Access Granted – Carefully: Securing Model Information in Collaborative Modeling

Malvina Latifaj[a], Federico Ciccozzi[a], Antonio Cicchetti[a]

[a]*Mälardalen University, , Västerås, 722 20, Sweden*

Abstract

The collaborative nature of model-driven software engineering introduces significant challenges in safeguarding the confidentiality and integrity of the collaborative model. Existing access control mechanisms often rely on transient, virtual views lacking persistence and fine-grained permissions, making them unsuitable for scenarios requiring offline collaboration and leading to potential security breaches and user frustration. This work describes a dual-layered approach leveraging role-based access control policies to enhance security in collaborative modeling environments. The first layer utilizes multi-view modeling techniques to create materialized view models tailored to specific user roles, thereby restricting unnecessary access to the entire model. The second layer refines access at the individual element level within these view models, establishing fine-grained permissions enforced by model editors. This proactive enforcement prevents unauthorized actions before they occur, improving user experience and efficiency. The proposed approach, implemented as an Eclipse plugin and demonstrated through an illustrative example, ensures the confidentiality and integrity of shared model data by granting stakeholders access only to information relevant to their specific responsibilities and expertise. By filtering out irrelevant data, the approach also mitigates information overload, enabling stakeholders to concentrate on task-relevant aspects of the model, thereby potentially improving collaborative efficiency and effectiveness.

Keywords: Model Driven Engineering, Role Based Access Control, Multi View Modeling, Collaborative Modeling

Email addresses: malvina.latifaj@mdu.se (Malvina Latifaj, federico.ciccozzi@mdu.se (Federico Ciccozzi), antonio.cicchetti@mdu.se (Antonio Cicchetti)

1. Introduction

Model-Driven Software Engineering (MDSE) [1] has redirected the focus of software engineering towards prioritizing models as the fundamental artifacts in the development of complex software systems. MDSE aims to increase the level of abstraction and reduce the accidental complexity associated with the tools and methods used during development [2]. Despite its benefits, the inherent complexity and ongoing development demands of such systems render MDSE an endeavor that cannot always be managed single-handedly. Collaborative MDSE emerges as an integration of collaborative software engineering principles [3] with the abstraction and automation advantages provided by MDSE [4, 5]. This approach fosters the development and maintenance of models collaboratively, enhancing the efficiency and quality of the software development process [6]. While collaborative modeling practices offer considerable advantages, they also introduce notable challenges, particularly in safeguarding the confidentiality and integrity of sensitive information carried by the collaborative models. This type of environment encompasses a wide range of stakeholders, including developers, domain experts, and managers, each bringing their distinct expertise and responsibilities to the table. Their collaboration on a single, base model exposes far more information than necessary to each participant, significantly increasing the risk of confidentiality breaches and compromising the integrity of the collaborative model. This issue is especially alarming considering that models can incorporate proprietary algorithms, business logic, and personal data, making privacy and security paramount. Insights from industrial practices highlight the essential need for trustworthy collaborative modeling environments to feature comprehensive access control mechanisms [7], which are pivotal in safeguarding the confidentiality, integrity, and availability – collectively known as the CIA triad – of information [8]. Access control mechanisms should be tailored to meet practitioners' needs, increasing the likelihood of their adoption in collaborative modeling environments. They should support the definition of fine-grained access permissions and facilitate the management of these permissions to adapt to evolving project needs. In addition, beyond the mere definition and management of access permissions, they must ensure the consistent and accurate enforcement of access permissions through automated processes.

Existing access control approaches hinge on transient, virtual view models that lack independence from their base models [9]. This limitation renders the models

unsuitable for contexts requiring persistent views such as offline collaboration scenarios. Others enforce access permissions via bidirectional transformations, potentially leading to user frustration due to delayed feedback on unauthorized actions [10]. Additionally, most current methodologies focus primarily on basic read and write permissions. Such gaps highlight the need for approaches that support persistent views and provide immediate, granular access control feedback to enhance user experience and efficiency.

In this article, we propose a dual-layered approach that leverages the role-based access control (RBAC) policy [11] to safeguard the confidentiality and integrity of collaborative model information in collaborative modeling environments based on the Eclipse Modeling Framework (EMF) [12]. The first layer limits access to the base model by employing multi-view modeling techniques [13] to create materialized view models, which are essentially subsets of the base model containing only the elements essential for specific user roles. Users can only access and interact with the view models designated to their roles, effectively preventing access to the entire base model. The second layer further refines access down to the individual elements within view models, establishing fine-tuned access permissions. These permissions are enforced by model editors that dictate the extent to which a specific user role can interact with and manipulate each element of the view model. Unlike the trial and error methods, this approach proactively prevents restricted operations.

The remainder of this paper is structured as follows. Section 2 provides background information on the key concepts, while Section 3 describes the related work to this research. Section 4 illustrates a running example used throughout the paper. Section 5 presents the proposed approach, while Section 6 describes the application of the approach on the illustrative running example. Section 7 evaluates the approach on an industrial domain-specific language, while Section 8 provides a discussion on the benefits and limitations of the approach, and threats to validity. Section 9 concludes the paper and describes future research directions.

2. Background

This section describes the key concepts relevant to our study. Section 2.1 outlines the modeling framework for our proposed solution. Sections 2.2 and 2.3 discuss multi-view modeling and access control, respectively.

2.1. Eclipse Modeling Framework and Ecore

Eclipse Modeling Framework (EMF) is a modeling framework and code generation facility for building tools and other applications based on a structured data model [12]. It utilizes XMI-based model specifications to generate a suite of Java classes, complemented by adapter classes that enable viewing and command-based editing of the model, and a basic editor. EMF consists of three fundamental parts. EMF's core framework encompasses Ecore[1], a metamodel for defining models, and provides runtime support including change notifications, default XMI serialization for persistence, and a reflective API for efficient manipulation of EMF objects. The EMF.Edit framework provides generic reusable classes for building editors for EMF models. Lastly, the EMF.Codegen, a code generation facility, is designed to generate all necessary components for a complete EMF model editor. This includes a GUI for setting generation options and initiating generators.

2.2. Multi-View Modeling

Multi-view modeling delivers customized views designed to cater to the unique needs, expertise, and goals of various stakeholders, ensuring alignment and relevance to their specific contributions. The core of multi-view modeling lies in the viewpoint/view/model paradigm, as formalized by the ISO/IEC 42010:2011 standard [14]. According to this standard, a viewpoint represents a specific abstraction using a chosen set of constructs and rules, addressing specific concerns within a system. Consequently, it determines the conventions, like notations, languages, and model types, for crafting a specific kind of view. A view is the resulting instance of applying a viewpoint to a particular system of interest and is composed of one or more models. In the context of multi-view modeling, domain-specific languages (DSLs) are leveraged to address certain system concerns. The models conforming to each DSL then provide a distinct view of the system. Multi-view modeling approaches are classified into *projective* and *synthetic* [15]. Projective methods involve defining viewpoints by selectively abstracting concepts from an existing base language. This approach is notable for facilitating automatic synchronization through centralized manipulations in a single model. However, it requires a well-defined semantics of the base language and may restrict customizability due to the static nature and predefined views of the base language. On

[1] https://download.eclipse.org/modeling/emf/emf/javadoc/2.11/org/eclipse/emf/ecore/package-summary.html

the other hand, synthetic methods establish viewpoints as independent metamodels. In this approach, synchronization is achieved by defining interactions among different viewpoints or views. As the number of views increases, this method becomes increasingly complex, making synchronization progressively more challenging. Prior work [15] has developed a hybrid methodology that combines the projective and synthetic approaches. This method allows for the creation of views based on a base metamodel, similar to the projective approach, yet these views emerge as separate metamodels, similar to the synthetic approach. This ensures inherent synchronization during view definition, alongside the flexibility of introducing views at any development stage.

2.3. Access Control

Access control refers to a security mechanism, pivotal in safeguarding shared resources against unauthorized access, thereby ensuring information security [16]. It acts as a defensive barrier, blocking unauthorized individuals from accessing or altering sensitive data, including proprietary algorithms and strategic business information. This mechanism not only preserves the confidentiality of information but also safeguards its integrity from malicious tampering. Furthermore, access control policies are key in mitigating unintentional alterations by individuals who might not possess the required knowledge or expertise, such as novice engineers. Access control operates fundamentally through two key processes being *authentication* and *authorization* [17]. Authentication entails the verification of a user's credentials, assuring that users are who they claim to be. It necessitates users to provide valid credentials, which are cross-verified against a pre-established database, thereby ensuring that only authorized individuals can access specific information within an organization. On the other hand, authorization determines the extent of access and actions permissible to authenticated users. Authorization operates through access control policies, incorporating rules that dictate the allowable levels of access to various data resources. Two main types of access control policies are recognized [18]:

- Discretionary Access Control (DAC): a user-centric approach that grants users the autonomy to assign access permissions. For instance, platforms like Google Drive [2] allow owners to share files or folders, granting specific access levels to other users, such as edit or view permissions, which can be modified or revoked as necessary.

[2] https://www.google.com/drive/

- Non-Discretionary Access Control (NDAC): here, the determination of access permissions is centralized and administered by system authorities or administrators, rather than the resource owner. This model is particularly apt for scenarios necessitating rigorous security and hierarchical access controls, such as in Amazon Web Services (AWS) Identity and Access Management (IAM) [3], where access is managed centrally using predefined policies and roles.

Of the many access control policies encompassed within NDAC—like multi-level security (MLS), attribute-based access control (ABAC), and Separation of Duty (SoD), we employ Role-Based Access Control (RBAC) [17]. The selection of RBAC stems from a deliberate consideration of organizational structures and the principles of effective access management. RBAC mirrors the hierarchical and role-oriented nature of organizational structures, making it a more intuitive choice for access control in contexts where users are entrusted with permissions based on their designated roles, responsibilities, and hierarchical positions within the organization. Furthermore, RBAC offers a more streamlined and scalable approach to access management, as it simplifies the process of assigning and managing permissions by associating them with roles rather than individual users.

3. Related Work

Previous works [15, 19] propose a hybrid strategy for multi-view modeling that leverages an arbitrary number of views built atop a base modeling language. However, with respect to access control, these works only entail the specification of basic read-only and editable permissions. Martinez et al. [9] introduce a method that leverages views as mechanisms for enforcing access control, utilizing EMFViews [20] to dynamically generate views through live queries on the base model. The elements within these virtual models serve merely as proxies for the actual elements in the base model; hence, view models remain transient and can neither be viewed nor edited independently in other modeling contexts. By having self-persistent views our approach supports offline collaboration, but also benefits from the versatility of a typical metamodel, such as linking any desired concrete syntax or be modified if necessary, including extensions or additional abstraction layers. Other works [10, 21–23] propose a secure collaborative modeling framework utilizing lenses to generate and maintain synchronized secure

[3] https://aws.amazon.com/iam/

views with the underlying base model. Lenses support bidirectional model transformation mechanisms called GET and PUTBACK. The GET function controls read access by filtering the gold (base) model into a front (view) model based on read permissions, while the PUTBACK function validates front model alterations against write permissions before potentially updating the gold model. Hence, the enforcement of access permissions is done by the bidirectional transformations, whereas in our work, this enforcement is managed by the model editor, and is not dependent on the underlying synchronization infrastructure. In addition, this approach to change propagation informs users about their editing permissions through a trial-and-error process, a method the authors themselves acknowledge could lead to user frustration and inefficiency due to delayed feedback on unauthorized modifications. Connected Data Objects (CDO)[4] model repository by Eclipse follows a similar approach in which users are only informed of restrictions on write operations at the time of commit. Our approach adopts a preventive model by integrating a model editor designed to enforce editing permissions. Additionally, while the existing approaches are focused on read and write operations, our solution extends the functionality to CRUD operations, thus providing a more fine-grained control over model interactions. Another relevant study [24] explores the definition of finely-grained role-based access control, but it targets mobile collaborative modeling with active DSLs with a primary focus on mobility aspects. An alternative path of research in access control management proposes implementing security policies directly at the file level. A notable example is Apache Subversion [5], which provides administrators the capability to enforce path-based authorization, thereby controlling user access to specific segments of the repository. Within a collaborative MDSE environment, this approach necessitates the division of models into distinct files. Such fragmentation may obstruct seamless collaboration and is limited to enforcing access control policies with relatively coarse granularity. As can be observed, our discussion in this section is intentionally directed towards strategies that predominantly address access control rather than the broader scope of synchronization in multi-view modeling. This decision stems from our research's use of multi-view modeling primarily as a tool for achieving finely-grained access control efficiently. However, we acknowledge the existence of other approaches for providing synchronization in multi-view modeling, such as view triple graph grammars (VTGGs) [25] [26] and lenses [27].

[4] https://wiki.eclipse.org/CDO/Security_Manager

[5] https://subversion.apache.org

Since the enforcement of permissions is separate from the synchronization infrastructure, and the consistency rules applied during the definition of views and permissions ensure the well-formedness of the view model, any appropriate synchronization transformation approach can be utilized.

4. Running Example

This section introduces a simplified version of a university metamodel, serving as an illustrative example throughout the paper to elucidate the proposed solution and its workflow. Figure 1 illustrates the Ecore-based metamodel of the structure of the *university* consisting of multiple *departments*. Each *department* is tasked with the administration of various academic *programs*. Each *program* is comprised of a series of individual *courses*. Department *employees*, are responsible for delivering course content. Additionally, each department undertakes a variety of research *projects*. *Projects*, represent collaborative research initiatives that may involve a combination of department *employees* and external *partners*, which may include individuals from *industrial* sectors or other *academic* institutions.

4.1. User Roles

In this example, we outline three main user roles engaging with the university model.

- *Project Managers:* represent a group of individuals hired by the university to coordinate the research projects for each department, and oversee the assignment of both internal employees and external partners to various projects.
- *Program Coordinators:* represent a group of individuals hired by the university to coordinate the development and management of academic programs, and guarantee that both programs and courses remain current and relevant.
- *Administration:* their role is related to the coordination of administrative matters related to both research and academic programs. As such, they maintain an overview of the entire model.

User roles and users are defined by an authorized individual, which throughout the rest of this paper we refer to as *admin*. The admin is also in charge of assigning users to user roles.

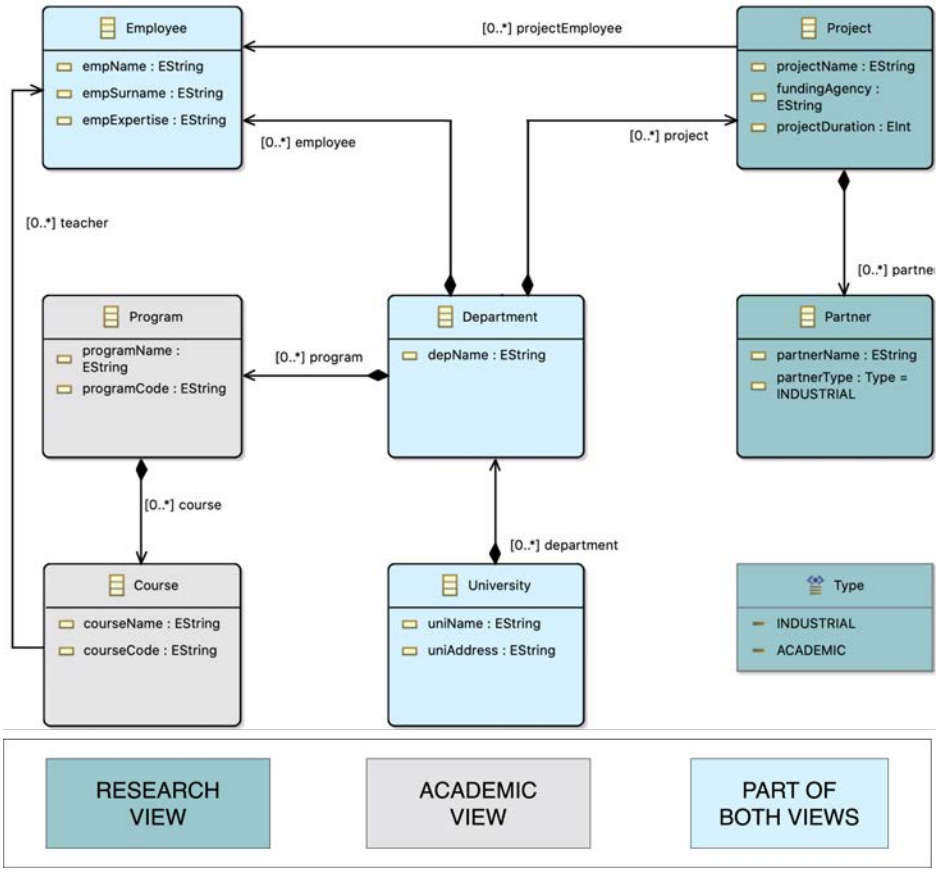


Figure 1: University metamodel

4.2. Views

Each of the outlined user roles has unique responsibilities and areas of expertise, necessitating interaction with only certain parts of the university model that are relevant to their needs. Defining views that contain only a few aspects of the overall base university model allows them to concentrate on the elements that are most crucial to their roles and goals. These views act as a primary access control layer by excluding non-essential elements, thereby minimizing the risk of exposure to potential security vulnerabilities. For this example, the following views are required:

- *Teaching View:* tailored to the needs and concerns of program coordinators, while also accessible to the administration. It includes the following elements: University, Department, Program, Course, and Employee.

- *Research view:* tailored to the needs and concerns of project managers, while also accessible to the administration. It includes the following elements: University, Department, Employee, Project, Partner, and Type.

Figure 1 employs color coding to visually distinguish the elements present in each view. The admin is responsible for defining the properties and meta-elements of a view and allocating access to this view to specific user roles.

5. Proposed Approach

The proposed approach is designed to ensure the confidentiality and integrity of model information in collaborative modeling environments through a dual-layered strategy. Essentially, the first layer filters access by defining view models on top of the base model in a multi-view modeling fashion and allocating roles to these views, thereby allowing interactions solely to users with the designated roles. The second layer refines this access, allowing for precise control over what each role can do within the view model through fine-grained access permissions. The solution is tailored to work with Ecore-based metamodels using EMF tree-based model editors. Before delving into the details and technical aspects of our approach, we clarify some core terminology. A *base metamodel* refers to a metamodel defined in terms of Ecore and representing the foundation for the views. A *base model* is a model that conforms to the base metamodel. A *view metamodel* represents a selection of elements from the base metamodel, also defined in Ecore. A *view model* conforms to a view metamodel.

Figure 2 provides a high-level overview of our proposed solution. For the first layer, an admin (not shown in the figure) establishes a group of *users* and *roles*, and then *assigns* users to these roles. At the same time, an admin can also define a series of *view metamodels*, which may have overlapping elements. This process results also in the generation of *view models*, subsets of the base model, and conforming to the defined *view metamodels*. A *synchronization infrastructure* is generated and maintained between view (meta)models and base (meta)model. The established roles are granted access to these views. For the second layer, the admin sets specific *permissions*, defining the extent of access each role has over a particular view. To enforce these permissions, each role uses a dedicated *model editor* for each view it can access. The total number of model editors required is the sum of those needed by each role. These model editors enforce the specified permissions, allowing users with a given role to *manipulate* the view models within their permitted scope. For instance, Sara, a program coordinator,

uses model editor PC_A to manipulate the academic view model. This editor is tailored to enforce the permissions associated with the program coordinator's role when manipulating the academic view model. Any modifications that Sara makes to the academic view model are then systematically reflected in the base university model. If there are elements that overlap between the academic and research view models and Sara has altered these in the academic view, these changes are automatically propagated from the university model to the research view model.

One important consideration, is the fact that conceptually, the two layers represent a logical separation rather than a strict requirement. While the approach is designed to support scenarios involving multiple views which are commonly used in practice to address separation of concerns and team-specific responsibilities, it does not impose this structure as mandatory. Theoretically, in cases where a single metamodel is sufficient and no view differentiation is needed, create, read, update, and delete permissions could be applied directly to the base metamodel without generating separate view models. The approach accommodates common multi-view scenarios while remaining adaptable to simpler use cases without compromising its applicability or effectiveness.

Section 5.1 describes the design and technical execution of the first layer, highlighting the development of the multi-view modeling environment. Sections 5.2 and 5.3 delve into the design and technical execution of the second layer, focusing on the definition and enforcement of access permissions.

5.1. Setting Up the Multi-View Modeling Environment

The first step of our approach deals with determining the users and their roles. These roles will be granted access to the views, utilizing the specialized wizards outlined in Section 5.1.1. The second step involves the development of the multi-view modeling environment, including the definition and generation of views, as detailed in Sections 5.1.2 and 5.1.3, and the setup of a synchronization framework among these views, as explained in Section 5.1.4.

5.1.1. Role and User Management

The definition of users and roles is carried out by the admin via specialized Java SWT [6] wizards. The *role wizard* streamlines role management, allowing for the creation, alteration, and removal of roles. Each new role shall have a name and a description that outlines its specific functions. The *user wizard*, facilitates

[6] https://www.eclipse.org/swt/

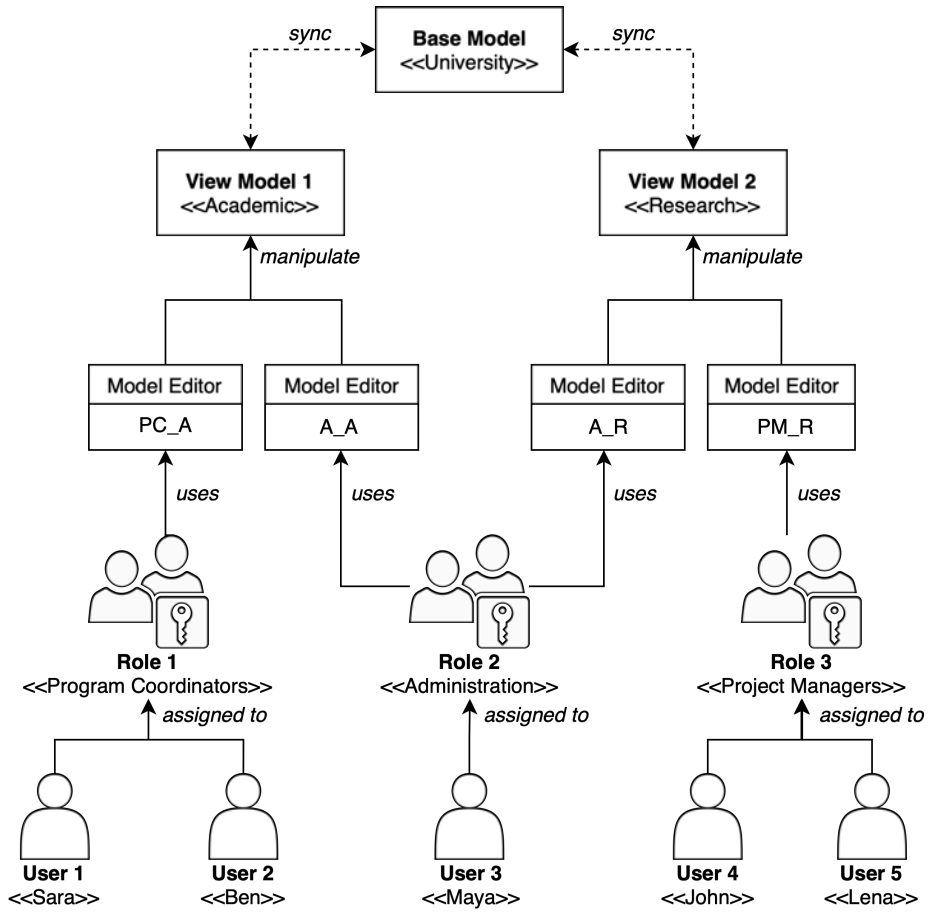


Figure 2: Workflow of the proposed approach

the management of user accounts, allowing for their creation, modification, and deletion. It necessitates details such as the user's first name, last name, username, email, password, and the roles allocated to each user. The solution supports the assignment of multiple roles to a single user, too.

5.1.2. View Metamodel Definition

The view metamodel's definition is administered through a specialized *view wizard*, comprised of several pages, each with a specific purpose. On the *details page*, essential parameters such as viewName, viewNSUri, and viewPrefix are required to identify each new metamodel. To streamline this process, the system automatically generates default values for these details. The admin can then mod-

ify these auto-generated values if customization is desired. In addition, the admin assigns the roles authorized to access the view models and loads the base model. The *selection page* displays the meta elements from the base metamodel. Here, the admin chooses the meta elements to be included in the view metamodel. Such selection follows the set of rules described in related work [19] for the views and the base metamodel to be consistent and their respective models to be synchronizable. In addition, for each EClass element, the admin selects a sub-element that serves as a unique identifier for matching elements between the base and view model.

5.1.3. View Metamodel Generation

View definition is followed by the generation of the view metamodel. The generation process operates by traversing the elements of the base metamodel, organized in a tree structure. For each element, it evaluates whether it has been selected by the admin. Selected elements are included in the view metamodel, while the rest are omitted. Once all elements have been processed, an Ecore Modeling Project is automatically generated. This project is configured with the necessary structure and properties. The generated view metamodel, containing only the user-selected elements, is then saved within this project as an Ecore file.

5.1.4. Synchronization Infrastructure

Multi-view modeling environments have two fundamental aspects: the former involves defining and creating views as described in Section 5.1.2 and Section 5.1.3, respectively; the latter involves developing a synchronization infrastructure that propagates changes between the base model and view models. Considering the broad applicability of our solution across various Ecore metamodels and the potential for users to define numerous views, the synchronization infrastructure in this work is automatically generated. The generation process builds upon previous related work [28], where the authors contributed with a mapping modeling language for specifying relationships between elements within two Ecore-based metamodels, along with higher-order transformations (HOTs) that facilitate the generation of model-to-model (M2M) transformations based on these mapping models. Figure 3 depicts the generation of the synchronization infrastructure, combining the aforementioned solution [28] with necessary customizations for its adaptation to the current scenario, and is referenced repeatedly in this section for a detailed exploration of the synchronization infrastructure. Our current solution sets itself apart in two main aspects.

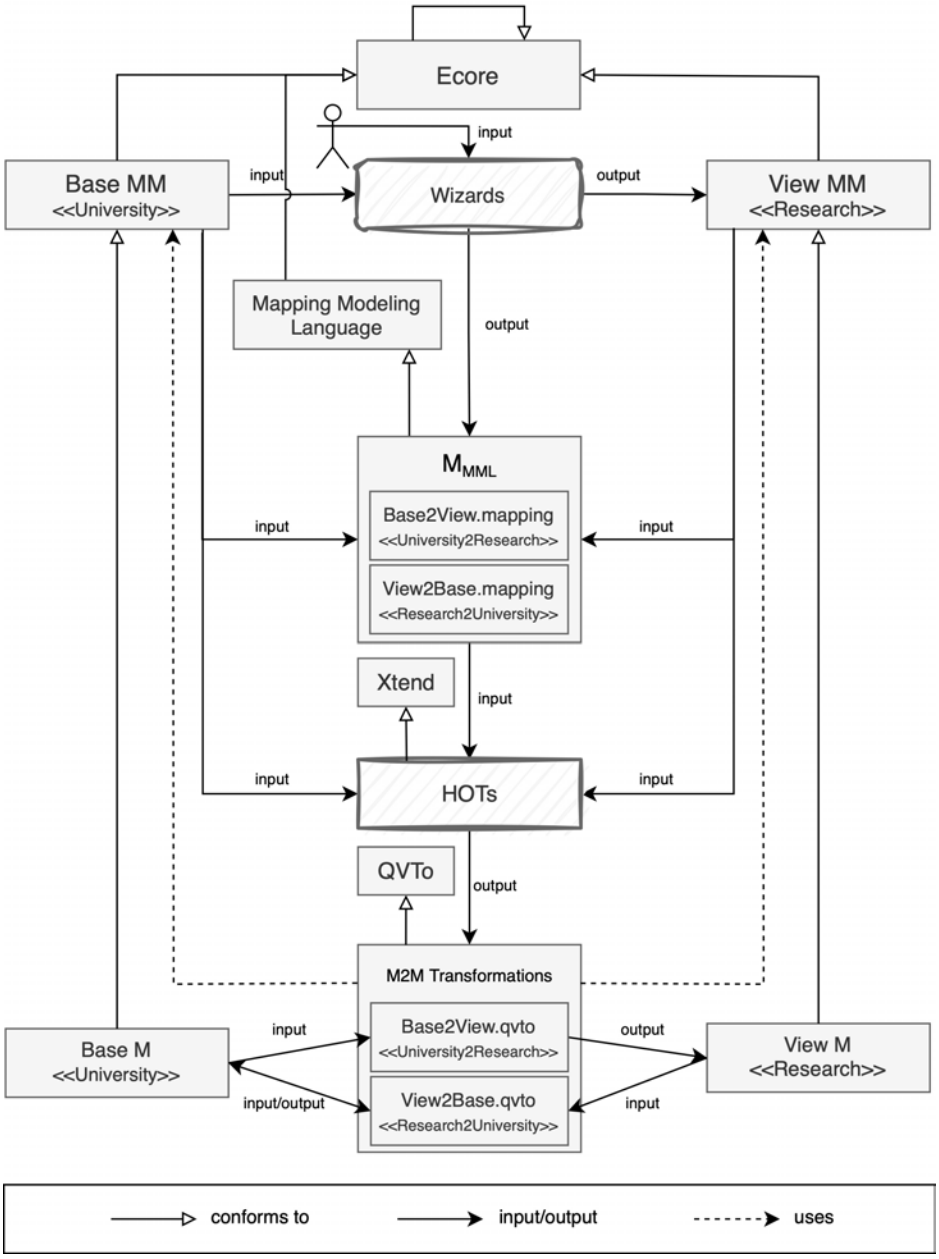


Figure 3: Setup of the synchronization infrastructure between base and view model

- We have streamlined the process by automating the generation of mapping models. This is a significant improvement over the related work, which re-

quires manual user-definition of these mapping models. This enhancement is a direct outcome of the narrower and more defined context in this work, where the view metamodel forms a precise subset of the base metamodel, and every meta element in the view metamodel is identical to its counterpart in the base metamodel, having been derived directly from the latter.

- The approach proposed in the related work was designed for scenarios where there is a complete mapping between the source and target metamodels, and the developed HOTs are intended for such scenarios. In our current approach, however, the view metamodel is just a part of the base metamodel, hence, not every element of the base has a corresponding element in the view. As a consequence, employing the previously defined HOTs could lead to the generation of model transformations, erroneously modifying elements of the base model not pertaining to the view. To overcome this challenge, we have designed and implemented new HOTs specifically aimed at ensuring these transformations effectively retain all the unique elements and information of the base model.

The following paragraphs provide details on the generation of mapping models and the HOTs employed for the generation of the synchronization infrastructure (i.e., model transformations).

Generation of Mapping Models.. The selection of elements for a particular view triggers not only the construction of an Ecore metamodel representing the view, but also the generation of two mapping models (i.e., `Base2View.mapping` and `View2Base.mapping`), which contain the relationships between elements of the base and view metamodels. These mapping models conform to the mapping modeling language, introduced in the related work [28], and encapsulate the links between the corresponding elements of the two metamodels. Given that the view metamodel represents an exact subset of the base metamodel, the process inherently ensures that each element in the view metamodel has an unambiguous correspondence with a counterpart in the base metamodel and vice versa.

Higher-Order Transformations.. HOTs are a type of model transformation where the input and/or output are transformation models themselves [29]. They are employed to leverage the capabilities of transformations, treating them as objects. Prior research [28] has proposed HOTs that, driven by user-defined mapping models, generate M2M transformations conforming to the Query/View/Transfor-

mation Operational (QVTo)[7] language. These transformations rebuild the target model based on information from the source model. In the given context, employing these HOTs has proven effective for initially generating and subsequently updating the view model (i.e., target) from the base model (i.e., source). This is because the generated model transformation (i.e., `Base2View.qvto`) can account for every element in the view model by referencing its counterpart in the base model. However, challenges arise when attempting to propagate changes from the view model back to the base model. The generated M2M regenerates the base model (i.e., target) to match the view model, but cannot account for all its elements since not all of them have a counterpart in the view model (i.e., source). As a result, information associated with these unmatched elements is lost during the transformation process. In response to this challenge, we developed HOTs that use the generated mapping model defining correspondences from the view model to the base model (i.e., `View2Base.mapping`) as input for generating a model transformation (i.e., `View2Base.qvto`) for propagating changes from the view model to the base model. This model transformation operates directly on the base model, updating elements that have a correspondence in the view model, all while preserving those elements that are unique to the base model. This can also be seen in Figure 3. In the case of `Base2View.qvto`, it takes a base model as input and produces a view model as output. On the other hand, `View2Base.qvto` takes both the view model and base model as input and directly modifies the base model to reflect changes made in the view model. In the following, we describe the outcome of our HOT, which is the generated `View2Base.qvto` model transformation.

Input/Output Specifications: the model transformation uses the base and view metamodels as input and base metamodel as output. During the execution of this transformation, it accepts a base and view model as input and produces an updated base model that accurately reflects the applied changes.

Element Matching: the model transformation requires matching elements between the base and view model by comparing the elements' unique identifiers defined at view definition phase. Using identifiers allows to correctly find the two corresponding elements between base and view models.

Handling of Containment EReferences: can be updated, added, or deleted (i.e.,

[7] `https://wiki.eclipse.org/QVTo`

their target EClass can be updated, added, or deleted). The model transformations use the information from the element matching process for differentiating between updates, additions, and deletions of elements.

- Update: an element in the view model with a match in the base model, implies that the element has neither been added or deleted, but may have been updated. Hence, an update rule is invoked to synchronize the possible changes.
- Addition: an element in the view model with no match in the base model implies the addition of the element in the view model. Thus, this element is also added to the base model.
- Deletion: an element in the base model without a match in the view model implies the deletion of the element from the view model. Thus, this element is also deleted from the base model.

Handling of Non-Containment EReferences: can be added or removed (i.e., their target EClass can be added or removed from the list of referenced EClasses). The model transformations use the information from the element matching process to differentiate between the two.

- Addition: a non-containment EReference in the view model, pointing to a target element with no match among the target elements of the same non-containment EReference in the base model, implies an addition. Thus, this element is also added to the list of target elements of the non-containment EReference in the base model.
- Removal: a non-containment EReference in the base model, pointing to a target element with no match among the target elements of the same non-containment EReference in the view model, implies a removal. Thus, this element is also removed from the list of target elements of the non-containment EReference in the base model.

Handling of EAttributes and EEnumLiterals: For EAttributes and EEnumLiterals, which by design cannot be added or deleted, a standard update transformation is applied. This step ensures that they are consistently updated between the view and base model.

We leveraged Xtend [8] – a high-level, Java-based programming language noted for its efficacy in code generation tasks – to write the HOTs used generate the model transformation from a mapping model. The generated model transformation propagates changes from the view to the base model while simultaneously ensuring that elements in the base model, which are not impacted by the transformation, remain intact. The completion of the synchronization infrastructure, achieved through the generation of two unidirectional model transformations, marks the establishment of the initial access layer. This layer permits users to access and alter only designated areas of the base model, referred to as view models, according to their allocated roles, and supports bidirectional synchronization.

5.2. Access Permissions Definition

The generation of the multi-view modeling environment is followed by the definition of access permissions for each user role to interact with a given view. Definition of access permissions is carried out by the admin using CRUD (Create, Read, Update, Delete) operations [30]. The admin selects the allowed operations for each role and in relation to each meta element in the view metamodel, as described in Section 5.2.1. The selection follows the set of predefined rules outlined in Section 5.2.2, ensuring the consistency of access permissions.

5.2.1. Wizard for Access Permissions Definition

The definition of access permissions for a set of roles on a given view is carried out using the *view wizard* introduced in Section 5.1.2. The *permissions* page is populated with the roles with access to the view and the view's meta elements. Each meta element is associated with four checkboxes, representing CRUD operations, as shown in Figure 4. Depending on the specific kind of EObject – be it an EClass, a containment or non-containment EReference, EAttribute, EENum, or EENumLiteral – only certain checkboxes are active and selectable. Other checkboxes are not active since the operations that they correspond to do not apply to the EObjects they are associated with. An overview of checkbox status for each EObject type is provided in Table 1.

To maintain a consistent layout that helps users navigate the wizard interface more intuitively, we have retained the non active checkboxes in the wizard. Although not directly selectable, they serve to provide a clear and coherent structure. Additionally, checkboxes marked with an asterisk (*) are designed to be automatically activated in response to the selection of certain related checkboxes, even

[8] https://eclipse.dev/Xtext/xtend/documentation/index.html

	C	R	U	D
EClass	A	A	NA*	A
ERef (containment)	A	A	NA	A
ERef (non-containment)	NA	A	A	NA*
EAttribute	NA	A	A	NA*
EENum	NA	A	NA	NA
EENumLiteral	NA	A	NA	NA

Table 1: Checkbox status per type of EObject (A - Active; NA - Non Active)

though they remain non active for direct user interaction. For instance, when a user selects the Create (C), Update (U), or Delete (D) checkboxes for any EAttribute or EReference within an EClass, the solution triggers the selection of the Update (U) checkbox for that EClass. Similarly, selecting the Delete (D) checkbox for an EClass triggers the automatic selection of the Delete (D) checkbox for all contained EReferences and EAttributes.

5.2.2. Consistency Rules

The mechanism for automatically managing checkbox states goes beyond just handling inactive ones. We established a comprehensive set of rules to ensure uniform behavior across all checkboxes. For instance, if an EClass is removed, it naturally entails the removal of its associated EAttributes, EReferences, and any EClasses linked through containment EReferences. This reasoning is embedded in the wizard to avoid potential errors in the enforcement of permissions. The wizard updates checkbox statuses (either selecting or deselecting them) on-the-fly. This real-time mechanism gives users a clear understanding of how their choices affect the permissions of related meta-elements. To define the consistency rules, we started with overarching principles. These principles served as the foundation for defining the rules. The latter are presented in Table 2 and their interpretation is facilitated by the legend provided in the same table. Each rule is directly linked to the underlying principles, which are outlined below.

P1: Permission to perform a create (C), update (U), or delete (D) operation on an object is conditional to having the read (R) permission on the object – see R1, ~~R5~~ in Table 2.

P2: Permission to perform any CRUD operation on a nested object is conditional to having the read (R) permission on the container object. In addition, any CUD operation on a nested object is conditional to having the update (U) permission on the container object – see R2, R3, ~~R6~~, ~~R7~~.

P3: Permission to perform a delete (D) operation on a container object is conditional to having the delete (D) permission on all nested objects – see R4.

P4: Permission to perform any CRUD operation on an EClass is conditional to having those permissions on its incoming containment EReference and the container of that EReference (the latter is based on P2) – see R8 to R11, ~~R8~~ to~~R11~~.

P5: Permission to perform any CRUD operation on an EReference is conditional to having those permissions on its source and target EClass – see R12 to R17, ~~R12~~ to ~~R15~~.

P6: Permission to perform a read (R) or update (U) operation on an EAttribute is conditional to having the read (R) permission on the container EClass – see R18 to R21.

P7: Permission to perform a read (R) operation on an EEnumLiteral is conditional to having the read (R) permission on the container EEnum – see R22, ~~R23~~.

In alignment with the established principles, we formulated the set of consistency rules delineated in Table 2. The first column lists the identifiers of the rules, the second outlines the user actions in the wizard interface, the third describes the effects of these actions on other checkboxes, and the fourth provides concrete examples for each rule, based on the wizard shown in Figure 4. The user actions and the examples have been structured to ensure a precise understanding of the effects. For clarity, we tried out all examples involving the selection of a specific checkbox with the wizard in a baseline state, with all checkboxes initially deselected. Similarly, for the deselection of checkboxes, we consistently used the same checkbox that was previously selected. This methodological consistency allowed us to isolate the effects of each action without interference. It is important to note that the user's interaction with a checkbox (either through selection or deselection) can initiate a ripple effect, where each affected checkbox might further alter the state of others. This chain reaction continues until all affected checkboxes are appropriately adjusted. To illustrate, assume all checkboxes in Figure 4 are initially deselected. If a user selects checkbox R(9) – which corresponds to the read (R) operation for `EAttribute:depName` at index 9 – this action triggers the automatic selection of the read (R) operation checkbox for `EClass:Department`, by principle P2. Following this, selecting R(7) would lead to the selection of

R(22), as dictated by principle P4. Subsequently, selecting R(22) would result in the selection of R(21), again following principle P2. This example represents a three-level chain reaction within the checkbox interactions (i.e., R(9) $\xrightarrow{L1}$ R(7) $\xrightarrow{L2}$ R(22) $\xrightarrow{L3}$ R(21)). For the sake of readability and conciseness, in Table 2 we limited the illustration of chain reactions to just the second level. This applies to both the examples shown and their effects, to avoid the complexity of longer chains that could span multiple levels.

5.3. Access Permissions Enforcement

The enforcement of the defined access permissions is achieved through EMF tree-based model editors. These editors provide a graphical user interface that allows users to visualize and modify models through a hierarchical tree structure. While they come with essential functionalities, their design allows for customization to meet specific needs. We leverage this flexibility to enforce the defined access permissions. Section 5.3.1 details EMF's standard approach to creating these tree-based model editors. Section 5.3.2 details the customization methods we have employed for generating model editors that enforce the defined access permissions.

5.3.1. Generation of EMF Tree-Based Model Editors

EMF tree-based model editors are generated through an automated process supported by the framework itself. Starting from an Ecore model, the process first involves the creation of a generator model (GenModel). The GenModel is a configuration model that dictates the generation of the following plugins.

- *Model Plugin:* hosts the Java classes that represent the Ecore model. These classes are derived from the Ecore model, conforming to the GenModel's specifications.
- *Edit Plugin:* provides infrastructure for structured interaction with the model. The key components of this plugin are the Item Providers. Item Providers are Java classes that support viewing and editing objects within the EMF tree-based editor. They specify how model elements are presented and manipulated in the editor.
- *Editor Plugin:* comprises the graphical user interface for the EMF editor. This includes the tree-based interface for model interaction and additional UI components like wizards that enhance user engagement with the model.

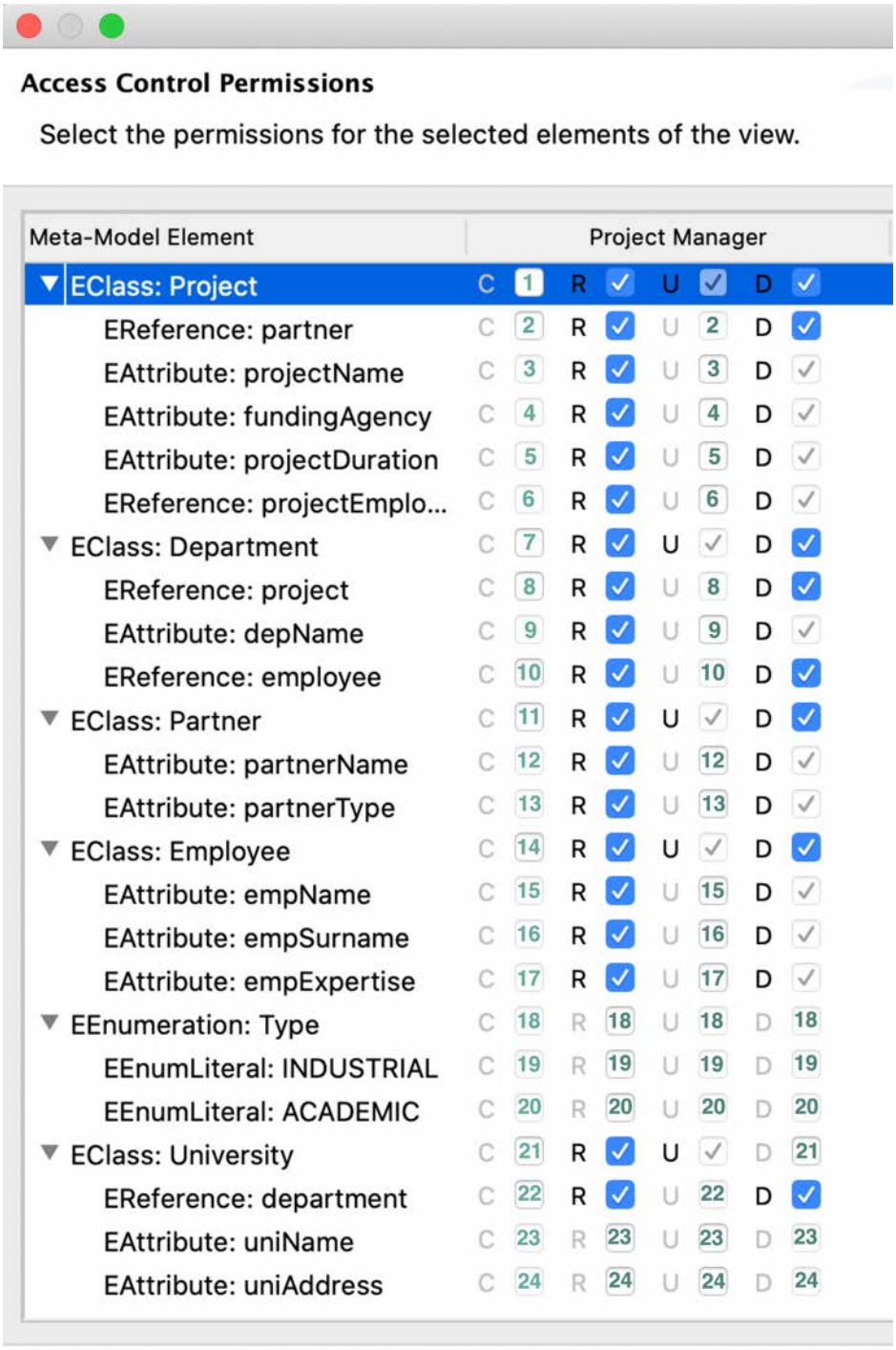


Figure 4: Permissions triggered by selection of D(22)

Rule	Action	Effect	Example
General rules			
R1	$P_{C\|U\|D}(O)$	$P_R(O)$	$C(21) \Rightarrow R(21)$
R2	$P_R(N)$	$P_R(C)$	$R(23) \Rightarrow R(21)$
R3	$P_{C\|U\|D}(N)$	$R1 \wedge P_{RU}(C)$	$U(23) \Rightarrow R(23) \wedge RU(21)$
R4	$P_D(C)$	$P_D\forall(N) \rightarrow R3\forall(N)$	$D(7) \Rightarrow D(8\text{–}10) \rightarrow [R(7\text{–}10) \wedge U(7)]$
~~R5~~	~~$P_R(O)$~~	~~$P_{CUD}(O)$~~	~~$R(21) \Rightarrow CUD(21)$~~
~~R6~~	~~$P_R(C)$~~	$R5 \wedge$ ~~$P_{CRUD}\forall(N)$~~	~~$R(21) \Rightarrow CUD(21) \wedge CRUD(22\text{–}24)$~~
~~R7~~	~~$P_{CUD}\forall(N)$~~	~~$P_U(C)$~~	~~$CUD(22\text{–}24) \Rightarrow U(21)$~~
EClass (Class) specific consistency rules (not the root)			
R8	$P_{C\|R\|D}(Class)$	$P_{C\|R\|D}(CR)$	$R(7) \Rightarrow R(22)$
~~R8~~	~~$P_{C\|R\|D}(Class)$~~	~~$P_{C\|R\|D}(CR)$~~	~~$R(7) \Rightarrow R(22)$~~
R9	$P_C(Class)$	$R1 \wedge [P_C(inCR) \rightarrow R3(inCR)]$	$C(7) \Rightarrow R(7) \wedge [CR(22) \rightarrow RU(21)]$
~~R9~~	~~$P_C(Class)$~~	~~$P_C(inCR) \rightarrow R7(inCR)$~~	~~$C(7) \Rightarrow C(22) \rightarrow U(21)$~~
R10	$P_R(Class)$	$P_R(inCR) \rightarrow R2(inCR)$	$R(7) \Rightarrow R(22) \rightarrow R(21)$
~~R10~~	~~$P_R(Class)$~~	~~$P_R(inCR)$~~	~~$R(7) \Rightarrow R(22)$~~
R11	$P_D(Class)$	$[R4 \rightarrow R13(CR)] \wedge [P_D(inCR) \rightarrow R15(inCR)]$	$D(7) \Rightarrow$ (see Figure 4)
~~R11~~	~~$P_D(Class)$~~	~~$R8 \rightarrow R7$~~	~~$D(7) \Rightarrow D(22) \wedge U(21)$~~
Containment EReference (CR) specific consistency rules			
R12	$P_{C\|R\|D}(CR)$	$P_{C\|R\|D}(tClass)$	$R(22) \Rightarrow R(7)$
~~R12~~	~~$P_{C\|R\|D}(CR)$~~	~~$P_{C\|R\|D}(tClass)$~~	~~$C(22) \Rightarrow C(7)$~~
R13	$P_C(CR)$	$R3 \wedge [R12 \rightarrow R1(tClass)]$	$C(22) \Rightarrow R(22) \wedge RU(21) \wedge [C(7) \rightarrow R(7)]$
~~R13~~	~~$P_C(CR)$~~	~~$R7 \wedge R12$~~	~~$C(22) \Rightarrow U(21) \wedge C(7)$~~
R14	$R(CR)$	$R2 \wedge R12$	$R(22) \Rightarrow R(21) \wedge R(7)$
~~R14~~	~~$P_R(CR)$~~	~~$R12$~~	~~$R(22) \Rightarrow R(7)$~~
R15	$P_D(CR)$	$R3 \wedge [R12 \rightarrow R11(tClass)]$	$D(22) \Rightarrow$ (see Figure 4)
~~R15~~	~~$P_D(CR)$~~	~~$R7 \wedge R12$~~	~~$D(22) \Rightarrow U(21) \wedge D(7)$~~
Non-Containment EReference (NCR) specific consistency rules			
R16	$P_R(NCRef)$	$[R2 \rightarrow R10(sClass)] \wedge [P_R(tClass) \rightarrow R10(tClass)]$	$R(6) \Rightarrow [R(1) \rightarrow [R(8) \rightarrow R(7)]] \wedge [R(14) \rightarrow [R(10) \rightarrow R(7)]]$
R17	$P_U(NCRef)$	$R3 \rightarrow R16$	$U(6) \Rightarrow [R(6) \rightarrow [R(1) \rightarrow [R(8) \rightarrow R(7)]]] \wedge [R(14) \rightarrow [R(10) \rightarrow R(7)]]$
EAttribute (Att) specific consistency rules (R20 and R21: Attribute EType = ENum)			
R18	$P_R(Att)$	$R2 \rightarrow R10(C)$	$R(12) \Rightarrow R(11) \rightarrow R(2\text{-}1)$
R19	$P_U(Att)$	$R3 \rightarrow R10(C)$	$U(12) \Rightarrow [R(12) \wedge RU(11)] \rightarrow R(2\text{-}1)$
R20	$P_R(Att)$	$R18 \wedge [P_R(EN) \rightarrow R22(EN)]$	$R(13) \Rightarrow [R(11) \rightarrow R(2\text{-}1)] \wedge [R(18) \rightarrow R(19\text{-}20)]$
R21	$P_U(Att)$	$R19 \wedge [P_R(EN) \rightarrow R22(EN)]$	$U(13) \Rightarrow [R(13) \wedge [R(11) \rightarrow R(2\text{-}1)]] \wedge [R(18) \rightarrow R(19\text{-}20)]$
ENum (EN) and ENumLiteral (ENL) specific consistency rules			
R22	$P_R(ENL)$	$P_R(EN)$	$R(19) \Rightarrow R(18)$
~~R23~~	~~$P_R(EN)$~~	~~$P_R\forall(ENL)$~~	~~$R(18) \Rightarrow R(19\text{-}20)$~~

Legend	
Description of acronyms used is as follows: object (O), container object (C), nested object (N), containment reference (CR), non-containment reference (NCR), incoming containment reference (inCR), source class (sClass), target class (tClass), attribute (Att), enumeration (EN), enumeration literal (ENL).	
$P_X(O)$ and ~~$P_X(O)$~~	Selection and deselection of a permission $X \in \{C,R,U,D\}$ on an object O
$P_X\forall(O)$ and ~~$P_X\forall(O)$~~	Selection/deselection of a permission on all objects O
$P_{C\|U\|D}(O)$ and ~~$P_{C\|U\|D}(O)$~~	Selection/deselection of at least one of $\{C,U,D\}$ on O
$P_{CUD}(O)$ and ~~$P_{CUD}(O)$~~	Selection/deselection of all of $\{C,U,D\}$ on O

Table 2: Set of rules for consistency enforcement between access permissions and legend for reading the table

The generation process of these plugins involves the use of Java Emitter Templates (JET)[9]. EMF comes with a suite of JET templates that are written in a (Java Server Pages) JSP-like syntax and express EMF code patterns. These templates

[9] https://projects.eclipse.org/projects/modeling.m2t.jet

undergo processing by the JET engine, which converts each template into the source of a Java class. Subsequently, the JET engine compiles, dynamically loads, and utilizes these classes to produce the specified output. By default, EMF uses *static templates* which are converted into template classes and compiled ahead of time to speed up the generation process [12].

5.3.2. Customization of EMF Tree-Based Model Editors

To enforce the user-defined access permissions in EMF tree-based model editors, we focus on customizing Item Providers, which manage the viewing and editing of elements. While one could customize each generated Item Provider separately, this approach is time-consuming and inefficient. Our methodology, therefore, adopts a streamlined and automated process where we initially customize the JET template utilized for generating Item Providers to consider the defined access permissions. Then, we guide the generator to use the customized JET template, which is also referred to as *dynamic* because, unlike static templates, it has not been pre-compiled. This process is described in Figure 5. When the admin finalizes the *view wizard*, it initiates the generation of *view.ecore*, *view.genmodel*, and *view.rbac*. The *view.rbac* file represents a model conforming to a custom-designed language (detailed in Section 5.3.3) which encapsulates the defined access permissions. The *view.ecore* file encapsulates the specifics of the established classes, while the *view.genmodel* file contains essential information required for code generation. When generating the *view.genmodel*, we specified particular properties to instruct the generator to utilize the customized JET template. Specifically, we enabled the `dynamic templates` property by setting its value to `true`, guiding the system to skip the use of default templates from `org.eclipse.emf.codegen.ecore.genmodel` and instead choose to translate and compile the JET templates we provided. Moreover, we configured the `template directory` to point to the location of the dynamic templates in our project. The combination of these sources of information enables the EMF code generator to generate a tree-based model editor enforcing the defined access permissions.

The advantages of this approach are multiple. Firstly, we automate the customization process of Item Providers, avoiding the need for individual manual modifications. Secondly, by embedding the access permissions directly into the generation process of the Item Providers, we ensure a consistent implementation of these permissions across the model editor. Finally, we offer flexibility and adaptability, allowing for straightforward updates to access permissions without extensive manual reworking of the underlying Item Provider code. Section 5.3.3 describes the role-based access control domain-specific modeling lan-

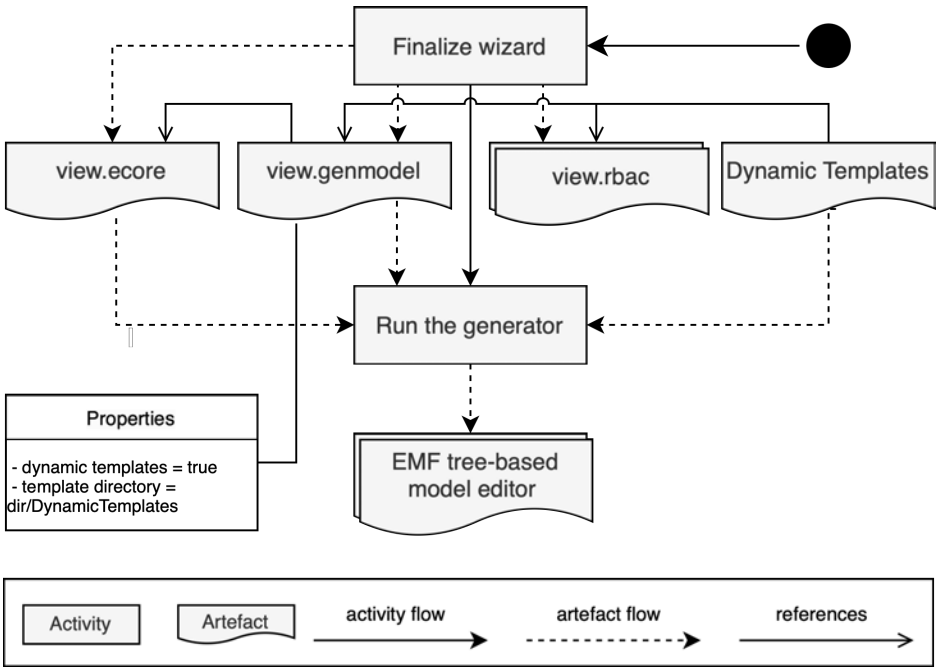


Figure 5: Generation workflow of tree-based model editors with RBAC

guage (DSML). Section 5.3.4 details the customizations made to the JET template responsible for the generation of Item Providers.

5.3.3. Role-Based Access Control DSML

To encapsulate the user-defined access permissions, we defined a role-based access control DSML. For each role that has access to a particular view, a unique RBAC model is generated. These RBAC models encapsulate the access permissions each role has over each element within the view. We intentionally structured the DSML to imitate the structure of an Ecore metamodel for seamless integration as input to the JET template and possible reverse engineering needed for future work. As depicted in Figure 6, the root element is the *AccessControlModel* characterized by attributes *name* – reflecting the name of the view metamodel – and *role*, denoting the user role to which the permissions apply. The elements *EClass* and *EEnum* are compositionally linked to the root element. In a similar compositional manner, *EAttribute* and *EReference* are linked to *EClass*, while *EEnumLiteral* to *EEnum*. Each of these elements possesses a set of features that they inherit from the *ElementPermission* abstract class. These features include a *name* attribute for the EObject they are associated with, and a reference to the EObject *element*.

Moreover, they feature a list of *permissions* that define the CRUD operations, as detailed in the *Permissions* enumeration.

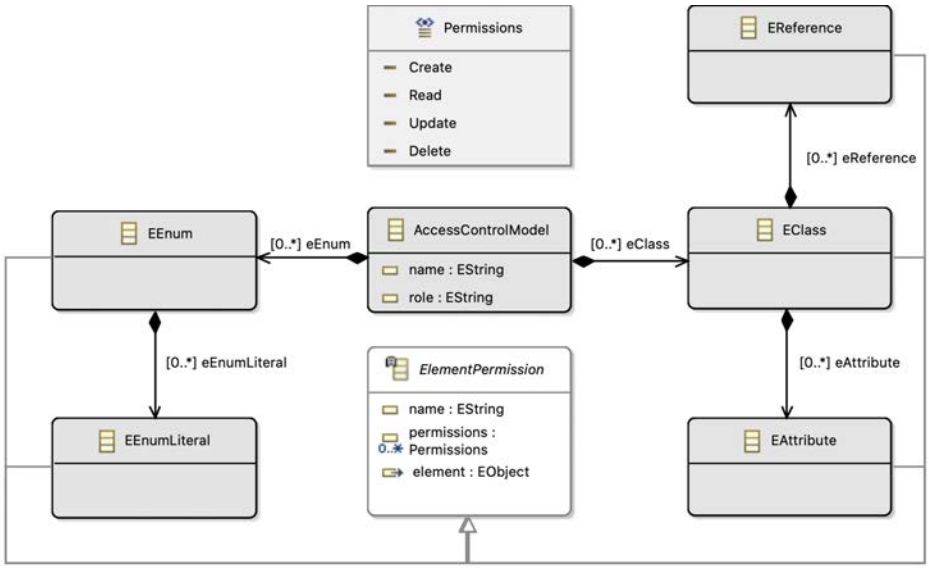


Figure 6: Role-based access control DSML

5.3.4. Dynamic Templates

EMF utilizes a collection of JET templates to automatically generate tree-based model editors. To refine this generation process to enforce access permissions within the model editor, we adapted the JET template responsible for Item Provider generation. This adaptation involves using a generated RBAC model as input, directing the generation of Item Providers in a way that enforces the specified permissions. Initially, we duplicated the JET templates into our project and we modified the `ItemProvider.javajet` template accordingly. Since the tree-based editor's default setup allows for unrestricted CRUD operations on all model elements, we intervened to implement constraints where the out-of-the-box functionality permits more than what the RBAC model dictates. Specifically, our custom logic in the template evaluates the RBAC model's permissions for each model element and adjusts the generation of Item Providers to enforce these restrictions accordingly. For each element, depending on which operations are restricted, the template selectively generates or omits code segments in the Item Providers. For instance, if the RBAC model restricts the creation of certain EClasses for specific roles, the template is designed to omit the generation of code segments in Item

Providers that would otherwise enable such creation capabilities. This principle is similarly applied for all the remaining operations.

The mechanisms described so far represent the realization of the second security layer which permits users to manipulate the elements of the accessible view models according to the access permissions defined for each element. The interested reader can find further details on the required adjustments to the Item Providers for restricting each operation type in our GitHub repository [10].

6. Illustrative Example

To provide a complete overview of the running example introduced in Section 4, this section details the entire process leading up to the generation of the customized model editor. A demo illustrating these steps is available in our GitHub repository [10]. Figure 7 illustrates the *permissions'* page of the view wizard. In this example, we define the *project manager's* access permissions on the *research view*. The first column of the wizard displays the meta model elements that comprise the *research view*, selected in prior step. The second column features four checkboxes for each view element, representing the complete set of CRUD operations. The final column includes a checkbox for each EAttribute object, enabling the user to select a unique identifier for each EClass. The wizard ensures real-time consistency in access permissions through a dynamic check activated each time a checkbox is selected. This process may prompt the selection of additional checkboxes to maintain consistency. For instance, selection of delete (D) operation on `EClass:Project` leads to the automatic selection of read (R) operation on `EClass:Project` , selection of delete (D) permission on all its children, and selection of delete (D) operation on `EReference:project`. This cascade of automatic selections continues, ensuring all relevant checkboxes are selected to preserve the consistency of access permissions. Finalization of the wizard results in the creation of several artefacts, including the view metamodel and the corresponding view model, where the latter is a subset of the base model. The base model illustrated in Figure 8a is an instantiation of the university (base) metamodel illustrated in Figure 1, while the view model illustrated in Figure 8b, is an instantiation of the research (view) metamodel comprised of the view elements illustrated in Figure 7. Model transformations, generated upon finalizing the wizard, ensure that modifications in the base and view models are accurately propagated among one another,

[10] `https://github.com/MLJworkspace/RBAC_Solution`

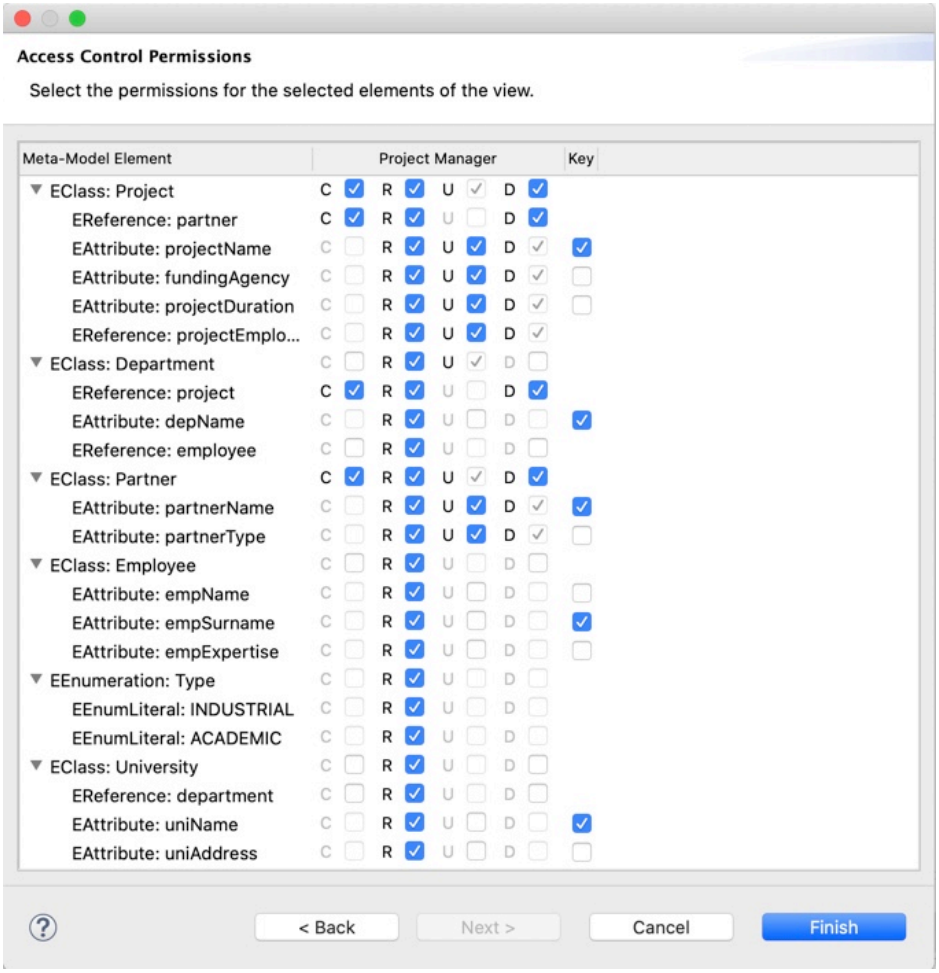


Figure 7: Defined access permissions for illustrative example

and guarantee that information is preserved without loss during the propagation of changes. For example, when the project is renamed from Saturn to Jupiter in the view model, this change is propagated in the base model. Simultaneously, the program and course elements, which exist in the base model but not in the view model (nor are they part of the view metamodel), remain intact.

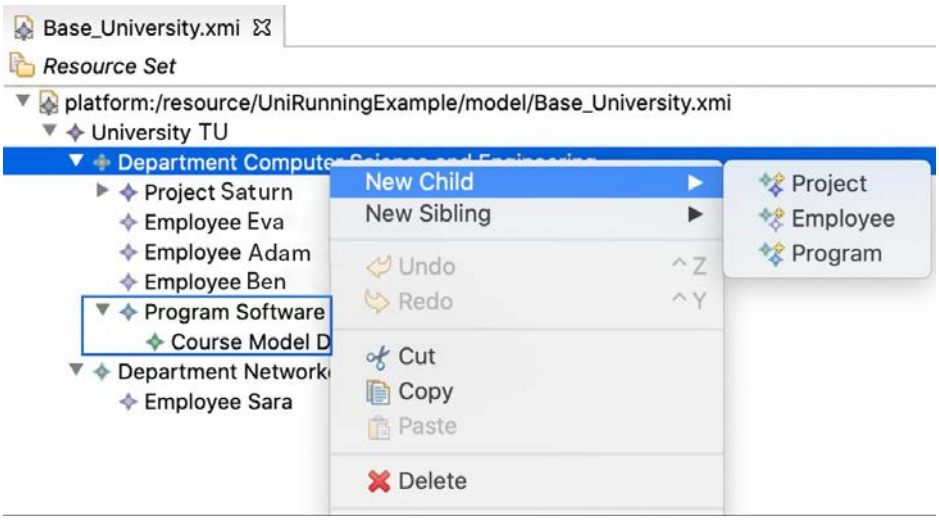
In the context of enforcing access permissions, the models illustrated in Figure 8 are accessed through their corresponding customized EMF tree-based model editors. The comparison between the two reveals distinct permission settings. In the base model, it is possible to delete a *department* instance and create new *projects*, *employees*, and *programs*. Conversely, the view model restricts these capabilities. Here, deleting a *department* instance is disabled, and the creation of *employees* and *programs* is prohibited. These limitations align with the access permission established in Figure 7, which explicitly restricts project managers from performing these operations. Access permission enforcement is achieved by customizing the generation process of the Item Providers – responsible for viewing and editing elements in the model editor – which in the customized version, considers the permissions defined in the wizard in Figure 7 and generates the Item Providers accordingly. Listing 1 presents a code snippet that illustrates the `createRemoveCommand()` method as implemented in the University ItemProvider. This method is not generated by the standard generation process. Instead, it is specifically generated due to the project manager's lack of delete (D) permission for the `EClass:Department`. As a result, this method disables the Delete option when right-clicking on a Department element, a functionality highlighted in Figure 8b.

```
1  public Command createRemoveCommand (EditingDomain editingDomain, EObject owner, EStructuralFeature feature,
       Collection<?> collection) {
2      if (owner instanceof University) {
3          for (Object object : collection) {
4              if (object instanceof Department) {
5                  return UnexecutableCommand.INSTANCE;
6              }
7          }
8      }
9      return RemoveCommand. create (editingDomain, owner, feature, collection);
10 }
```

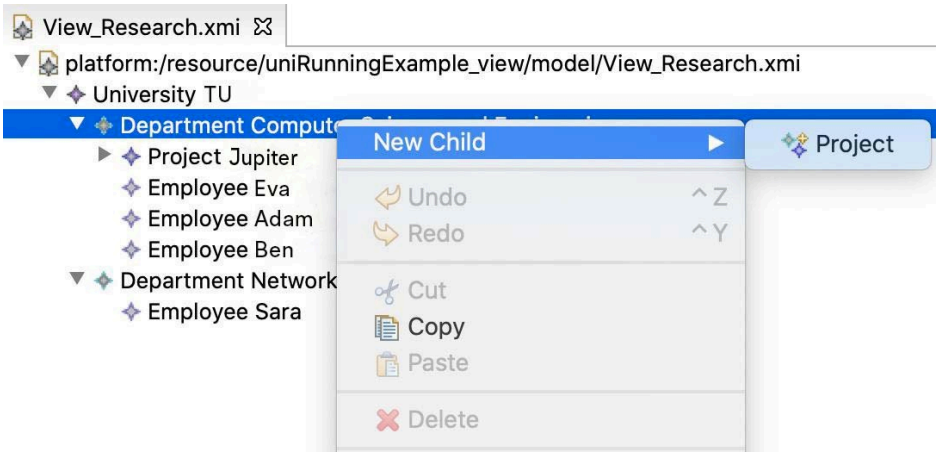
Listing 1: createRemoveCommand() method for Department in University ItemProvider

7. Evaluation

We evaluate our approach and prototype on a commercial textual language designed for developing stateful and event-driven real-time applications. This



(a) Editor providing full access to the base model



(b) Editor providing limited access to the view model

Figure 8: Tree-based editors for base and view models

language is part of the Code RealTime [11] extension, which operates for both Visual Studio Code and Eclipse Theia. The language, known as Art, introduces a new syntax for well-established concepts rooted in the Real-Time Object-Oriented Modeling (ROOM) language [31], which has been successfully used in industry for over three decades. Art is particularly well-suited for modeling both the behavior and structure of real-time applications. The structural part defines the high-level architecture through elements such as capsules, ports, connectors, protocols, and capsule parts. The behavioral part captures the dynamic behavior of the system using state machines, transitions, states, pseudostates, actions, and other relevant elements. Although Art is a commercial language, comprehensive descriptions of its features and usage are publicly available [12]. Our evaluation process emphasizes the behavioral part, demonstrating how well-defined access controls can secure sensitive behavioral information. One of the key reasons for selecting Art for this evaluation is our involvement in its initial development, particularly in defining the language's behavioral part [32]. Moreover, Art's behavioral modeling capabilities are sufficiently complex to demonstrate the effectiveness of fine-grained access control. While the Art language itself cannot be made publicly available due to its commercial nature, we can provide an overview of its complexity to offer context for our evaluation. The metamodel consists of ≈ 50 classifiers, with abstract classes forming a strong foundation for inheritance, and includes ≈ 80 structural features. By utilizing multiple inheritance and nesting up to three layers, the model establishes a clear and effective abstraction hierarchy.

7.1. Defining the Behavioral View

The initial phase of the evaluation process involved the definition of a view representing the behavioral aspect of Art. This was accomplished by selecting elements that specifically characterize the system's dynamic behavior. To ensure the validity of the resulting view metamodel, the solution automatically included several specialized classes and structural elements, with capsules being a key example. Capsules are crucial because they provide the runtime context for state machine execution. In the metamodel, state machines are contained within capsules, which, according to Rule 5 in the related work, means that both the containing class (the capsule) and its corresponding containment reference must be included. This necessity became even more evident when elements were selected without enabling the automatic consistency check. Attempts to finalize the

[11] https://secure-dev-ops.github.io/code-realtime/

[12] https://secure-dev-ops.github.io/code-realtime/art-lang/

view metamodel under these conditions resulted in errors, specifically indicating that `EClass: Capsule is required by EClass: TopStateMachine due to containment dependencies`. Consequently, the inclusion of additional required elements ensures that behavioral constructs maintain correct referencing and inheritance relationships, thereby producing a valid metamodel.

7.2. Role and Permission Definition

As shown in Figure 9, we defined three key roles, each representing a typical stakeholder in a real-time system modeling scenario:

- *System Architects:* responsible for describing the overall architecture of the system. Have full access to all instances of behavioral elements. In addition, they have full access control over instances of structural elements automatically included in the behavioral view, such as capsules, allowing them to make the necessary adjustments to both structural and behavioral aspects.
- *Behavioral Part Architects*: responsible for describing behavior. Have full access to all instances of behavioral concepts but only read access to structural ones like capsules that were automatically added. This ensures that the structural configurations defined by the system architects remain unchanged.
- *Testers:* responsible for validating system behavior against expected outcomes. Have only read access to instances of both behavioral and structural elements, enabling them to analyze state transitions, verify behavioral logic, and ensure traceability between test cases and behavioral specifications without modifying the models.

During the permission definition process, the solution automatically determined which additional, interdependent elements required selection of permissions based on dynamic consistency checks. Although this automation may appear trivial in terms of reducing individual clicks, it brings a significant benefit being that the person in charge of defining permissions does not have to manually trace and assess dependencies. This not only streamlines the overall configuration process but also reduces the likelihood of missing or incorrectly assigning permissions, ultimately resulting in a more reliable and maintainable permission setup. Figure 10 illustrates the permission settings for the view, detailing the access rights for three distinct user roles. The System Architect is granted full permissions for all view elements. In contrast, the Behavioral Architect has full access to behavioral components but read-only access to structural ones. The Tester

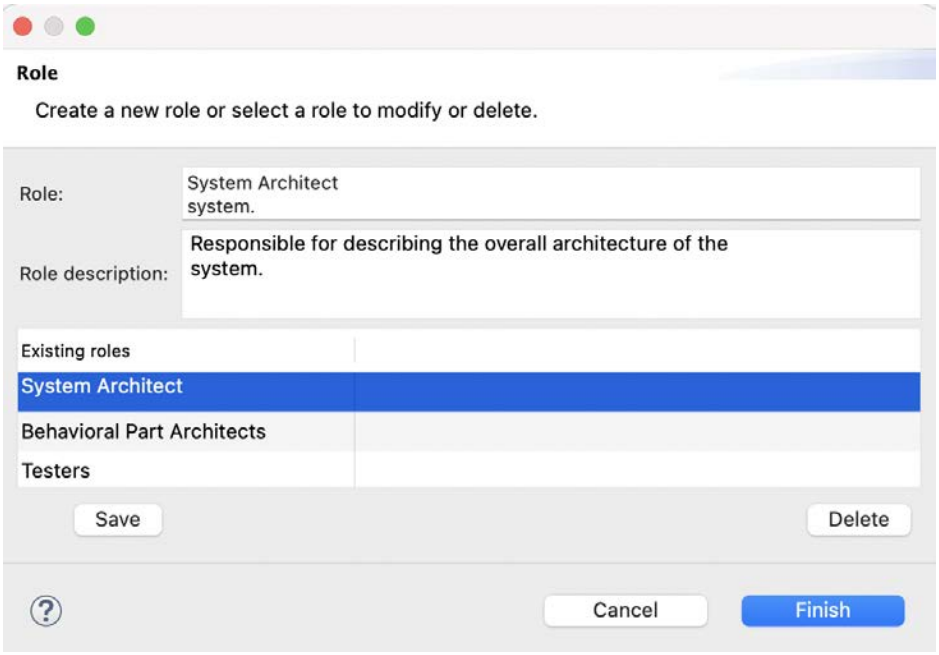


Figure 9: Role definition wizard

role is the most restricted, with read-only access to the entire view. Note that the column for the key identifier attribute is empty. This is because the elements shown do not define this attribute in their own classes. Instead, they inherit it from the abstract NamedElement class. Consequently, the identifier must be configured directly on the NamedElement definition, which is located in a different section of the wizard not visible in this excerpt.

7.3. Generation of Artifacts

Upon finalizing the configuration in the permission definition wizard (Figure 10), pressing the Finish button automatically generates the following primary artifacts:

1. *View Metamodel:* the generated view metamodel for the behavioral part of Art includes 31 classifiers and 27 structural features, with key abstract classes forming a clear inheritance hierarchy. It supports hierarchical behavioral modeling with nested structures reaching a depth of three levels. Validation checks confirmed the metamodel's correctness, ensuring consistency and completeness in representing the behavioral view.

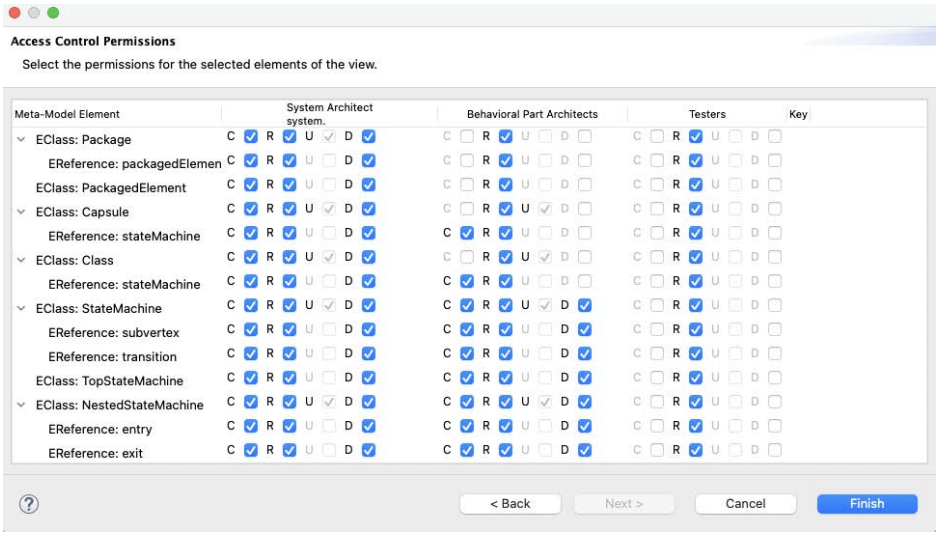


Figure 10: Permissions' definition wizard

2. *Synchronization Infrastructure:* the approach generates two model transformations to ensure consistency between the base model conforming to the base metamodel of Art, and the subset model conforming to the generated view metamodel of the behavioral part of Art. The first transformation extracts and maps relevant elements from the base model to create a simplified subset, requiring $\approx$ 180 lines of code. The second transformation, which propagates changes from the subset model back to the base model, is more complex, with $\approx$ 800 lines of code. This increased complexity is due to the need to synchronize changes without overwriting or losing information present in the base model but absent from the subset. It involves detailed checks to manage additions, updates, and deletions, as well as logic to preserve relationships and maintain model consistency. The synchronization infrastructure ensures that any changes made in either model are accurately reflected in the other without loss of information. Listing 2 shows an excerpt of the in-place transformation (Pkg2Pkg, lines 1-30) that synchronizes a Package element in a base model with its counterpart from a view model. The logic begins by identifying and selecting the corresponding view package for processing (lines 7-8). The transformation then handles additions and updates by iterating through the elements of the view package (lines 9-23). If an element from the view (viewElem) is not found in the base model, a new Capsule element is instantiated and added (lines 10-18).

Conversely, if an element already exists in both models, the transformation delegates to a more specific mapping (syncElement) to recursively synchronize its internal properties (lines 19-22). Finally, the routine handles the removal of obsolete elements. The packagedElement collection is updated by retaining only those elements from the base model that also exist in the view model (lines 24-26). This process ensures any element not present in the view is removed from the base model. The application of these model transformations to available Code RealTime Art sample models confirmed their correctness, demonstrating accurate synchronization and consistency between the base and subset models.

```
1  mapping artlang::Package::Pkg2Pkg() : artlang::Package
2  inherits artlang::NamedElement::syncNamedElem
3  {
4      init {
5          result := self;
6      }
7      if (viewModel.objectsOfType(view::Package)->exists(p | p.id = self.id)) then {
8          viewModel.objectsOfType(view::Package)->forOne(viewPkg | viewPkg.id = self.id) {
9              viewPkg.packagedElement->forEach(viewElem) {
10                 if (self.packagedElement->exists(baseElem | baseElem.id = viewElem.id) = false) {
11                     if (viewElem.oclIsTypeOf(view::Capsule)) {
12                         var newElem := object artlang::Capsule {
13                             name := viewElem.name;
14                             id := viewElem.id;
15                         };
16                         result.packagedElement += newElem;
17                     }
18                     ... // other types
19                 }
20                 else {
21                     self.packagedElement->selectOne(e | e.id = viewElem.id).map syncElement();
22                 }
23             };
24             result.packagedElement := self.packagedElement->select(baseElem |
25                 viewPkg.packagedElement->exists(v | v.id = baseElem.id)
26             );
27         }
28     }
29     endif;
30 }
```

Listing 2: View2Base.qvto trasformation excerpt for Package element

3. *Customized Editors:* three EMF tree-based editors were automatically generated, one for each defined role, to enforce the specified access permissions. Each editor's implementation consists of 33 files: 31 corresponding to the subset metamodel's classifiers and two additional files for the edit plugin and provider adapter factory. For example, the editor generated for the tester role, which requires read only access to all elements, shows that all 31 classifier related files are automatically adapted according to the access control model. As a result, any operations for creating, modifying, or deleting instances of these classifiers or their structural features are disabled in the generated editor. Listing 3 shows an excerpt from the Capsule's ItemProvider file, highlighting the `createRemoveCommand()`, which disables

deletion rights for the tester role.

```
@Override
public Command createRemoveCommand(EditingDomain editingDomain, EObject owner, EStructuralFeature
      feature, Collection<?> collection) {
    if (owner instanceof Capsule) {
        for (Object object : collection) {
            if (object instanceof TopStateMachine) {
                return UnexecutableCommand.INSTANCE;
            }
        }
    }
    return RemoveCommand.create(editingDomain, owner, feature, collection);
}
```

Listing 3: createRemoveCommand() automatically added to the Capsule ItemProvider to disable deletion rights for the tester role

The evaluation demonstrates that the approach effectively automates three critical aspects: generating a valid subset metamodel, establishing synchronization between the base and subset models, and enforcing role-based access permissions through customized model editors. The process requires minimal manual input. Users select the elements to be included in the subset, with the system automatically incorporating all additional elements necessary to ensure metamodel validity. Similarly, when defining access permissions, real time consistency checks automatically adjust related operations and elements to maintain coherent permission structures. The synchronization infrastructure, also generated automatically, ensures that any modifications in the subset model are accurately reflected in the base model and vice versa, without information loss. Manually performing these tasks would be considerably more complex and error-prone, involving detailed analysis to maintain metamodel validity, the implementation of intricate synchronization logic to guarantee consistency, and the customization of model editors to enforce permissions. The approach significantly reduces this complexity, providing a reliable and efficient solution for defining and managing access-controlled views.

8. Discussion

Benefits of the proposed solution. The proposed solution enhances the confidentiality and integrity of model information in collaborative modeling environments. Simultaneously, it aims to streamline the individual modeling experience by minimizing the information overload that results from interacting with a comprehensive base model. The latter is a result of offering customized view (meta)models, allowing users to engage with only the relevant aspects of the base (meta)model. The solution employs the RBAC policy and supports the definition of fine-grained

access permissions using CRUD operations. Central to the novelty of the solution is a process that addresses both the consistency of the security policy and its enforcement. Firstly, a set of automatic consistency rules prevents the definition of contradictory permissions, mitigating the risk of human error at the policy creation stage. Secondly, these verified and consistent security rules are then used to drive a modified code generation process. This represents a fundamental change to the standard EMF implementation, enabling "enforcement by construction". Rather than requiring manual and error-prone security coding, the permissions are automatically embedded into the editor's source code as it is created, making the final tool inherently secure by design. This tight integration of automated consistency checking and proactive enforcement not only ensures the integrity of the security policy but also provides significant flexibility. Any changes in permissions are immediately and correctly reflected in the model editor, allowing the system to respond quickly to changes in project requirements and team composition. Incorporating these capabilities, the solution effectively meets the technical requirements for a fine-grained, consistent, and flexible access control system. It safeguards shared model information by mitigating the risks of confidentiality breaches and ensuring the integrity of model data through its automated enforcement mechanisms. These technical features are designed with the goal of improving the collaborative process. By supporting users to perform efficiently in their designated sections of a shared model, the approach is intended to foster a more effective environment for teamwork and improve overall productivity.

Limitations of the proposed solution. The solution is confined to Ecore-based metamodels within the Eclipse Modeling Framework (EMF). This dependency on a specific technological space, while enabling us to leverage a rich tooling ecosystem, restricts the direct application of our implementation to other metametamodeling languages or platforms. A primary limitation comes from the enforcement mechanism, which is currently coupled with EMF's default tree-based editors. This presents a conflict when considering its application to graphically oriented standard notations like UML or SysML. Although our approach is fully applicable to the underlying data structure of these notations, given the use of their standardized Ecore representations, the security enforcement does not automatically extend to their graphical notation. For instance, in an Eclipse-based tool like Papyrus, our generated secure tree editor could operate alongside the graphical editor, but permissions would not be enforced on the diagrams themselves. For non-Eclipse tools like MagicDraw or Enterprise Architect, a direct integration is not feasible. The approach would need to be reimplemented from the ground up

on that platform. Furthermore, beyond technical integration, applying our work to general-purpose languages like UML or SysML introduces a methodological trade-off. The standard practice for extending these languages is through profiles, which avoids invalidating the core language. However, when used for rights management, this approach typically relies on post-operation validation. To replicate the proactive prevention offered by our solution, a standard tool would need to be enhanced with a non-standard mechanism to interpret the edit rights profiles proactively, or one could adopt alternative research techniques like metamodel pruning [33] to achieve an outcome comparable to the mechanisms proposed in our approach. Despite platform-specific challenges, we consider the conceptual blueprint of our work to remain transferable. The fundamental idea of using materialized views to govern coarse-grained access, followed by the enforcement of fine-grained permissions at the editor level, is a strategy that could be reimplemented in any sufficiently extensible modeling tool. Broadening support to various model editors and graphical notations is a clear direction for future enhancement.

On another note, the current implementation focuses on the authorization aspect of access control, with authentication being a separate concern that has not yet been addressed. The practical implication is that, as of now, the system does not restrict user access to the various role-specific model editors upon login. The authorization logic is independent of any specific authentication method, which allows for different mechanisms to be integrated in the future. Addressing this marks a clear path for future enhancement, where an authentication layer that could range from traditional username/password systems to integration with enterprise-level identity providers could be introduced. More forward-looking approaches, such as the use of Decentralized Identifiers (DID), would be particularly well-suited, offering greater user autonomy and security in distributed, collaborative modeling environments. In terms of synchronization infrastructure, our approach relies on user-selected unique identifiers for element matching, which poses a risk of inconsistencies. Additionally, the synchronization process is manually triggered, highlighting the need for automated change detection and conflict management strategies which fall outside the scope of this paper. However, the existing body of literature on conflict management provides a solid foundation for future research in this area [34]. Overall, these limitations, mostly of implementative nature, reflect our choice to focus on access control related aspects in the scope of this initial solution, laying the groundwork for future improvements.

Threats to validity. Our evaluation combines lightweight toy examples used to exercise every core feature of the plugin and a case study with the industry-adopted Art language, demonstrating both conceptual soundness and practical viability. However, we acknowledge several threats to the validity of our findings [35]. With respect to internal validity, researcher bias may have influenced our examples and case-study design, potentially favoring scenarios that showcase the strengths of our approach. By publishing all artifacts and inviting independent replication, we lay the groundwork for objective scrutiny and enable any researcher or practitioner to replicate our results, inspect our implementation, and extend our approach. With respect to construct validity, we have not yet conducted formal end-user studies, so our evaluation does not yet capture formal usability or efficiency. Our validation strategy was to first establish the technical feasibility and correctness of our framework, as we believe this is a necessary prerequisite for a meaningful user study. Our case study with the industry-adopted Art language served this purpose, confirming that the core mechanisms function correctly on a complex model. Therefore, while benefits like enhanced security are verifiable technical outcomes of our design, other user-oriented claims (e.g., streamlining the modeling experience and enhancing collaborative efficiency) are posited as direct consequences of our technical features and their adherence to well-known usability principles. We acknowledge that while our approach is designed to be user-friendly, a formal usability study is required to quantitatively measure its impact on user productivity and satisfaction. To confirm and refine our assumptions, we plan structured usability experiments measuring task completion time, error rates, and subjective satisfaction, and pilot deployments with industry partners to observe how the tool performs in practice and identify any adoption challenges. With respect to external validity, our evaluation relies on a single, industrial case study using the Art language. While this demonstrates practical viability, further case studies across different domains and with varied modeling language characteristics are necessary to build a stronger case for generalizability.

9. Conclusions and Future Work

This article proposes a dual-layered approach that provides role-based access control to ensure the confidentiality and integrity of model information in collaborative modeling environments. The first layer focuses on the creation of view (meta)models on top of a base (meta)model, along with a seamless synchronization mechanism between them. This efficiently restricts access to the base model, limiting users to interact solely with the view models assigned to their respective

roles. The second layer fine-tunes this access by establishing fine-grained access permissions describing the permissible operations over each element type within the view models. Each building block of the solution was either substantially extended or developed anew, and these pieces are glued into a single, automated pipeline. Key contributions include:

- Automated, in-place synchronization: We extended existing solution by: i) automatically generating mapping models between view and base models, and by ii) defining higher-order transformation (HOT) templates to automatically consume the derived mapping models and produce QVTo scripts that preserve unmapped elements. This eliminates manual mapping and prevents data loss.

- Live policy consistency engine: We designed and implemented from scratch a set of 23 formal CRUD consitency rules, enforcing them on-the-fly within the access-permission wizard. Administrators receive immediate feedback on conflicting or incomplete policies, ensuring correctness before any code is generated.

- Security-aware editor generation: By replacing EMF's default static JET templates with custom, RBAC-driven templates, our framework emits modeling editors that enforce access constraints by construction. The resulting editors require no post-hoc hand-coded checks.

- End-to-end automation: Launching the view-and-permission wizards triggers an automatic process that assembles the standalone view metamodel, dynamic GenModel, RBAC specification, synchronization transformations, and role-specific editors, fully eliminating manual intervention between slicing, synchronization, and enforcement.

Our end-to-end solution spanning view definition, bidirectional synchronization, and role-based access control specification and enforcement provides a level of integration and automation required in industrial modeling projects.

For future work, we plan to extend the solution to apply to blended editors. Moreover, we aim to extend this research by focusing on establishing permissions at the instance level, assigning permissions to each instance, rather than applying a one-size-fits-all rule to all instances of a given meta element. Future work will also combine the RBAC policy with the attribute-based access control (ABAC) policy, adding restrictions based on attributes, since the current approach can be

easily adapted to include additional access control policies without altering the underlying framework for defining and enforcing permissions. For example, in ABAC, for attributes that are static and manually defined this can be achieved by: (i) establishing a method to define attributes and reference them during permission specification, and (ii) implementing a method to extract attribute values and enforce permissions accordingly. Attributes that are dynamic, like location, necessitate distinct mechanisms for detection and value extraction. The methodologies proposed by [24] offer valuable insights into addressing the latter scenario. In addition, in future work, the adaptability of the theoretical approach is intended to be explored across various modeling frameworks beyond EMF. Our analysis into the dependencies among model elements and CRUD operations has yielded reusable consistency rules applicable across various technological stacks, potentially requiring only minimal adjustments. Moreover, the strategy for enforcing permissions by utilizing predefined permissions as inputs in the model editor generation process can be extended to other modeling platforms. However, a more in depth examination of existing modeling platforms is essential. Finally, upcoming efforts will be dedicated to assessing the proposed solution in industrial contexts.

References

[1] D. C. Schmidt, Model-driven engineering, Computer-IEEE Computer Society-39 (2) (2006) 25.

[2] R. France, B. Rumpe, Model-driven development of complex software: A research roadmap, in: Future of Software Engineering (FOSE'07), IEEE, 2007, pp. 37–54.

[3] J. Whitehead, Collaboration in software engineering: A roadmap, in: Future of Software Engineering (FOSE'07), IEEE, 2007, pp. 214–225.

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

[5] H. Muccini, J. Bosch, A. van der Hoek, Collaborative modeling in software engineering, IEEE Software 35 (6) (2018) 20–24.

[6] D. Di Ruscio, M. Franzago, I. Malavolta, H. Muccini, Envisioning the future of collaborative model-driven software engineering, in: 2017 IEEE/ACM 39th International Conference on Software Engineering Companion (ICSE-C), IEEE, 2017, pp. 219–221.

[7] I. David, K. Aslam, I. Malavolta, P. Lago, Collaborative model-driven software engineering—a systematic survey of practices and needs in industry, Journal of Systems and Software 199 (2023) 111626.

[8] M. Nieles, K. Dempsey, V. Y. Pillitteri, An introduction to information security, NIST special publication 800 (12) (2017) 101.

[9] S. Martínez, A. Fouche, S. Gérard, J. Cabot, Automatic generation of security compliant (virtual) model views, in: Conceptual Modeling: 37th International Conference, ER 2018, Xi'an, China, October 22–25, 2018, Proceedings 37, Springer, 2018, pp. 109–117.

[10] C. Debreceni, G. Bergmann, I. Ráth, D. Varró, Enforcing fine-grained access control for secure collaborative modelling using bidirectional transformations, Software & Systems Modeling 18 (2019) 1737–1769.

[11] R. S. Sandhu, Role-based access control, in: Advances in computers, Vol. 46, Elsevier, 1998, pp. 237–286.

[12] D. Steinberg, F. Budinsky, E. Merks, M. Paternostro, EMF: Eclipse Modeling Framework, Pearson Education, 2008.

[13] A. Cicchetti, F. Ciccozzi, A. Pierantonio, Multi-view approaches for software and system modelling: a systematic literature review, Software and Systems Modeling 18 (2019) 3207–3233.

[14] ISO/IEC/IEEE, Systems and software engineering–architecture description, ISO/IEC/IEEE 42010: 2011 (E)(Revision of ISO/IEC 42010: 2007 and IEEE Std 1471-2000) 2011 (2011) 1–46.

[15] A. Cicchetti, F. Ciccozzi, T. Leveque, Supporting incremental synchronization in hybrid multi-view modelling, Models in Software Engineering (2012) 89–103.

[16] H. C. v. Tilborg, S. Jajodia, Encyclopedia of Cryptography and Security (2011).

[17] R. Sandhu, D. Ferraiolo, R. Kuhn, The NIST model for role-based access control: Towards a unified standard, in: ACM workshop on role-based access control, Vol. 10, 2000.

[18] V. C. Hu, R. Kuhn, D. Yaga, Verification and test methods for access control policies/models, NIST Special Publication 800 (2017) 192.

[19] A. Cicchetti, F. Ciccozzi, T. Leveque, A hybrid approach for multi-view modeling, Electronic Communications of the EASST 50 (2012).

[20] H. Bruneliere, J. G. Perez, M. Wimmer, J. Cabot, EMF Views: A view mechanism for integrating heterogeneous models, in: 34th International Conference on Conceptual Modeling (ER 2015), 2015.

[21] G. Bergmann, C. Debreceni, I. Ráth, D. Varró, Query-based access control for secure collaborative modeling using bidirectional transformations, in: Proceedings of the ACM/IEEE 19th International Conference on Model Driven Engineering Languages and Systems, 2016, pp. 351–361.

[22] C. Debreceni, G. Bergmann, M. Búr, I. Ráth, D. Varró, The MONDO collaboration framework: secure collaborative modeling over existing version control systems, in: Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering, 2017, pp. 984–988.

[23] C. Debreceni, G. Bergmann, I. Ráth, D. Varró, Secure views for collaborative modeling, IEEE Software 35 (6) (2018) 32–38.

[24] L. Brunschwig, E. Guerra, J. de Lara, Towards access control for collaborative modelling apps, in: Proceedings of the 23rd ACM/IEEE International Conference on Model Driven Engineering Languages and Systems: Companion Proceedings, 2020, pp. 1–10.

[25] A. Anjorin, S. Rose, F. Deckwerth, A. Schürr, Efficient model synchronization with view triple graph grammars, in: Modelling Foundations and Applications: 10th European Conference, ECMFA 2014, Springer, 2014, pp. 1–17.

[26] J. Jakob, A. Königs, A. Schürr, Non-materialized model view specification with triple graph grammars, in: Graph Transformations: Third International Conference, ICGT 2006 Natal, Rio Grande do Norte, Brazil, September 17-23, 2006 Proceedings 3, Springer, 2006, pp. 321–335.

[27] J. N. Foster, M. B. Greenwald, J. T. Moore, B. C. Pierce, A. Schmitt, Combinators for bidirectional tree transformations: A linguistic approach to the view-update problem, ACM Transactions on Programming Languages and Systems (TOPLAS) 29 (3) (2007) 17–es.

[28] M. Latifaj, F. Ciccozzi, M. Mohlin, Higher-order transformations for the generation of synchronization infrastructures in blended modeling, Frontiers in Computer Science 4 (2023) 1008062.

[29] M. Tisi, F. Jouault, P. Fraternali, S. Ceri, J. Bézivin, On the use of higher-order model transformations, in: European Conference on Model Driven Architecture-Foundations and Applications, Springer, 2009, pp. 18–33.

[30] J. Martin, Managing the data base environment, Prentice Hall PTR, 1983.

[31] B. Selic, G. Gullekson, J. McGee, I. Engelberg, Room: An object-oriented methodology for developing real-time systems, in: CASE'92 Fifth International Workshop on Computer-Aided Software Engineering, 1992, pp. 230–240.

[32] M. Latifaj, F. Ciccozzi, M. W. Anwar, M. Mohlin, Blended graphical and textual modelling of UML-RT state-machines: An industrial experience, in: European Conference on Software Architecture, Springer, 2021, pp. 22–44.

[33] S. Sen, N. Moha, B. Baudry, J.-M. Jézéquel, Meta-model pruning, in: International Conference on Model Driven Engineering Languages and Systems, Springer, 2009, pp. 32–46.

[34] M. Sharbaf, B. Zamani, G. Sunyé, Conflict management techniques for model merging: a systematic mapping review, Software and Systems Modeling 22 (3) (2023) 1031–1079.

[35] C. Wohlin, Guidelines for snowballing in systematic literature studies and a replication in software engineering, in: Proceedings of the 18th international conference on evaluation and assessment in software engineering, 2014, pp. 1–10.