Model Evolution through Transformations. In *UML 2002 — The Unified Modeling Language*, Jean-Marc Jézéquel, Heinrich Hussmann, and Stephen Cook (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 212–227.

[8] Eclipse Foundation. 2019. *EMF Compare Project*. Retrieved June 6, 2019 from http://www.eclipse.org/emf/compare

[9] Erich Gamma. 1995. *Design patterns: elements of reusable object-oriented software*. Pearson Education India.

[10] Kelly Garcés, Frédéric Jouault, Pierre Cointe, and Jean Bézivin. 2009. Managing Model Adaptation by Precise Detection of Metamodel Changes. In *Model Driven Architecture - Foundations and Applications*, Richard F. Paige, Alan Hartman, and Arend Rensink (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 34–49.

[11] Object Management Group. 2016. *OMG Meta Object Facility (MOF) Core Specification*. Technical Report. Object Management Group.

[12] David HAREL. 1987. STATECHARTS: A VISUAL FORMALISM FOR COMPLEX SYSTEMS. *Science of Computer Programming* 8 (1987), 231–274.

[13] Florian Heidenreich, Jendrik Johannes, Mirko Seifert, and Christian Wende. 2010. Closing the Gap between Modelling and Java. In *Software Language Engineering*, Mark van den Brand, Dragan Gašević, and Jeff Gray (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 374–383.

[14] Markus Herrmannsdoerfer, Sebastian Benz, and Elmar Juergens. 2009. COPE - Automating Coupled Evolution of Metamodels and Models. In *ECOOP 2009 – Object-Oriented Programming*, Sophia Drossopoulou (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 52–76.

[15] Markus Herrmannsdoerfer and Maximilian Koegel. 2010. Towards a Generic Operation Recorder for Model Evolution. In *Proceedings of the 1st Intl. Workshop on Model Comparison in Practice*. ACM, New York, NY, USA, 76–81.

[16] K.C. Kang, S.G. Cohen, J.A. Hess, W.E. Novak, and A.S. Peterson. 1990. *Feature-Oriented Domain Analysis (FODA) Feasibility Study*. Technical Report. DTIC Document.

[17] Kyo C Kang, Sholom G Cohen, James A Hess, William E Novak, and A Spencer Peterson. 1990. *Feature-oriented domain analysis (FODA) feasibility study*. Technical Report. Carnegie-Mellon Univ Pittsburgh Pa Software Engineering Inst.

[18] T. Kehrer, U. Kelter, and G. Taentzer. 2013. Consistency-preserving edit scripts in model versioning. In *2013 28th IEEE/ACM Intl. Conference on Automated Software Engineering (ASE)*. 191–201.

[19] Wael Kessentini, Houari Sahraoui, and Manuel Wimmer. 2016. Automated Metamodel/Model Co-evolution Using a Multi-objective Optimization Approach. In *Proceedings of the 12th European Conference on Modelling Foundations and Applications*. Springer-Verlag, Berlin, Heidelberg, 138–155.

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

[21] Nancy Leveson and Steve Goetsch. 1996. Safeware: System Safety and Computers. *Medical Physics-New York-Institute of Physics* 23, 10 (1996), 1821.

[22] Sascha Lity, Remo Lachmann, Malte Lochau, and Ina Schaefer. 2013. *Delta-oriented Software Product Line Test Models - The Body Comfort System Case Study*. Technical Report. TU Braunschweig.

[23] J. Mendling, H.A. Reijers, and W.M.P. van der Aalst. 2010. Seven process modeling guidelines (7PMG). *Information and Software Technology* 52, 2 (2010), 127 – 136.

[24] Sophia Nahrendorf, Sascha Lity, and Ina Schaefer. 2018. *Applying Higher-Order Delta Modeling for the Evolution of Delta-Oriented Software Product Lines*. Technical Report. TU Braunschweig.

[25] Tung Thanh Nguyen, Hoan Anh Nguyen, Nam H. Pham, and Tien N. Nguyen. 2010. Operation-Based, Fine-Grained Version Control Model for Tree-Based Representation. In *Fundamental Approaches to Software Engineering*, David S. Rosenblum and Gabriele Taentzer (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 74–90.

[26] Michael Nieke, Christoph Seidl, and Sven Schuster. 2016. Guaranteeing Configuration Validity in Evolving Software Product Lines. In *Proceedings of the Tenth Intl. Workshop on Variability Modelling of Software-intensive Systems*. ACM, New York, NY, USA, 73–80.

[27] Michael Nieke, Christoph Seidl, and Thomas Thüm. 2018. Back to the future: avoiding paradoxes in feature-model evolution. In *Proceeedings of the 22nd Intl. Systems and Software Product Line Conference - Volume 2, Gothenburg, Sweden, September 10-14, 2018*. 48–51.

[28] Klaus Pohl, Günter Böckle, and Frank J. van der Linden. 2005. *Software Product Line Engineering - Foundations, Principles and Techniques*. Springer Berlin/Heidelberg.

[29] Louis M. Rose, Dimitrios S. Kolovos, Richard F. Paige, and Fiona A. C. Polack. 2010. Model Migration with Epsilon Flock. In *Theory and Practice of Model Transformations*, Laurence Tratt and Martin Gogolla (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 184–198.

[30] Ina Schaefer, Rick Rabiser, Dave Clarke, Lorenzo Bettini, David Benavides, Goetz Botterweck, Animesh Pathak, Salvador Trujillo, and Karina Villela. 2012. Software diversity: state of the art and perspectives. *STTT* 14, 5 (2012), 477–495.

[31] D. Spinellis. 2005. Version control systems. *IEEE Software* 22, 5 (Sep. 2005), 108–109.

[32] Gabriele Taentzer, Claudia Ermel, Philip Langer, and Manuel Wimmer. 2014. A fundamental approach to model versioning based on graph modifications: from theory to implementation. *Software & Systems Modeling* 13, 1 (01 Feb 2014), 239–272.

[33] John Vlissides and John M Vlissides. 1998. *Pattern hatching: design patterns applied*. Addison-Wesley Reading.

[34] Guido Wachsmuth. 2007. Metamodel Adaptation and Model Co-adaptation. In *ECOOP 2007 – Object-Oriented Programming*, Erik Ernst (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 600–624.

[35] David Wille, Tobias Runge, Christoph Seidl, and Sandro Schulze. 2017. Extractive Software Product Line Engineering Using Model-based Delta Module Generation. In *Proceedings of the Eleventh Intl. Workshop on Variability Modelling of Software-intensive Systems*. ACM, New York, NY, USA, 36–43.

[36] Z. Xing and E. Stroulia. 2006. Refactoring Detection based on UMLDiff Change-Facts Queries. In *13th Working Conference on Reverse Engineering*. 263–274.