Noname manuscript No.
(will be inserted by the editor)

Enforcing Fine-grained Access Control for Secure Collaborative Modeling using Bidirectional Transformations

Csaba Debreceni · Gábor Bergmann · István Ráth · Dániel Varró

Received: date / Accepted: date

Abstract Large-scale model-driven system engineering projects are carried out collaboratively. Engineering artifacts stored in model repositories are developed in either offline (checkout-modify-commit) or online (GoogleDoc-style) scenarios. Complex systems frequently integrate models and components developed by different teams, vendors and suppliers. Thus confidentiality and integrity of design artifacts need to be protected in accordance with access control policies.

We propose a secure collaborative modeling approach where fine-grained access control for models is strictly enforced by bidirectional model transformations. Collaborators obtain filtered local copies of the model containing only those model elements which they are allowed to read; write access control policies are checked on the server upon submitting model changes.

We present a formal collaboration schema which provenly guarantees certain correctness constraints, and its adaption to online scenarios with on-the-fly change propagation and the integration into existing version control systems to support offline scenarios. The approach is illustrated and its scalability is evaluated using a case study of the MONDO EU project.

Keywords collaborative modeling · secured views · access control · online collaboration · offline collaboration · bidirectional model transformation

Csaba Debreceni[1,2] · Gábor Bergmann[1,2] · István Ráth[1] · Dániel Varró[1,2,3]
[1]Budapest University of Technology and Economics, Department of Measurement and Information Systems, H-1117 Magyar tudósok krt. 2, Budapest, Hungary;
[2]MTA-BME Lendület Research Group on Cyber-Physical Systems;
[3]McGill University, Dept. of Electrical and Computer Engineering
E-mail: {debreceni, bergmann, rath, varro}@mit.bme.hu

1 Introduction

1.1 Collaborative modeling in MDE

The adoption of model driven engineering (MDE) by system integrators (like airframers or car manufacturers) has been steadily increasing in the recent years [55], since it enables to detect design flaws early and generate various artifacts (source code, documentation, configuration tables, etc.) automatically from high-quality system models.

The use of models also intensifies collaboration between distributed teams of different stakeholders (system integrators, software engineers of component providers/suppliers, hardware engineers, certification authorities, etc.) via model repositories, which significantly enhances productivity and reduces time to market. An emerging industrial practice of system integrators is to outsource the development of various design artifacts to subcontractors in an architecture-driven supply chain.

Collaboration scenarios include traditional *offline collaborations* with asynchronous long transactions (i.e. to check out an artifact from a version control system and commit local changes afterwards) as well as *online collaborations* with short and synchronous transactions (e.g. when a group of collaborators simultaneously edit a model, similarly to well-known on-line document / spreadsheet editors). Several collaborative modeling frameworks (like CDO [49], EMFStore [50], etc.) exist to support such scenarios.

However, such collaborative scenarios introduce significant challenges for security management in order to protect the Intellectual Property Rights (IPR) of different parties. For instance, the detailed internal design of a specific component needs to be hidden to competitors who might supply a different component in the overall system, but needs to be revealed to certification authorities in order to obtain airworthiness. Large research projects in the avionics domain (like CESAR [1] or SAVI [4]) address certain collaborative aspects of the design process (e.g. by assuming multiple subcontractors), but security aspects are restricted to that of the system under design.

An increased level of collaboration in a model-driven development process introduces additional confidentiality challenges to sufficiently protect the IPR of the collaborating parties, which are either overlooked or significantly underestimated by existing initiatives. Even within a single company, there are often teams with differentiated responsibilities, areas of competence and clearances. Such processes likewise demand confidentiality and integrity of certain modeling artifacts.

1.2 Problems of coarse-grained access control

Existing practices for managing access control of models rely primarily upon the access control features of the back-end repository. *Coarse-grained access control policies* aim to restrict access to the files that store models. For instance, EMF models can be persisted as standard XMI documents, which can be stored in repositories providing file-based access and change management (as in SVN [5], CVS [26]). *Fine-grained access control policies*, on the other hand, may restrict access to the model on the row level (as in relational databases) or triple level (as in RDF repositories). Unfortunately, coarse-grained security policies are captured directly on the storage (file) level often result in inflexible fragmentation of models in collaborative scenarios.

To illustrate the problem of coarse-grained permissions, let us consider two collaborators, *SW Provider*$_1$ and *HW Supplier*$_1$ having full control over their model (fragment). Now if *HW Supplier*$_1$ intends to share part of their model with *SW Provider*$_1$, then either they need to grant access to the entire model (which would mean losing the confidentiality of certain intellectual properties), or split their model into two files, and give access to only one fragment. For each additional actor *SW Provider*$_2$, the same argument applies; in the end, a collaboratively developed system model would end up being split into several fragments.

Even in the simple case depicted in Fig. 1, the model needs to be split into two files (*Model Fragment*$_1$ and *Model Fragment*$_2$) and access needs to be granted separately for each file when a *SW Provider*$_1$ and a *HW Supplier*$_1$ collaborates. When a new collaborator, *SW Provider*$_2$ joins in the future who is allowed to partially read all two existing fragments, each model fragment needs to be divided at least in two. In this example, 5 fragments are required: one that can be read by both *SW Provider*$_1$ ⇔ *SW Provider*$_2$, one accessible to *HW Supplier*$_2$ ⇔ *SW Provider*$_2$, and three more private model fragments for the three collaborators. If additional collaborators join the collaboration, the number of fragments has to be increased further.

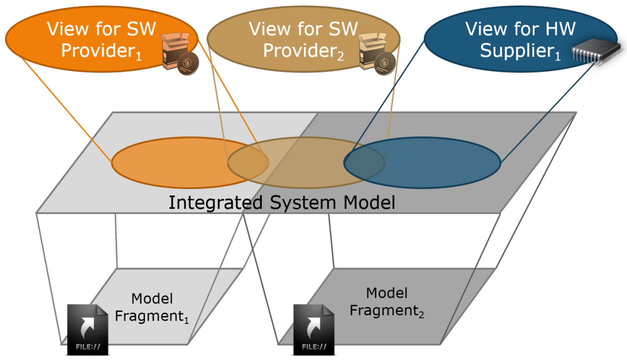


Fig. 1: Problem with File-level Access Control

The example is updated in accordance with Fig. 1

As a result, *coarse-grained access control can lead to significant model fragmentation*, which greatly increases the complexity of storage and access control management. In industrial practice, automotive models may be split into more than 1000 fragments, which poses a significant challenge for tool developers. Some model persistence technologies (such as EMF's default XMI serialization) do not allow model fragments to cyclically refer to each other, putting a stricter limit to fragmentation. Hence, MDE use cases often demand the ability to define access for each object (or even each property of each object) independently.

Furthermore, *coarse-grained access control lacks flexibility*, especially when accessing models from heterogeneous information sources in different collaboration scenarios. For instance, they disallow type-specific access control, i.e., to grant or restrict access to model elements of a specific type (e.g., to all classes in a UML model), which are stored in multiple files.

On the other hand, *fine-grained access control necessitates to assign access rights to each model element.* As the size of the model grows, these permissions or restrictions cannot be set and maintained manually for each individual model element, but a systematic assignment technique is needed.

1.3 Goals

The main objective of the paper is to achieve secure collaborative modeling with fine-grained access control, by using advanced model transformation techniques, while relying upon existing storage back-ends to follow current industrial best practices. In particular, we aim to address the following high-level goals (refined later into technical goals in Sec. 2):

G1 *Fine-grained Access Control Management* to enforce read and write permissions of users separately to each model object, attribute or reference.

G2 *Secure and Versatile Offline Collaboration* where each collaborator can work with a model fragment filtered in accordance with read permissions, and processed using off-the-shelf MDE tools (e.g. editor, verifier). A user may be disconnected from any server or access control mechanism, and then submit (commit) his updated version in the end.

G3 *Secure and Efficient Online Collaboration* where multiple users can view and edit a model hosted on a server repository in real-time while imposing different read and write permissions. Small changes performed by one collaborator are quickly and efficiently propagated to the views visible to other users, without reinterpreting the entire model.

G4 *General Collaboration Schema* that is adaptable for online and offline scenario defining workflows of a server and multiple clients to handle fine-grained access control management.

1.4 Contributions

In this paper, we define an approach for *secure collaborative modeling* using *bidirectional model transformations* to derive *filtered secure views* for each collaborator and to propagate changes introduced into these views back to a server. Our approach is uniformly applicable to support both *online and offline collaboration scenarios*, and it enforces fine-grained access control policies for each collaborator during the *derivation of views* and the *back-propagation* of changes.

We *formalize the collaboration schema* using communicating state machines and *provide formal proofs for certain correctness criteria* using the *FDR4 tool* [32]. The schema is *integrated into existing version control systems* using *hook* programs triggered by repository events to support offline collaborative scenarios whereas *a prototype tool of online collaboration* is also realized on the top of *Eclipse RAP* [51].

Finally, a detailed scalability evaluation is carried out using models from the Wind Turbine Case Study of the MONDO European FP7 project, which serves as a motivating example for the paper.

This paper is an extension of [11, 20] by providing (1) an in-depth precise specification of the bidirectional transformation as well as the collaboration scheme, (2) further technical details on its realization for both offline and online collaborative scenarios, and (3) an extended scalability evaluation of our approach now also covering the offline scenario.

1.5 Structure

Rest of the paper is organized as follows. Our motivating example is detailed and the challenges are introduced in Sec. 2. Sec. 3 defines how models can be decomposed into individual assests, and introduces the rules that assign read and write permissions to assets. In Sec. 4, we overview our bidirectional model transformation for access control, while Sec. 5 describes our secure collaboration schema and proves its correctness. In Sec. 6, we give a brief overview on how to adapt this collaborative modeling schema to online and offline scenarios. Sec. 7 describes the evaluation of our approach and related work is overviewed in Sec. 8. Finally, Sec. 9 concludes our paper.

2 Case Study

2.1 Modeling Language

Our approach will be illustrated using a simplified version of a modeling language for system integrators of offshore wind turbine controllers, which served as one of the case studies of the MONDO EU FP7 project [7]. The metamodel, defined in Ecore [52] and depicted in Fig. 2, describes how the system builds up from modules (Module) providing and consuming signals (Signal) that send messages after a specific amount of time defined by the frequency attribute. Modules are organized in a containment hierarchy of composite modules (Composite) shipped by external vendors (vendor attribute), and ultimately containing control unit modules (Control) responsible for a given type of physical device (such as pumps, heaters or fans: FanControl, HeaterControl, PumpControl, respectively) with specific cycle priorities (cycle attribute). A documentation is attached to each signal (documentation attribute) to clarify its responsibilities. Some of the signals are treated as confidential intellectual property (ConfidentialSignal).

The design of wind turbine control units requires specialized knowledge. There are three kinds of control units, and each kind can only be modified by specialist

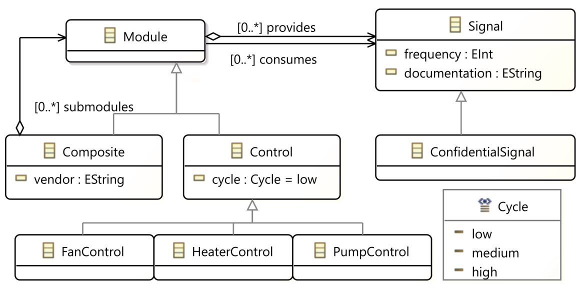


Fig. 2: Simplified Metamodel of Wind Turbine Controllers

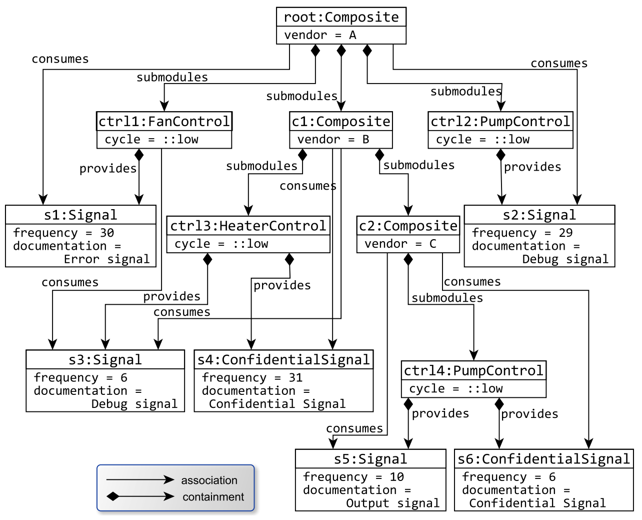


Fig. 3: Sample Wind Turbine Instance Model

users with the appropriate qualification: *fan*, *heater* and *pump control engineers*.

A sample instance model containing a hierarchy of 3 `Composite` modules with 4 `Control` units as submodules, providing 6 `Signals` altogether where two of them are `Confidential Signals`, is shown on Fig. 3. Boxes represent objects (with attribute values as entries within the box and their types shown as labels on the tops). Arrows with diamonds represent containment edges, while arrows without diamonds represent cross-references.

2.2 Security Requirements

Specialists are not allowed to modify (and in some cases, read) parts of the model. For this purpose, the following security requirements are stated for control unit specialists:

R1 Each group of specialists shall be responsible for a specific kind of control unit (*owned control units*).

R2 Specialists shall see only those signals that are within the scope for their owned control units, i.e. signals provided by a module that is either (a) a composite that directly contains an owned control unit, or (b) any submodule (including the owned control unit) contained *transitively* in such a composite.

R3 Specialists shall be able to modify signals provided by their owned control units.

R4 Specialists shall observe which modules consume signals provided by their owned control units.

R5 Specialists shall see the vendor attributes in an obfuscated form.

R6 Specialists must not see confidential signals.

2.3 Usage Scenarios

The system integrator company is hosting the wind turbine control model on their collaboration server, where it is stored, versioned, etc. There are two ways for users to interact with it.

Online collaboration. A group of users may participate in online collaboration, when they are continuously connected to the central repository via an appropriate client (e.g. web browser). Each user sees a live view of those parts of the model, that he is allowed to access. Changes need to be propagated on-the-fly between the views of users in short transactions. These transactions contain each modification such as create, update, delete or move. Finally, the collaboration tool has to reject a modification immediately when it violates a security requirement.

The users can modify the model through their client, which will directly forward the change to the collaboration server. The server will decide whether the change is permitted under write access restrictions. If it is allowed, then the views of all connected users will be updated transparently and immediately, though the change may be filtered for them according to their read privileges.

Offline collaboration. In case of offline collaboration, when connecting to the server, each user can download a model file containing those model elements that he is allowed to see. The user can then view, process, and modify his downloaded model file locally. The model can be developed with unmodified off-the-shelf tool, that need not be aware of collaboration and access control. After the modification, the changes will be uploaded to the server in a long transaction.

2.4 Challenges

Deriving from the goals stated in Sec. 1.3, we identify the following challenges.

C1 *Fine-grained Access Control of Model Artifacts.*
To meet **G1**, the approach must enforce to allow or deny model access separately for individual model elements.

C2.1 *Model Compatibility.*
To meet **G2** in off-line collaboration scenario, the approach must be able to present the information available to a given user as a self-contained model, in a format that can be stored, processed, displayed and edited by off-the-shelf modeling tools.

C2.2 *Offline Models.*
To meet **G2** in an off-line collaboration scenario, the approach must be able to present only the available information to a given user without maintaining connectivity with any central server or authority responsible for access control.

C3.1 *Incrementality.*
To meet **G3** in an on-line collaboration scenario, the approach must be able to process model modifications initiated by a user and apply the consequences to the views available to other users without re-processing the unchanged parts of the model.

C4.1 *Correctness Criteria.*
To meet **G4**, the approach must define the correctness criteria of the collaboration schema and prove their fulfillment.

C4.2 *Adaptability.*
To meet **G4**, the approach must realize the collaboration schema both in offline and online scenarios.

3 Access Control of Models

3.1 Modeling Preliminaries

In order to tackle challenges **C1.1** and **C2.1**, we first analyze how models can be decomposed into individual *assets* for which access can be permitted and denied, and under what conditions a filtered set of such assets can be represented as a model that can be processed by standard tools.

For the purposes of access control, a model is conceived as a set of elementary *model assets.* An *Asset* is an entity that the access control policy will protect. Generally, models can be decomposed into object, reference and attribute assets.

Definition 1. *Object assets* are pairs formed of a model element with its exact class for each model element object;

$$ObjectAsset = \langle object, type \rangle$$

Definition 2. *Reference assets* are triples formed of a source object, a reference and the referenced target object, for each containment link and cross-link between objects;

$$ReferenceAsset = \langle object_s, reference, object_t \rangle$$

Definition 3. *Attribute assets* are triples formed of a source object, an attribute and a data value, for each (non-default) attribute value assignment;

$$AttributeAsset = \langle object, attribute, value \rangle$$

Definition 4. *Models* are triples formed of a set of object, reference and attribute assets.

$$M = \langle \{ObjectAssset\}, \{ReferenceAsset\}, \{AttributeAsset\} \rangle$$

Note that there can be multi-valued attributes and references in certain modeling platforms (e.g. EMF), where an object is allowed to host multiple attribute values (or reference endpoints) for that property. For such properties, each entry at a source object will be represented by separate attribute (or reference) assets.

Example 1. $ObjectAsset(o1, Composite)$ is an object asset, $AttributeAsset(o10, cycle, low)$ is an attribute asset and $ReferenceAsset(o2, consumes, o12)$ is a reference asset in our running example (depicted in Fig. 3).

3.2 Consistency of Models

An arbitrary set of model assets does not necessarily constitute a valid model; there may be *consistency rules* imposed on the assets by the modeling platform to ensure the integrity of the model representation and the ability to persist, read, and traverse models. Challenge **C2.1** requires that filtered models must be synthesized as a set of model assets compatible with all consistency rules of the underlying modeling platform.

Object Existence. Attributes and references imply that the objects involved exist, having a type compatible with the type of the attribute or reference.

Containment Hierarchy. In modeling languages that have a notion of containment, certain references are denoted as *containment types* realizing a *containment hierarchy* of objects. This hierarchy implies a *containment forest* of all objects. Therefore, objects must either be root objects of the model, or be transitively contained by a root object via a chain of objects that are all existing. (Modeling languages that do not have containment are of course also supported, with all objects considered root objects.)

Opposite Features. There are *opposite* references defined as a pair of references where the existence of a relation depends on its pair. For reference types having an opposite, reference assets of the two types exist in symmetric pairs.

Multiplicity Constraints. The number of reference assets for a given reference of an object needs to satisfy the multiplicity constraints.

The following paragraph (along with a clarification to containment hierarchies) are added to answer the reviewers questions related to containment hierarchy and OCL Constraints.

We distinguish these low-level internal consistency rules from high-level, language-specific *well-formedness constraints.* Well-formedness constraints (also known as design rules or consistency rules) define additional restrictions to the metamodel that the instance models need to satisfy. These type of constraints are often described using OCL [2]. The difference between the two concepts is that violating the latter kind does not prevent a model from being processed and stored in a given modeling technology. Thus only internal consistency is required for access control.

3.3 Model Obfuscation

Obfuscation is defined as the process of *"making something less clear and harder to understand, especially intentionally"* [3]. The first purpose of obfuscation in programming was to distribute C sources in an encrypted way to prevent access to confidential intellectual property in the code [35].

A model obfuscation takes a model as input and yields another model as output where the structure of the model remains the same but data values (such as names, identifiers or other strings) are altered. Two data values that were identical before the obfuscation will also be identical after it, but the obfuscated value computed based on a different input string will be completely different. Moreover, all the altered values can be reverted by the original owner of the model using a private key.

In the context of access control, obfuscation can be applied to data values of attribute assets. An obfuscated data describes its presence in the model (e.g. the value of an object's attribute is not empty), but the real content of that asset remains hidden.

Definition 5. The *obf* function takes a data *Value* and a *Seed* as inputs and maps the value to $(\widehat{Value})$. The Obf^{-1} function is the inverse of *obf* which returns the original data if the same *Seed* is used.

$$obf :: (Value, Seed) \rightarrow \widehat{Value}$$

$$obf^{-1} :: (\widehat{Value}, Seed) \rightarrow Value$$

Example 2. In our example, the security requirement **R5** prescribes to obfuscate the vendor attribute A of object *root* that may become "oA3DD43CF5" in the views.

3.4 Access Control Rules and Permissions

Our fine-grained access control policy has to assign permissions separately for each model asset. In case of a large model, there can be thousands of assets where it is tedious to manually assign permissions one-by-one. Therefore the policies are constructed from a list of *access control rules*, each of which controls the access to a selected set of model assets by certain users or groups, and may either allow or deny the read and/or write operation.

Definition 6. An *access control rule* (*ac-rule*) defines a partial function that applies *judgments* (allow, obfuscate, deny) to specify the privileges of a certain *user* $\in$ *Users* for an *operation type* (read or write) on a given subset of *assets.*

$$\textbf{let } Op = [read, write]$$

$$\textbf{let } Judgement = [allow,\ obfuscate,\ deny]$$

$$ac\text{-}rule :: Assets \times Op \times Users \rightarrow Judgment$$

Definition 7. An *access control policy* defines an effective permission function ($permission_{Eff}$) derived from a list of *access control rules* that applies *judgments* (allow, obfuscate, deny) for both *operation types* (read and write) of each *assets* in the context of a certain *user* $\in$ *Users*

$$permission_{Eff} :: Asset \times Op \times Users \rightarrow Judgement$$

To manage the challenge **C2.1**, it is necessary to eliminate inconsistencies introduced by *access control rules.* In addition, these *access control rules* can be contradictory as one access control rule might grant a permission for a given part of the model while another rule may deny it at the same time.

Added: "... as one access control rule might grant a permission for a given part of the model while another rule may deny it at the same time" to describe how two access control rules can be contradictory

Hence, the effective permission function ($permission_{Eff}$) needs to derive a consistent and conflict-free set of judgments. Our previous work [12,20] describes the effective permission calculation in more detail, but here we give a brief overview on conflict resolution, permission dependencies, and outline some reconciliation strategies as well.

Conflicts. Conflicting policy rules can be resolved by assigning priorities to each rule. Hence, the rules with higher priority overrides the other rules.

Sanity. The sanity of the policy implies that a user should not be allowed to write values and model assets that are not readable to them. Therefore without effective read permission, write permission is automatically denied as well, even if there are no rules to deny the write permissions.

Read dependencies. Read permissions may depend on permissions on other model assets.

If a model element is unreadable, its incoming and outgoing references and its attributes shall not be readable either, otherwise the set of readable assets would not form a self-consistent model.

In modeling platforms (such as EMF) with a notion of containment between objects, readable objects cannot be contained in unreadable objects (as the latter do not exist in the front model); this needs to introduce a new container for the orphan object (e.g. promoting it to a top-level object of the model). Alternatively, this implies that an object hidden from the front model will hide the entire containment subtree rooted there (this latter choice is used in the case study).

Write dependencies. Write permissions likewise have dependencies on other model assets.

In general, creating/modifying/removing references between objects requires a writable source object and a readable target object; but some modeling platforms including EMF have bidirectional references (or opposites), for which internal consistency dictates that the target object must be writable as well.

A metamodel may constrain a reference (or attribute) to be single-valued; assigning a new target to the reference would automatically remove the old one, so a user can only be allowed the former write operation if they are allowed the latter.

Similarly, removing an object from the model implies removing all references pointing to it, and all objects contained within it.

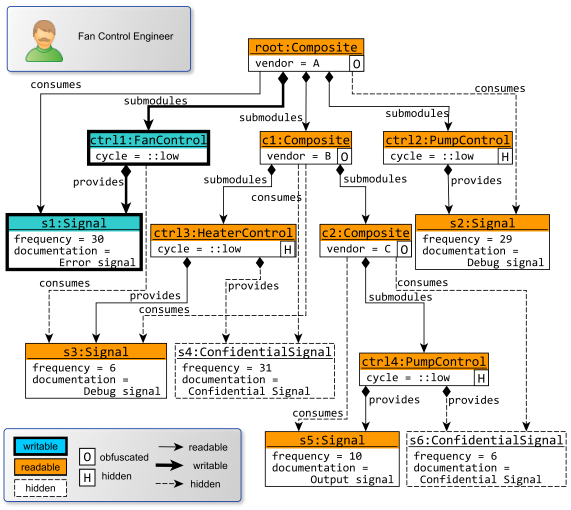
> Example 3. An access control policy is set up to meet the security needs of the running example introduced in Sec. 2.2. A possible permission function ($permission_{Eff}$) is visualized in Fig. 4. For instance, *Pump Control Engineer*s have full access to PumpControl objects and their provided Signals (squares marked with bold borders and blue headers); however they cannot access ConfidentialSignal objects (squares with dashed borders). The rest of the objects are readable, but not writable by this group of users (squares with thick borders and orange headers). If an object is only required to preserve read dependencies, its identifier is obfuscated (marked with "O" letter in a square next to the attribute) and all other attributes remain hidden ("H" letter in the square). Finally, bold edges are writable by the engineers, i.e. the writable signals (`s2`, `s5`) can be removed from their container, or new signals can be created under the writable controls (`ctrl2`, `ctrl4`); thick edges represent readable references (in this example, these are required mostly to preserve containment hierarchy); and the rest of the dashed edges are hidden from the engineers.

4 Bidirectional Model Transformation for Access Control Management

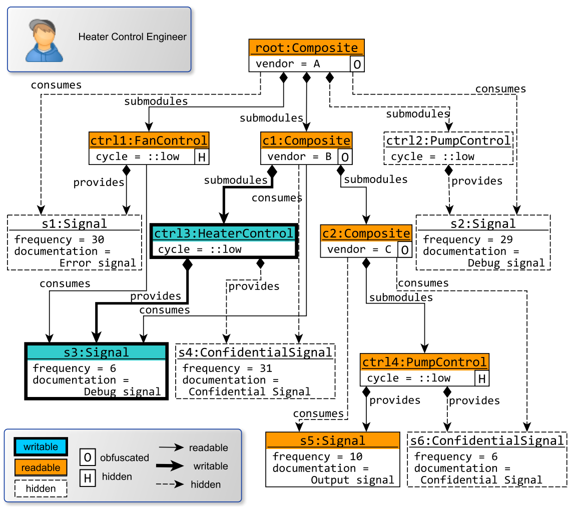
4.1 The Access Control Lens

Due to read access control, some users are not allowed to learn certain model assets. This means that the complete model (which we will refer to as the *gold* model) differs from the view of the complete model that is exposed to a particular user (the *front* model).

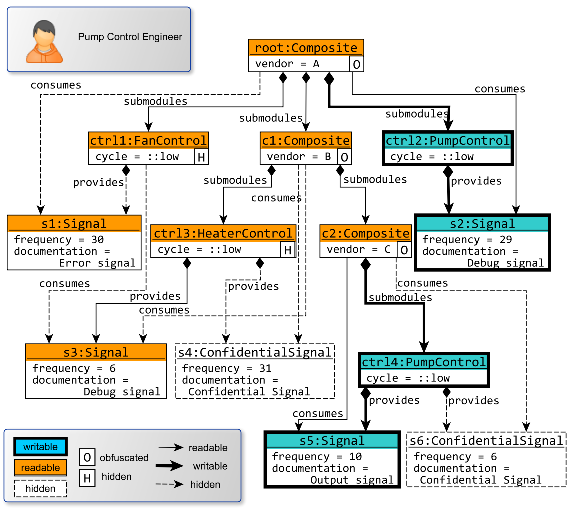
In theory, access control could be implemented without manifesting the front model, by hiding the entire gold model behind a model access layer that is aware of the security policy and enforces access control rules upon each read and write operation performed by the user. However, challenge **C2.1** requires users to access their front models using standard modeling tools; moreover, while challenge **C2.2** requires that in the offline collaboration scenario, they can "take home" their front model files without being directly connected to the gold model. In order to meet these goals, we propose to manifest the front models of users as regular stand-alone models, derived from a corresponding gold model by applying a *bidirectional model transformation.*



(a) Fan Control Engineer



(b) Heater Control Engineer



(c) Pump Control Engineer

Fig. 4: Effective Permissions of the Example

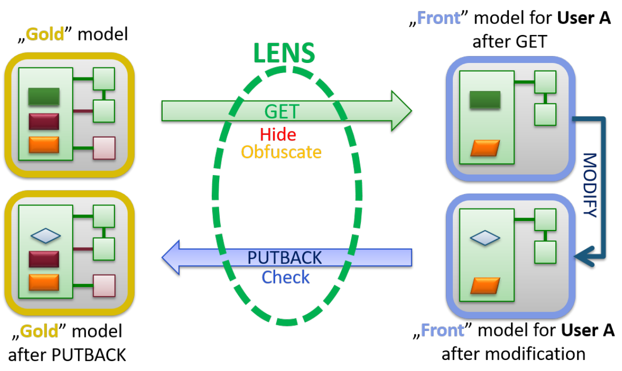

„Gold" model
LENS
„Front" model for User A after GET
GET
Hide
Obfuscate
MODIFY
PUTBACK
Check
„Gold" model after PUTBACK
„Front" model for User A after modification

Fig. 5: Secure Access Control by Bidirectional Lenses

In the literature of bidirectional transformations [23], a *lens* (or view-update) is defined as an asymmetric bidirectional transformation relation where a source knowledge base (KB) completely determines a derived (view) KB, while the latter may not contain all information contained in the former, but it can still be updated directly. The two operations of crucial importance in realizing a lens relationship are the following:

– GET obtains the derived KB from the source KB that completely determines it, and
– PUTBACK updates the source KB, based on the derived view and the previous version of the source (the latter is required as the derived view may not contain all information).

The bidirectional transformation relations between a gold model (containing all assets) and a front model (containing a filtered view) satisfies the definition of a lens. The GET process applies the access control policy for filtering the gold model into the front model. The PUTBACK process takes a front model updated by the user, and transfers the changes back into the gold model.

DEFINITION 8. The GET process derives the front model from the gold model in accordance with the read permissions.

$$\text{GET} :: (M_G, permission_{\text{Eff}}) \rightarrow M_F$$

DEFINITION 9. The PUTBACK process enforces the write permissions and derives the updated gold model from the modified front model and the original version of the gold model.

$$\text{PUTBACK} :: (M'_F, M_G, permission_{\text{Eff}}) \rightarrow M'_G$$

The lens concept is illustrated by Fig. 5. Initially, the GET operation is carried out to obtain the front model for a given user from the gold model. Due to the read access control rules, some objects in the model may be hidden (along with their connections to other objects); additionally, some connections between otherwise readable objects may be hidden as well; finally, some attribute values of readable objects may be omitted, obfuscated, or hidden altogether. If the user subsequently updates the front model, the PutBack operation checks whether these modifications were allowed by the write access control rules. If yes, the changes are propagated back to the gold model, keeping those model elements that were hidden from the user intact (preserved from the previous version of the gold model).

Write access control checks are performed by the PutBack operation as they (a) may prevent a user from writing to the model, and (b) access control rules needs to be evaluated on the gold model.

Access control rules cannot be evaluated directly on the front model since only the gold model contains all information. Thus write access control can only be enforced by taking into account the gold model as well. Therefore, write access control must be combined with the lens transformation. In particular, PutBack must check write permissions; and fail (by rolling back any effects of the commit or operation) if a certain modification cannot be applied to the gold model.

Example 4. In our running example, the original model (Fig. 3) acts as the gold model containing all the information. The Get transformation applies the permissions and produces a front model for each specialist. In Fig. 4, each front model consists of

- the objects with bold or solid borders;
- the references with solid lines;

but the objects and references with dashed borders and lines are removed. Whereas, the attributes marked with

- an "O" in a square are obfuscated;
- an "H" in a square are removed.

When a *PumpControlEngineer* tries to modify the frequency of the signal s3 from 6 to 10, the PutBack operation is responsible for declining this change as the access control rules deny the modification (the signal s3 is readable but not writable).

On the other hand, if a *PumpControlEngineer* tries to modify the frequency of the signal s2 from 29 to 17 the PutBack operation propagates the change back to the gold model (the signal s2 is writable) by identifying the signal s2 in the gold model and setting its frequency attribute.

s3 is replaced with s2 in accordance with the example

4.2 Transformation Design

Both Get and PutBack are designed as rule-based model transformations [18]. In the terminology of model transformation, gold and front models act as the *source* and *target* models, respectively.

To address challenge **C3.1**, the transformations need to be *reactive* and *incremental* computations in the online collaboration scenario.

Reactive transformations [10] follow an *event-driven behavior* where the *events* are triggered by *model manipulations* such as creation/modification/deletion of model assets. The transformation observes these events and reacts to them.

Incrementality [18] means that there is no need to re-execute the whole transformation upon a small change introduced into the model. *Source incrementality* is the property of a transformation that only re-evaluates the modified parts of the source model. *Target incrementality*, means that only the necessary parts of the target model are modified by the transformation, there is no need to recreate the new target model from scratch.

Definition 10. A transformation rule *rule* is associated with a *precondition* $\in$ *Preconditions*, an *action* $\in$ *Actions* (parametrized by a match of the precondition),and a numerical *priority* $\in \mathbb{P}$ value.

$$\mathit{rule} = (\mathit{precondition}, \mathit{action}, \mathbb{P})$$

Definition 11. A transformation $\mathcal{T}$ consists of a set of transformation rules ($\{\mathit{rule}_1, \mathit{rule}_2 \dots \mathit{rule}_n\}$) that a transformation engine $\mathcal{TE}$ executes to incrementally derive an updated target model M_T' from a source and target model M_S, M_T.

$$\mathcal{T} = \{\mathit{rule}_1, \mathit{rule}_2 \dots \mathit{rule}_n\}$$

$$\mathcal{TE} :: (M_S, \mathcal{T}, M_T) \rightarrow M_T'$$

Transformation execution repeatedly fires the rules as follows:

1. finds all the matches of rule preconditions of all rules (this set of matches is efficiently and incrementally maintained during the transformation),
2. selects a match from the rule with the highest priority,
3. executes the action of the rule along that match;

The loop terminates when there are no more precondition matches.

According to the process GET and PUTBACK of the lens, we define $\mathcal{T}_{\text{GET}}$ and $\mathcal{T}_{\text{PUTBACK}}$ transformations, respectively.

These transformations consist of four *groups of transformation rules* based on its direction (GET, PUTBACK) and whether it adds or removes assets from the model (*additive*, *subtractive*):

In case of GET process:

- *Additive* adds assets to M_F if no corresponding assets are present in M_G
- *Subtractive* removes assets from M_F if no corresponding assets are present in M_G

In case of PUTBACK process:

- *Additive* adds assets to M_G if no corresponding assets are present in M_F
- *Subtractive* removes assets from M_G if no corresponding assets are present in M_F

All four groups consist of one rule for each kind of model asset; in the context of this paper, we distinguish 3 kinds of model assets (see Sec. 3.1); this makes twelve transformation rules altogether, described in the tables of Appendices A and B.

The preconditions require to initialize correspondence between front and gold models. For that purpose, we introduce a *trace* function.

DEFINITION 12. The trace function is responsible for associating two object assets with each other:

$$\mathit{trace} :: (\mathit{ObjectAsset}(o_G, t)) \rightarrow \mathit{ObjectAsset}(o_F, t')$$

We select three example transformation rules listed in Table 1 to describe the key concept of how the access control is managed.

ADDITIVE GET OBJECT RULE ($\mathit{rule}_{\text{ADDITIVE GET OBJECT}}$)

The *additive* rule of $\mathcal{T}_{\text{GET}}$ related to object assets is responsible for propagating object addition from the gold model M_G to the front model M_F. A change is recognized in the precondition which selects pairs of $\mathit{ObjectAsset}(o_G, t)$ and $\mathit{ObjectAsset}(o_F, t)$ as follows: an $\mathit{ObjectAsset}(o_G, t)$ in the gold model that has no corresponding $\mathit{ObjectAsset}(o_F, t)$ in the front model, but it should be readable according to the $\mathit{permission}_{\text{Eff}}$. The action part will create a new $\mathit{ObjectAsset}(o_F, t)$ and establish a correspondence relation between these two objects.

"... which selects pairs of $\mathit{ObjectAsset}(o_G, t)$ and $\mathit{ObjectAsset}(o_F, t)$" is added to explain the number 2 in the superscript

EXAMPLE 5. A *system administrator* who has access to the original gold model (depicted in Fig. 3) adds a new signal object sN_G under the heater control unit *ctrl*3. This change needs to be propagated to the front models as the new signal should be at least readable (also writable for *Heater Control Engineers*).

$\mathcal{T}_{\text{GET}}$ transformation will be executed between M_G and the front model of *Pump Control Engineer* M_F^{pump} (depicted in Fig. 4c). The precondition of the $\mathit{rule}_{\text{ADDITIVE GET OBJECT}}$ selects the $\mathit{ObjectAsset}(sN_G, \mathsf{Signal})$ as it has no corresponding $\mathit{ObjectAsset}(sN_F, \mathsf{Signal})$ in the front model. The action part creates $\mathit{ObjectAsset}(sN_F, \mathsf{Signal})$ and traces it back to $\mathit{ObjectAsset}(sN_G, \mathsf{Signal})$. Exactly the same sequence happens in case of the front model of *Fan Control Engineer* $M\text{fan}_F$ (depicted in Fig. 4a)[a].

[a] $\mathit{rule}_{\text{ADDITIVE GET REFERENCE}}$ takes care of the containment reference between sN_G and *ctrl*3.

ADDITIVE GET ATTRIBUTE RULE ($\mathit{rule}_{\text{ADDITIVE GET ATTRIBUTE}}$)

The *additive* rule of $\mathcal{T}_{\text{GET}}$ related to attribute assets is responsible for propagating data value insertion on the gold model M_G to the front model M_F. The precondition of the $\mathit{rule}_{\text{ADDITIVE GET ATTRIBUTE}}$ selects $\mathit{AttributeAsset}(o_G, \mathit{attr}, v)$ in the gold model that has no corresponding $\mathit{AttributeAsset}(o_F, \mathit{attr}, v')$ in the front model, but it should be readable according to the $\mathit{permission}_{\text{Eff}}$. The value of v' is calculated in accordance with its read permission (potentially in an obfuscated form).

EXAMPLE 6. The *system administrator* modifies the frequency attribute of $s1_G$ from 30 to 15. This change needs to be propagated to the front models.

1) $\mathcal{T}_{\text{GET}}$ will be executed between M_G and the front model of *Pump Control Engineer* M_F^{pump} (depicted in Fig. 4c). The precondition of the $\mathit{rule}_{\text{ADDITIVE GET ATTRIBUTE}}$ selects the $s1_F$ object from the front model attribute *frequency* and value 15 as $\mathit{AttributeAsset}(s1_F, \mathit{frequency}, 15)$ does not exist, but it should be readable in F_{pump}. The action part adds the $\mathit{AttributeAsset}(s1_F, \mathit{frequency}, 15)$ to M_F^{pump}.[a]

2) $\mathcal{T}_{\text{GET}}$ will be executed between M_G and the front model of *Fan Control Engineer* M_F^{fan} (de-

rule	Additive GET Object	**Priority**	4
Precondition :: $(M_G(OA_G, RA_G, AA_G), M_F(OA_F, RA_F, AA_F), permission_{\mathrm{Eff}}) \rightarrow \{ObjectAsset\}^2$			
$\{ObjectAsset(o_G, t), ObjectAsset(o_F, t') \mid ObjectAsset(o_G, t) \in OA_G, permission_{\mathrm{Eff}}(ObjectAsset(o_G, t), \mathrm{read}) \neq \mathrm{deny},$ $\nexists ObjectAsset(o_F, t') \in OA_F : trace(ObjectAsset(o_G, t)) = ObjectAsset(o_F, t'), t = t'\}$			
Action			
$OA_F := OA_F \cup \{ObjectAsset(o_F, t')\}, trace(ObjectAsset(o_G, t)) := ObjectAsset(o_F, t')$			

rule	Additive GET Attribute	**Priority**	5
Precondition :: $(M_G(OA_G, RA_G, AA_G), M_F(OA_F, RA_F, AA_F), permission_{\mathrm{Eff}}) \rightarrow \{AttributeAsset\}$			
$\{AttributeAsset(o_F, attr, v') \mid AttributeAsset(o_G, attr, v) \in AA_G, permission_{\mathrm{Eff}}(AttributeAsset(o_G, attr, v), read) \neq deny,$ $\exists ObjectAsset(o_F, t) : ObjectAsset(o_F, t) \in AA_F, trace(ObjectAsset(o_G, t)) = ObjectAsset(o_F, t), \nexists AttributeAsset(o_F, attr, v') :$ $v' = \begin{cases} v, permission_{\mathrm{Eff}}(AttributeAsset(o_G, attr, v), read) = allow \\ obf(v), permission_{\mathrm{Eff}}(AttributeAsset(o_G, attr, v), read) = obfuscate \end{cases}, AttributeAsset(o_F, attr, v') \in AA_F\}$			
Action			
$AA_F := AA_F \cup \{AttributeAsset(o_F, attr, v')\}$			

rule	Subtractive PUTBACK Object	**Priority**	3
Precondition :: $(M_G(OA_G, RA_G, AA_G), M_F(OA_F, RA_F, AA_F), permission_{\mathrm{Eff}}) \rightarrow \{ObjectAsset\}$			
$\{ObjectAsset(o_G, t) \mid ObjectAsset(o_G, t) \in OA_G, permission_{\mathrm{Eff}}(ObjectAsset(o_G, t), \mathrm{read}) \neq \mathrm{deny},$ $\nexists ObjectAsset(o_F, t') : trace(ObjectAsset(o_G, t)) = ObjectAsset(o_F, t'), t = t'\}$			
Action			
If $permission_{\mathrm{Eff}}(ObjectAsset(o_G, type), write) \neq deny$ then $OA_G := OA_G \setminus ObjectAsset(o_G, t), trace \setminus ObjectAsset(o_G, t) \mid trace(ObjectAsset(o_G, t)) = ObjectAsset(o_F, t)$ else ⤩			

Table 1: *Additive* GET and *Subtractive* PUTBACK rules

> picted in Fig. 4a). But now, the precondition has no match as $s1$ is not readable in M_F^{fan}
>
> [a] Similarly, *rule*SUBTRACTIVE GET ATTRIBUTE will handle the removal of the previous attribute asset $AttributeAsset(s1_F, frequency, 30)$ before the addition.

SUBTRACTIVE PUTBACK OBJECT ($rule_{\text{SUBTRACTIVE PUTBACK OBJECT}}$)
The *subtractive* rule of $\mathcal{T}_{\text{PUTBACK}}$ related to object assets is responsible for propagating object asset removals from the front model M_F to the gold model M_G. A deletion is recognized in the precondition as follows: there is an $ObjectAsset(o_G, t)$ in the gold model that has no corresponding $ObjectAsset(o_F, t)$ in the front model. The action part checks the write permissions of $ObjectAsset(o_G, t)$. If the removal of the asset is denied, $\mathcal{T}_{\text{PUTBACK}}$ terminates after a rollback. Otherwise, it removes the selected object asset.

> EXAMPLE 7. A *Pump Control Engineer* removes $ctrl1_F$ object from his front model M_F^{pump} (depicted in Fig. 4c). This change needs to be propagated be to the gold model, thus PUTBACK transformation will be executed between M_F^{pump} and the gold model M_G (depicted in Fig. 4c). The precondition of the rule selects $ObjectAsset(ctrl1_G, FanControl)$. In the action part, the rule realizes that the permissions do not allow to delete $ctrl1$ object, thus the transformation terminates and rejects the change.

To sum up, GET is responsible for enforcing read permissions in front models, while PUTBACK takes care of write permissions. If any write permission is violated, the transformation terminates and the front model (*target*) is reverted to its original state.

4.3 Discussion and Analysis

At the request of reviewers, we have added this section with almost entirely new content to discuss properties of the bidirectional transformation, along with Appendix C to contain the proofs of these properties.

In the following paragraphs, we analyze and discuss properties of the lens transformations.

First, in Sec. 4.3.1, we state the properties that the lens transformations are expected to exhibit. In Sec. 4.3.2, we state and discuss an important assumption that will be vital to proving the aforementioned properties in Appendix C. Finally, in Sec. 4.3.3 we will turn our attention to deviations of the technical realization from the ideal formulation.

4.3.1 Desirable Properties of the Transformation

In the following, we present a number of properties that the lens transformations would be desirable to exhibit.

We first state these desirable properties, and then discuss them individually in subsequent sections to find which ones are met under which conditions by the presented transformations.

Transformation Property 1 (Termination) *Given a pair of starting models,* GET *and* PUTBACK *shall both terminate after a finite number of rule executions.*

Transformation Property 2 (Confluence) *Given a pair of starting models and running the transformation to completion, the terminal state of both* GET *and* PUTBACK *shall be independent from the chosen execution order of rule application, i.e. both* GET *and* PUTBACK *define a deterministic function.*

Transformation Property 3 (Confidentiality) GET *shall yield a front model that contains exactly those assets that are visible according to effective read permissions.*

Transformation Property 4 (Integrity) PUTBACK *shall successfully accept a modified front model if and only if its differences from the original front model do not violate effective write permissions.*

Transformation Property 5 (GetPut) PUTBACK *shall be a no-op when applied on the front model directly returned by* GET*, i.e. if the user makes no changes, the gold model shall not be updated.*

Transformation Property 6 (PutGet) GET *shall be a no-op when applied on the gold model previously updated by a successful* PUTBACK *(from the same front), i.e. if the gold model has not changed, the front model shall not be updated.*

Transformation Property 7 (PutPut) *A user applying a sequence of successful* PUTBACK *operations (and changing the front model inbetween) should have the same ultimate effect on the gold model as applying only the last one.*

The first few properties (Prop. 1, Prop. 2) are generally expected of most rule-based model transformations, in order to define an actual deterministic transformation function. Then Prop. 3 and Prop. 4 state the security-specific requirements.

Next, Prop. 5 and Prop. 6 pertain specifically to bidirectional transformations, and are widely promoted (see e.g. [23,47], also [27] specifically for security views) as very important "well-behavedness" properties that users of bidirectional transformations would most certainly expect. They enable the lens transformations to truly realize an updateable view.

Finally, Prop. 7 provides even stronger predictability guarantees, but is often considered very restrictive and therefore optional in the literature. A benefit of this law is that user modifications are *undoable*, i.e. the original state of the system (incl. gold model) can be restored when a change to the from model is reverted. On the other hand, it might unfortunately disallow certain sensible extensions, an example for which we include below.

As introduced in [12], a possible sample refinement of the write permission levels could be $\{deny < dangle < allow\}$. Cross-references with write permission level *dangle* can not be normally modified by the user, but they can be removed as the side effect of deleting the source or target object of the reference (if that deletion is permitted). Unlike the usual allowed write permission, *dangle* does not imply the readability of the asset, so this kind of deletion is possible even if the cross-link is not visible to the user. Imagine a traceability link that points from a hidden part of the model to a visible object; the difference between assigning *deny* or *dangle* is that the target object can not be deleted by the user in the former case, while its deletion would be allowed (with an invisible side-effect of removing the traceability link) in the latter case. These *dangle* semantics can be similarly extended to attributes or contained objects that might be attached as (invisible or read-only) tags to objects; they must not be modified by the user, but will be removed along with the object they are attached to if the object is deleted. It is easy to see that (a) such a feature would be quite useful in many practical applications of the approach presented in the paper, yet (b) Prop. 7 will not hold, as undoability is lost when dangling links/attributes/objects are removed.

4.3.2 Regularity of Policy

Constant complement [8], a common strategy for proving desirable properties of secure views, involves partitioning the data into a readable part and the so-called *complement.* This partitioning can be used e.g. to verify whether PUTBACK (translator in the terminology of [8]) may possibly change the complement (which would violate Prop. 7). Similarly, correctness proofs can benefit from applying a second kind of partitioning [27] into a writable and endorsed part of the model.

Unfortunately, these partitioning schemes do not apply perfectly to our approach for the simple reason that a single asset might move from one partition to another as the model evolves - in fact even during the execution of PUTBACK. This is due to the fact that our approach is more powerful: we do not consider explicit access control attributes of assets, but rather derive effective permissions from policy rules based on arbitrary model queries (over the gold model) that can take into account the wider context of assets; thus it is possible to change the effective permissions for a given asset (even without directly changing the asset itself). Therefore we first make the following assumption, and then discuss violating cases separately:

Assumption 1 (Regularity) *For any transformation run, there exists a constant permission set for all assets (i.e. a constant partitioning of assets based on permission levels) so that effective permissions will always evaluate to results consistent with this fixed permission set when they are evaluated during the run (i) as a condition excluding activations of higher-priority rules, or (ii) as a precondition to an individual rule to be executed, when higher-priority rules have already been found to have no matches, (iii) as a condition for rejecting disallowed write attempts, or (iv) to determine termination.*

Note that Asm. 1 does not require that the actual effective permissions (as evaluated following the policy by the appropriate algorithms [12, 20]) remain entirely constant, that would not be feasible. For example, if a user creates a new model element, the corresponding asset propagated to the gold model by PutBack would only evaluate as writeable once it exists in the first place. What is actually required is that permissions are not allowed to flip-flop; i.e. a transformation should never observe a particular asset change its effective permissions if the transformation has already acted upon the old value of the effective permissions. This condition is met in the previous example, as we can include the newly created asset in the constant permission set as writeable: the asset did not exists in the gold model before its creation, so no rules would have ever observed it as an existing but non-writeable asset; as far as the rules are concerned, the asset could have always been listed as writeable in the permission set.

Get leaves the gold model and thus the effective permissions unchanged, so Asm. 1 holds trivially. For PutBack, however, it is possible to come up with scenarios where Asm. 1 is violated. One of these cases is *privilege escalation*, where a user can make a change somewhere in the model that would grant them additional read or write privileges somewhere else that they did not previously have (even though those assets existed before). The other case is *lockout*, where a user can make a change that will have the side-effect of losing their read or write access on some assets (even though those assets continue to exist). We believe both of these cases are likely symptoms of defective policy definition, and a system can only be considered secure and reliable if it does not exhibit these behaviours (or only in a very controlled manner).

Therefore, in the proofs for the properties of Sec. 4.3.1, we consider Asm. 1 to hold, and apply partitioning-based arguments partly similar to those in [8, 27]. This limitation to the case with regularity is, on one hand, necessary to prove the properties stated earlier (the exception is Prop. 1; the transformations will be shown to terminate regardless whether regularity holds). On the other hand, the limitation is prudent for the above listed reasons of security. Finally, the limitation is also feasible, as it is fairly easy to screen changes during PutBack and reject them if they lead to either privilege escalation or lockout. Static analysis of policy definitions regarding their susceptibility to these problems is left as future work.

Now we are ready to sketch conditional proofs for each of the listed properties to hold for the transformations induced by rule sets in Appendix A and Appendix B; the proof sketches are found in Appendix C.

4.3.3 Realization in EMF

We realized the presented lens transformation in the Eclipse Modeling Platform (EMF) [52]. Instead of approaches specifically designed for easy specification of bidirectional transformations, the unidirectional and reactive VIATRA framework [54] has been chosen for its (a) target-incremental transformations and (b) source-incremental model queries to define rule preconditions.

The presented transformation rules, as well as the proof sketches, are formulated on technology-independent models defined as a set of model assets. Actual model representations - EMF in our case - expose an object-oriented API instead. Therefore, we have implemented a *relational model wrapper* layer that exposes the contents of the model through a writeable API as a set of model assets (essentially tuples). This abstraction, however, is incomplete: the underlying object-oriented model structure (i) may not be compatible with all set operations, and (ii) may allow a given set of operations only in certain sequences. The transformation must enforce these constraints.

The first problem occurs if adding or removing a model asset would violate the internal consistency (see Sec. 3.1) of the model; this is avoided by the consistency property of effective permission function in such a way that both Get and PutBack would only attempt valid changes to the front and gold models, respectively.

The second problem occurs if valid changes are attempted in the wrong order, e.g. if a reference asset is only deleted after deleting the object asset for one of the endpoints. By choosing the transformation rule priorities accordingly, the object, attribute and reference rules are ordered in a way that avoids violating these kinds of constraints in all but one cases. The remaining case is object containment, e.g. a child object cannot be added to the model before its container object is created. This depends on the ordering of two instances of the object rule, and thus cannot simply be expressed using rule-level fixed priorities. To solve this problem, the relational model wrapper temporarily allows "unrooted" model objects detached from the model. Note that the EMF API itself allows the existence of such detached model objects, they are just not treated as part of the model (resource set) by default, which our model wrapper needs to circumvent.

As a further effect of this abstraction layer, there is a genuine loss of information: the ordering of multi-valued collections is not preserved in the relational representation. See Sec. 6.3.2 for discussion.

As a slight technical hurdle, the EMF-based query engine of VIATRA, in charge of interpreting the query-based security policy, is not actually capable of evaluating permission queries on assets that are non-existent in the gold model. A workaround is applied in practice to the additive PUTBACK attribute rule (the only rule where this is relevant, due to obfuscation), which we omit here.

Finally, we note that in order to simplify the language of the discussion, we have informally described assets as potentially being contained in both the gold and front models. Since a single EMF object is contained in at most one model, it would be more precise to say that the two models contain disjoint assets, that are related by the equivalence induced by the *trace* function (see Sec. 4.2).

5 Collaboration Scheme

To satisfy our goal **G4**, a general collaboration scheme is required including the bidirectional lens transformation between a server and several clients to enforce access control policies correctly.

The server stores the gold models and clients can download their specific front models. Modifications, executed by a client, can be submitted to the server and downloaded by the other clients. These are the basic actions that nowadays, a version control system (VCS) should provide to a user. In case of various implementations, these actions may be called differently (e.g. *checkout*, *update*, *commit* in SVN or *clone*, *pull*, *push* in Git).

According to the basic actions supported by any VCS, we define the basic operations of the collaboration scheme as follows:

Checkout downloads the model from the server-side to the workspace of a specific client who initiated the operation.

Update retrieves the model changes from the server-side to the workspace of a specific client who initiated the operation.

Commit propagates the changes of a specific client to the server-side.

5.1 Formalization of the Collaboration

Fig. 6 and Fig. 7 describes the behavior of collaboration scheme as *state machines* for the server and the client.

A state machine consists of *states* (represented by boxes) and *transitions* (denoted by directed edges) between states. Each state-machine has an *initial state* (denoted by arrow from a black circle) and a *current state* that specifies the system at a certain time.

The system can accept *input events* and send *output events* during its process (denoted by labels on the edges where "?" and "!" mean receiving and sending a certain event, respectively, following process-algebraic notation). In the concept of collaboration, each event is assigned to a collaborator using *"."* symbol after the name of event e.g. *input.x/output.y* means, that the collaborator x initiates an *input* and the transition produces an *output* to the collaborator y. A transition will be executed immediately when its input event arrives and during the execution it produces its output event.

Compound state (visualized as boxes containing other states) refines the behavior of a given state by defining its own state-machine where only one state can be active.

Orthogonal regions (divided by dashed borders) separate the behavior of independent states and they are processed concurrently. In each region, only one state can be active at a time.

Two state-machines can *synchronize* on events sending by one and received by the other one.

***Server** (Fig. 6).* Its state machine has three orthogonal regions to handle the *commit*,*update* and *checkout* requests concurrently.

Checkout and Update. In case of receiving *checkout* and *update* requests, our approach rejects them when a user has no access to the model itself[1] by sending an *accessDenied* event followed by a *failure* event. Otherwise, a *success* event is sent.

[1] Note that we make a distinction between a user having no access to a model at all, and a user having access to the model, but nothing is readable in it.

Upon an *update* request, it is also checked, whether the client's model is up-to-date and then an *upToDate* event is produced followed by a *success* event.

Commit. The process of receiving *commit* requests consists of Idle, Locked, Synchronization and Unlock hierarchical states:

Idle state accepts commit requests from any collaborator. It produces an *accessDenied* event when the user has no access to the model; or a *needToUpdate* event when the user needs to update his/her model locally to be able to commit the modifications. Both events are followed by a *failure* event. Otherwise, the system locks the model to prevent concurrent processing any other commit requests and activates the Locked state.

Locked state executes the $\mathcal{T}_{\text{PutBack}}$ in the name of the commit owner (x) and rejects the commit requests from any collaborator (y) by sending an *otherCommitUnderExecution* event with a *failure* event. After the execution of the transformation, a *policyViolated* event is sent to x, if the changes violated the access control policy and the system steps to *Unlock* state. Otherwise, a *putback* event leads the system to the Synchronization state.

Synchronization state is responsible for sending the *success* event to the owner of the commit and executing $\mathcal{T}_{\text{Get}}$ to propagate the changes to other collaborators (denoted by output event *get* for all collaborator except the owner of the commit in the state *Sync*). Then systems moves forward to the *Unlock* state.

Unlock state is responsible for unlocking the model in all cases (*unlock* event). If the system is led to this state after a policy violation, the state produces a *failure* event before the unlock. It also rejects any other commit request by sending an *otherCommitUnderExecution* event with a *failure* event.

Client (Fig. 7). Its state machine cooperates with the server using the sending and receiving events that (i) trigger an operation (*commit*,*update*,*checkout*); (ii) indicate failures (*needToUpdate*); and (iii) indicate server responses (*success*, *failure*). It consists of Checkout, Idle and Update hierarchical states:

Checkout State. First, the clients need to checkout their models represented by sending a *checkout* event in the Checkout state. Based on the received server response, the clients can move to Idle state.

Idle State. In Idle state, clients can commit or update their changes by sending *commit* or *update* events. All the events produced by the server can be received, but only the *needToUpdate* event restricts the behavior of the client by moving to Update state.

Update State. The clients need to initiate an update request by sending a *update* event to be able to commit their changes again.

5.2 Correctness Criteria

To address the challenge **C4.1** we describe the *correctness criteria* that the collaboration scheme needs to satisfy:

Criterion 1. The scheme needs to be deadlock free (i.e. all the locks need to be unlocked during a commit operation).

Criterion 2. The scheme needs to be livelock free (i.e. all the operations need to finish at some point and lead the scheme to an idle state).

Criterion 3. Commit operation shall be rejected while another commit is under execution.

Criterion 4. Commit operation shall propagate the changes to all collaborators.

Criterion 5. Clients need to initiate an update operation when it is required by the server.

Note that Criterion 1. and Criterion 2. are required to ensure that the collaboration can run without any manual intervention. Criterion 3. declines overwriting changes without notification of a commit happened previously. Criterion 5. enforces the clients to avoid conflicting commits.

5.3 Proof of Correctness

In accordance with challenge **4.1**, we formalized our collaboration scheme as *communicating sequential processes (CSP)* [45, 46] described in the appendix D to prove its correctness. *CSP* is a formal specification language of concurrent programs or systems where the communications and interactions are presented in an algebraic style.

The **Server** and **Clients** processes define the behavior of exactly 1 server and n clients, respectively. The collaboration is specified as a concurrent execution (denoted by $||$) of the server and clients where the processes synchronize on a given set of events *SyncEvents*: {*commit, update, checkout, accessDenied, policyViolated, needToUpdate, failure, success*}.

$$\textbf{Collaboration} = \textbf{Server}||_{SyncEvents}\textbf{Clients}[1..n]$$

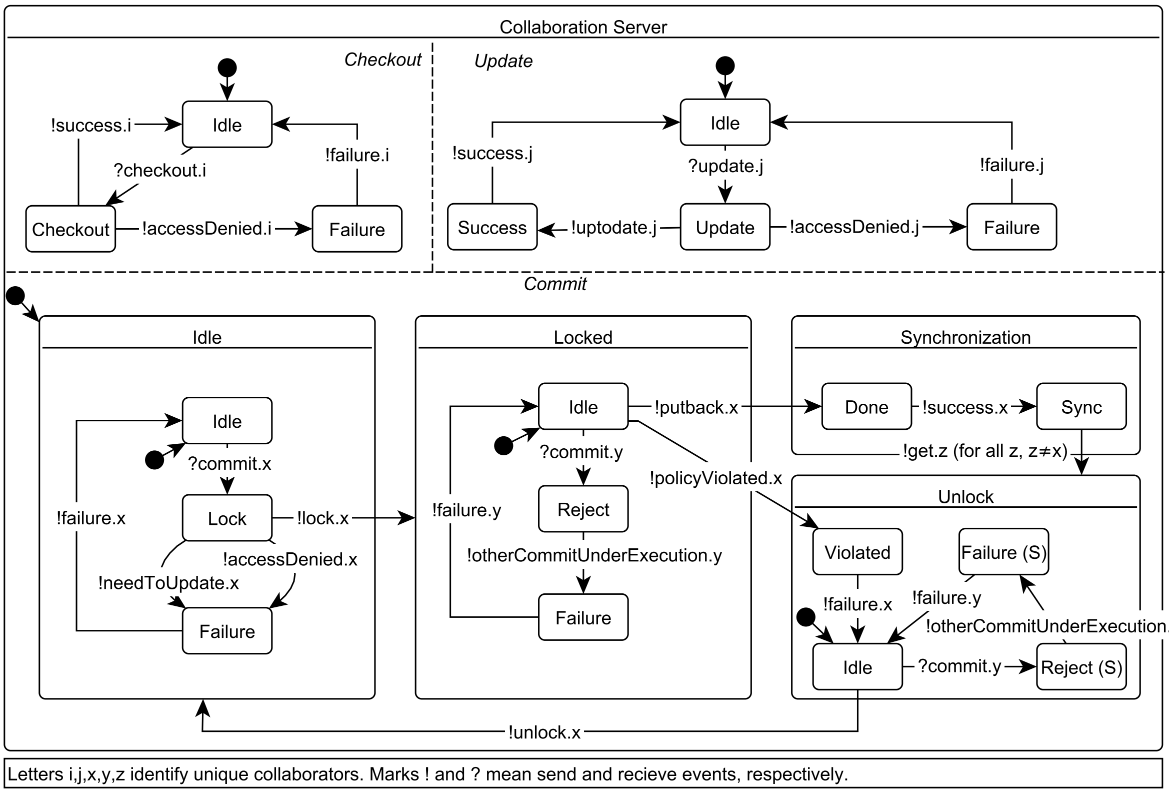


Fig. 6: State-machine of the Collaboration Server

For the analysis, we used the *FDR4 tool* [32] to evaluate *assertions* over certain properties of the processes.

Criteria 1. and 2. requires the entire collaboration process (**Collaboration**) to be deadlock and livelock free. To check these properties, the : [*deadlock free*] and : [*divergence free*] built-in structures are used, respectively.

$$\textit{assert}\ \textbf{Collaboration}\ :[\textit{deadlock free}] \qquad (1)$$
$$\textit{assert}\ \textbf{Collaboration}\ :[\textit{divergence free}] \qquad (2)$$

The rest of the criteria requires to evaluate whether the process formally refines a certain event sequence according to the CSP models, namely the *traces* and *failures* models. For that purpose, we use the $\mathbb{T}:$ and $\mathbb{F}:$ structures, respectively, where

P $\mathbb{T}: < a, b, c >$ means that process **P** must be able to perform the ordered sequence of the events a, b, c and only these events.

P $\mathbb{F}: < a, b, c >$ means that process **P** must not be able to refuse to perform the ordered sequence of events a, b, c without performing any other event.

To check the remaining criteria, we introduce the $\neg$ symbol to negate assertions; the $\backslash$ symbol that hides events from the process and $\mathcal{E}$ denotes the events provided by all processes. The combination of these symbols allows us to evaluate the processes in the context of certain event, e.g. **P** $\backslash(\mathcal{E} \cap \{a, b, c\})$ means that all events are hidden from the process **P** except a, b and c.

Criterion 3. includes that after executing a $\mathcal{T}_{\text{PutBack}}$, another $\mathcal{T}'_{\text{PutBack}}$ cannot be executed without unlocking the model.

$$\begin{aligned}&\textit{assert}\ \textbf{Server}\ \backslash(\mathcal{E} \cap \{unlock, putback\}) && (3)\\ &\neg\mathbb{T}:\ < putback.x, putback.y >\\ &x, y \in Int, x \neq y\end{aligned}$$

Criterion 4. requires to execute $\mathcal{T}_{\text{Get}}$ for all collaborators other than the owner of the commit after a successfully executed $\mathcal{T}_{\text{PutBack}}$ but before unlocking the model. As we start the synchronization with collaborator 1, and then 2, it implies that the collaboration scheme needs to execute it to the last collaborator, namely n.

$$\begin{aligned}&\textit{assert}\ \textbf{Server}\ \backslash(\mathcal{E} \cap \{get.n, putback, unlock\}) && (4)\\ &\mathbb{T}:\ < putback.x, get.n, unlock.x >\\ &x, N \in Int, x \neq n\end{aligned}$$

Table 2: Results of the assertion in *FDR4 tool*

		States	Transitions	time (s)
Assertions	1	89233	227591	0.44
	2	89233	227591	0.51
	3	452	1121	0.13
	4	417	1071	0.42
	5	10	11	0.12
	6	10	11	0.12

To satisfy CRITERION 5., after a commit operation rejected by the server with a *need to update* message, the client (i) cannot commit again and (ii) must be able to initiate update operation:

$$\textit{assert}\ \textbf{Clients} \setminus (\mathcal{E} \cap \{commit, update, needToUpdate\}) \quad (5)$$
$$\neg\mathbb{T}:\ < commit.x, needToUpdate.x, commit.x >$$
$$\textit{assert}\ \textbf{Clients} \setminus (\mathcal{E} \cap \{commit, update, needToUpdate\}) \quad (6)$$
$$\mathbb{F}:\ < commit.x, needToUpdate.x, update.x >$$
$$x \in Int$$

As in the assertions we hide several events, the *FDR4 tool* was able reduce the *state space* and the transitions that needs to be traversed.

We evaluated the assertions[2] for $n = 5$ users. The results are presented in Table 2. To check deadlock and livelock properties, all the events and states are required. Hence the tool traversed almost 90000 states and 230000 transitions to prove these properties. To verify the rest of the assertions, at most 500 states and 1100 transitions were enough to traverse. All the assertions are evaluated within less than 0.51 seconds and none of them failed.

According to the results, we state that our collaboration scheme for access control management satisfies the correctness criteria.

6 Realization of Collaboration Scheme[3]

In accordance with challenge **4.2**, our goal is to provide tool support for enforcing fine-grained model access control rules in *offline* and *online* scenario realizing the introduced *collaboration scheme* (see Sec. 5).

6.1 Offline Collaboration

In the offline scenario, models are serialized (e.g. in an XMI format) and stored in a Version Control System (VCS). Users work on local working copies of the models in long transactions called commits. The goal of our approach is to manage fine-grained access control on the top of existing security layers available in the VCS.

6.1.1 Realization

The concept of *gold* and *front* models is extended to the repository level where the two types of repositories are called *gold* and *front* repositories as depicted in Fig. 8.

"The concept of *gold* and *front* models is extended to the repository level ..." is added to introduce the difference between the model and repository level concepts

The *gold* repository contains complete information about the gold models, but it is not accessible to collaborators. Each user has a *front* repository, containing a full version history of front models. New model versions are first added to the front repository; then changes introduced in these revisions will be interleaved into the gold models using PUTBACK transformation. Finally, the new gold revision will be propagated to the front repositories of other users using GET transformation. As a result, each collaborator continues to work with a dedicated VCS as before, thus they are unaware that this front repository may contain filtered and obfuscated information only.

Existing access control mechanisms (such as firewalls) are used to ensure that the gold model is accessible to superusers only, and each regular user can only access their own front repository. These regular users can use any compatible VCS client to communicate with their front repository, being unaware of collaboration mechanisms in the background.

This scheme enforces the access control rules even if users access their personal front repositories using standard VCS clients and off-the-shelf modeling tools. Nevertheless, optional client-side collaboration tools may still be used to improve user experience, e.g. for smart model merging [21], user-friendly lock management [15], or preemptive warning about potential write access violations that greatly enhances the usability and applicability of the offline scenario.

6.1.2 Realization of the Collaboration Scheme

In the current prototype, our collaboration scheme is realized by extending an off-the-shelf VCS server, namely *Subversion* [5]. Subversion provides features of the collaboration scheme by default:

FILE-LEVEL ACCESS CONTROL
is responsible for sending *accessDenied* event to the

[2] The complete formal specification is available at: `goo.gl/pJzIX1`

[3] Source codes and more details are at `https://tinyurl.com/sosym-access-control-source`

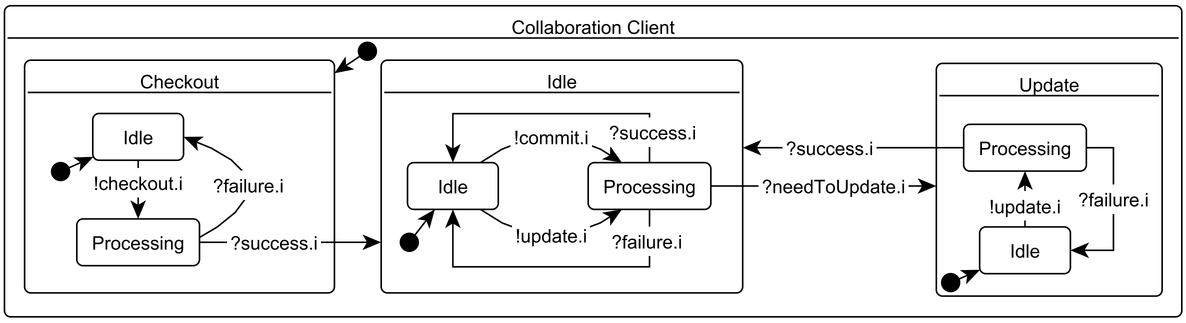


Fig. 7: State-machine of the Collaboration Client

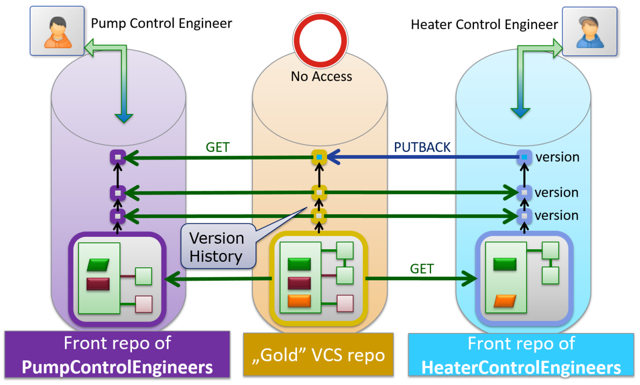


Fig. 8: The MONDO Offline Collaboration Server: Architecture

collaborators whenever their access is denied for a certain file (which contains models).

Version-control
allows to the users to download the files by sending a *checkout* event, submit their changes by sending a *commit* event and update the files by sending an *update* event.

Version check
checks the version of the files and sends *upToDate* the collaborators whether the files are already up-to-date upon an update or sends *needToUpdate* event they need to update upon a rejected commit.

File-level locking
allows users to lock files by sending a *lock* event and reject commits initiated by other users. They can also remove their locks by sending an *unlock* event.

Handling Multiple Requests
allows users to initiate multiple requests simultaneously that the server can accept.

Final Notification
notifies the users about the result of their requested operations by sending a *success* or a *failure* event.

As checkout and update operations of the collaboration scheme are fully handled by Subversion, we need to integrate the $\mathcal{T}_{\text{PutBack}}$ and $\mathcal{T}_{\text{Get}}$ into the commit operation to enforce fine-grained access control and propagate the changes.

Hooks are programs triggered by repository events such as *lock*, *unlock* or *commit*. The hook may be set up to be triggered before such an event (with the possibility of influencing its outcome, e.g. cancelling it upon failure) or directly afterwards (when the event is guaranteed to have happened).The following hook programs will be executed upon a commit operation.

Pre-Commit Hook. $\mathcal{T}_{\text{PutBack}}$ is invoked by *pre-commit hook* executing when a user attempts to commit a new revision of a model $M_F^{r'}$ (new revision r' of a model M_F). This hook performs the following steps to enforce access control policies corresponding to Fig. 9:

1. Parent revision M_F^r of $M_F^{r'}$ is identified.
2. Revision M_F^r is traced to the corresponding revision M_G^R in the gold repository.
3. The hook attempts to put a file-level *lock* to M_G in the gold repository.
   (a) If the locking attempt fails, the hook terminates sending an *otherCommitUnderExecution* event.
   (b) Otherwise, the lock on M_G is activated by sending a *lock* and the hook continues its process.
4. $\mathcal{T}_{\text{PutBack}}$ is executed between $M_F^{r'q}$ and M_G^R in the gold repository, in order to reflect the changes performed in the new commit.
   (a) If the $\mathcal{T}_{\text{PutBack}}$ detects any attempts to perform model modifications violating write permissions, then the commit process to the front repository terminates by sending a *policyViolated* event.
   (b) Otherwise, the commit is deemed successful, and $M_G^{R"}$ is committed to the gold repository (with metadata such as committer name and commit message copied over from the original front repository commit).

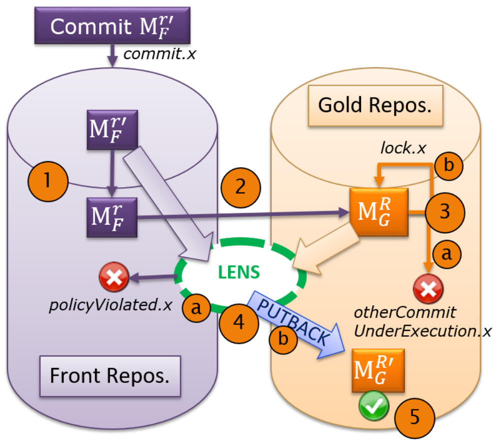


Fig. 9: Pre-commit hook at the front repository

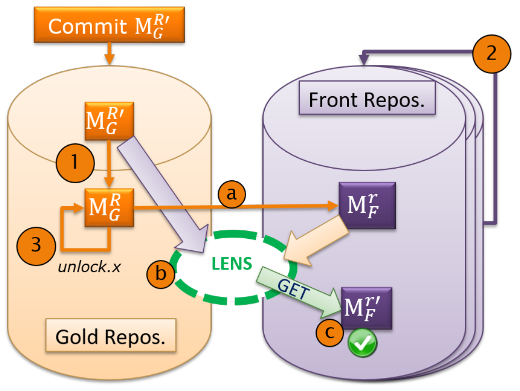


Fig. 10: Post-commit hook at the gold repository

5. Finally, the hook finishes successfully and let the VCS server to handle the request.

Post-Commit Hook. $\mathcal{T}_{\text{GET}}$ is invoked by *post-commit hook* synchronizing all front models M_F^r with the new revision of gold model $M_G^{R'}$. This hook is triggered after a commit of $M_G^{R'}$ finished successfully at the gold repository and performs the following steps correspond to Fig. 10 to propagate the new changes.

1. Parent revision M_G^R of $M_G^{R'}$ is identified.
2. The hook iterates over each front repository and execute the following steps. If the commit to the gold repository is initiated by a front repository then the originating front repository will be skipped.
   (a) Revision M_G^R is traced to the corresponding front revision M_F^r in the front repository.
   (b) $\mathcal{T}_{\text{GET}}$ is executed between $M_G^{R'}$ and M_F^r in order to reflect the changes performed in the commit. If M_F^r does not exist, it is handled as an empty model.
   (c) New revision of the model $M_F^{r'}$ is commited to the front repository (with metadata such as committer name and commit message copied over from the original front repository commit).
3. The hook removes the lock from $M_G^{R'}$ by sending an *unlock* event.
4. Finally, it finishes successfully and lets the VCS server to handle the request.

6.1.3 Discussion

It is worth discussing the following properties of the offline collaboration framework.

Generality. Our solution is general and adaptable to any VCS that supports *checkout*, *update*, *commit* operations (maybe they are named differently).

Server Response. Users get response to their commit right after the collaboration server attempts to propagate back the changes to the gold repository. If any access control rule is violated the pre-commit hook fails. At the last phase of pre-commit, the VCS declines the commit action to the gold repository, if any modified files are locked on the gold repository. Hence, the hook fails again and prevent the VCS specific file level locks. In contrast, if everything goes well the users do not need to wait for synchronizing with the remaining front repositories.

Multiple Models in a Commit. A single commit may update several models at once. In this case, the hooks are invoked for each model in the commit.

Non-blocking Commit. Commit operation does not block update and checkout operations as previous versions still readable in the front repositories.

Models stored among other project files. Our solution supports storing models along with non-model files in the repositories. The hooks can be parameterized with file extensions to determine whether a file needs to be handled as a model. When a file is not a model, it is simply copied from the gold repository to the front repositories.

Correspondence Relation. It is a challenging task to identify the correspondences between model assets of the front and gold models, where the models are stored independently as it is addressed in *C2.2*.

Our approach currently uses specific attributes to provide permanent identifiers. Such a permanent identifier is preserved across model revisions and lens mappings, and can therefore be used to pre-populate the object correspondence relation. In our running example, each object has a unique *id* attribute. Note that unlike EMF, some modeling platforms (e.g. IFC [34]) automatically provide such permanent identifiers.

While requiring permanent identifiers is a limitation of the approach, it is only relevant for modeling platforms that do not themselves provide this kind of traceability, and only in the offline collaboration scenario. Being able to identify model objects is a relatively low barrier for modeling languages; e.g. the original wind turbine language includes a unique identifier for all model objects.

Authorization Files We have taken the design decision that the *authorization files* are stored and versioned in the same VCS as the models. Thus policy files may evolve naturally along with the evolution of the contents of the repository.

Policy files are writable by superusers only, but readable by every user; this means that offline clients may evaluate security rules on their offline copies themselves. Note that we do not believe that this openness of the security policy causes major security concerns, as security by obscurity is not good security principal. In any way, names and parameters of security rules should not themselves contain sensitive design information.

6.2 Online Collaboration

In the online scenario, several users can simultaneously display and edit the same model with short transactions by using a web-based modeling tool where changes are propagated immediately to other users during *collaborative modeling sessions*. In contrast to the offline scenario, where users manipulated local copies of the models, models are kept in a server memory and users access the model directly on the server. The goal of our approach in accordance with *C3.1* is to incrementally enforce fine-grained model access control rules and on-the-fly change propagation between view models of different users.

6.2.1 Technical Realization

During a collaborative modeling session, a model kept in server memory for remote access may also be called a *whiteboard* depicted in Fig. 11. The collaboration server hosts a number of whiteboard sessions, each equipped with a gold model. Each user connected to a whiteboard is presented with their own front model, connected to the gold model via a lens relationship. The front models are initially created using GET. If a user modifies their front model, the changes are propagated to the gold model using PUTBACK, and propagated further to the other front models using GET again. In case of online collaboration, these lens operations are continuously and efficiently executed as a *live transformation* [18], thus users always see an up-to-date view of the model during the editing session.

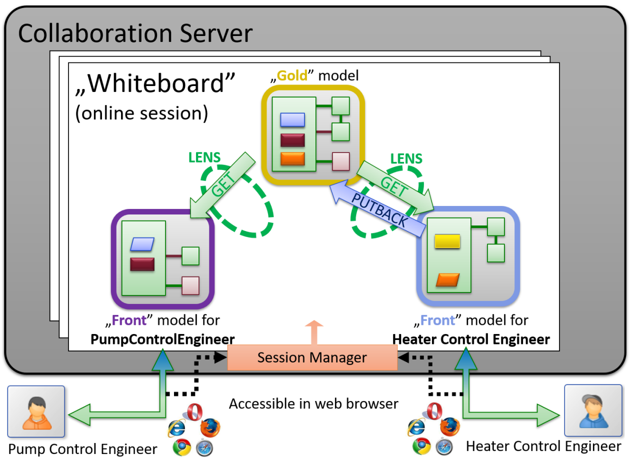


Fig. 11: Overview of Online Collaboration

Similarly to modern collaborative editing tools (such as Google Sheets [17]), whiteboards can be operated transparently: whenever the first user attempts to open a given model, a new whiteboard is started; subsequent users opening the model will join the existing whiteboard. When all users have left, the whiteboard can be disposed. The model may be persisted periodically, or on demand ("save button"). The *session manager* component enables collaborators to start, join or leave whiteboard sessions and persist models to disk.

6.2.2 Realization of the Collaboration Scheme

To achieve challenge **4.2**, we need to discuss how the online collaboration realizes the collaboration scheme.

1. The *checkout* operation is equivalent to joining the whiteboard session for the first time, except it requires to execute $\mathcal{T}_{\textsc{Get}}$ that achieves the front model on which the new user can work.
2. The *update* operation is equivalent to refresh the browser on client side. Via web-based technologies, the collaboration framework notifies and forces the clients' browsers to refresh when new changes are introduced into their front models. However, manual refresh usually results in an *upToDate* event, except when the notification and the manual refresh initiated at the same moment.
3. The *commit* operations are initiated right after users apply modifications on their front models. Other clients need to wait (receiving *otherCommitUnderExecution* event) until the commit finishes, including the execution of PUTBACK and all GET processes to propagate the changes.

After a successful commit, clients receive notification to force them to initiate and update. When a *policyViolated* event occurs, the change is immediately rolled back at the initiator's front model.

6.2.3 Discussion

It is worth discussing the following properties of the online collaboration framework.

Conflicts Handling. As the online collaboration operates with short transactions, it has only a small chance that conflict occurs during the session (e.g. a collaborator modifies an object that is deleted by another collaborator and the propagation of the deletion is under execution). However, if a conflict araises, it is resolved by accepting the remote changes. It also implies that the latter changes will be lost.

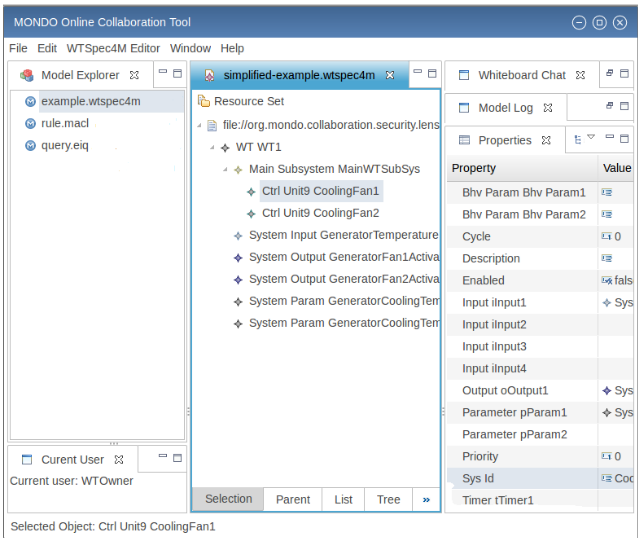
Blocking Checkout and Update. Version numbers are not considered in online collaboration and only one gold model exists during a session. Hence, checkout and update operations need to wait until the commit operation finishes if these operations were initiated during the execution of a commit.

Prototype User Interface. An initial user interface is implemented as a proof-of-concept, depicted in Fig. 12, that uses the editors automatically generated from EMF metamodels[4] which also provides similar modeling environment as the desktop Eclipse IDE. This prototype tool is to demonstrate how to adapt rule-based access control with bidirectional lenses to the online collaboration scenario. We strongly believe that several other existing tools such as GenMyModel [6, 49, 50] can easily adapt our solution.

Correspondence Relation. In the online case, the gold and its front models are initiated for a collaboration session. During a session, these models are stored in the memory and there is no need to reload any of them. Correspondences established during $\mathcal{T}_{\text{GET}}$ can be used through the online collaboration. Hence, there is no limitation about the models (unlike in the offline case, where unique identifiers are required).

Integration with Offline Collaboration. Models and authorization files can be persisted to an underlying gold repository provided by a VCS. The online collaboration tool can access them using checkout/update/commit commands. However, if file-level conflicts occur in the underlying VCS, they will need specific user interfaces to resolve them. Instead, we decided that new whiteboard sessions put file-level locks on the resources related to the models to prevent conflicts in the VCS upon persisting.

[4] EMF and RAP integration: `https://wiki.eclipse.org/RAP/EMF_Integration`


MONDO Online Collaboration Tool
File Edit WTSpec4M Editor Window Help
Model Explorer
example.wtspec4m
rule.macl
query.eiq
simplified-example.wtspec4m
Resource Set
file://org.mondo.collaboration.security.lens
WT WT1
Main Subsystem MainWTSubSys
Ctrl Unit9 CoolingFan1
Ctrl Unit9 CoolingFan2
System Input GeneratorTemperature
System Output GeneratorFan1Activa
System Output GeneratorFan2Activa
System Param GeneratorCoolingTem
System Param GeneratorCoolingTem
Selection
Parent
List
Tree
Whiteboard Chat
Model Log
Properties
Curent User
Current user: WTOwner
Selected Object: Ctrl Unit9 CoolingFan1

Property	Value
Bhv Param Bhv Param1	
Bhv Param Bhv Param2	
Cycle	0
Description	
Enabled	fals
Input iInput1	Sys
Input iInput2	
Input iInput3	
Input iInput4	
Output oOutput1	Sys
Parameter pParam1	Sys
Parameter pParam2	
Priority	0
Sys Id	Coc
Timer tTimer1	

Fig. 12: User Interface for Online Collaboration Prototype

6.3 Assumptions and Limitations

6.3.1 Feedback on Write Access Control

As discussed before, the means of write access control is the following. In the offline case, the server rejects unauthorized modifications only when the user finally submits them. In the online case, PutBack is a live transformation, and it can immediately reject non-compliant changes. (Note that rejected write attempts offer a *side channel* through which some information on the hidden parts of the gold model may be gained. While it is beside the point here, policy designers are advised to take such unintended effects into account.)

It can be frustrating and unproductive for users to learn about their insufficient permissions by trial and error. This is especially true in the offline case, where the feedback only arrives when modifications are actually committed. In a better system, write restrictions would be readily available to the user; advanced modeling tools may even incorporate this information into their model notation, e.g. to visually show read-only parts of the model as frozen.

However, such a tight feedback loop in the offline case would either require nonstandard communication channels (with their own security risks) to disclose the evaluated permission sets with the client; or alternatively, additional computations such as client-side approximation of the policy queries based on the incomplete information in the front model. Proposing a satisfactory solution is left as future work, e.g. by elaborating initial ideas of [15].

Motivated by the review, we have rephrased this entire subsection to better emphasise what the actual claims are, what limitations we see, and how we hope to address them in the future.

6.3.2 Ordered Lists

In EMF, some multi-valued references and attributes are *ordered* lists. Model assets introduced in Sec. 3.1 collectively represent all knowledge contained in an EMF model except for ordering information. Thus the lack of ordered lists is a limitation of the proposed solution. The core reason is that there is no unique way to provide PutBack for ordered lists that have been filtered; therefore such a lens would necessarily violate at least Prop. 7 and *undoability*. Finding an acceptable resolution of the problem (e.g. imposing a limitation that, for each user, ordered lists must be read-only unless entirely visible) is left as future work. For now, the proposed solution works properly for unordered collections.

6.3.3 Central Authority

Note that both **G2** and **G3** assume a central repository (owned by e.g. a system integrator) where the entire model is available. In a more general case, no single entity would be in possession of complete knowledge. There is an algebra [23] for combining lens transformations in various ways, suggesting a promising path for addressing this issue in future research. However, such a distributed scenario is out of scope for this paper; we address the centralized case, which is by far the most common in access control approaches used in model repositories.

7 Evaluation

We have carried out a scalability measurement in both offline and online scenario over the *Wind Turbine* case study [7] of the MONDO FP7 project. We state the following research questions in the evaluation:

Online Collaboration

Q1 Is the change propagation is incremental as it is requested in challenge **C3.1**?

Q1.1 How scalable is our approach to increasing *model size*?

Q1.2 How scalable is our approach to increasing number of active *users*?

Offline Collaboration

Q2 What is the overhead of using query-based access control over an existing VCS?

Q2.1 How scalable is our approach to increasing *model size*?

Q2.2 How scalable is our approach to increasing number of *front repositories*?

Q2.3 How scalable is our approach to increasing size of committed *changes*?

Finally, Sec. 6.3 will discuss limitations of our solution.

7.1 Scalability Evaluation

7.1.1 Measurement Setup

For the measurement, we used the simplified metamodel of Fig. 2 depicted in Fig. 13 which has slight modifications. The control unit `types` were abstracted to a string attribute, with K different permitted values to provide K different specialist, and the attributes of signals are removed. The corresponding access control rules are similar to our motivating example Sec. 2, with one specialist engineer for each control unit type (each having five access control rules dedicated to them) and an additional *system administrator* user who has read and write permission for the entire model. This means altogether $K + 1$ users and $5K + 5$ access control rules as it is shown in Fig. 14.

Measurements were performed with gold instance models of various size. The model of size M contains a root `Composite` object, which contains M copies of the structure depicted in Fig. 15. This means $1+M$ composite modules, $2M$ control units, $8M$ signals where $3M$ of them are confidential and $14M + M$ references where $4M$ of them are *consumes* cross-references. The copies are not completely identical: the `vendor` attributes are set to a different value in each copy; and `type` as well as `cycle` attributes of control units were chosen randomly from their respective ranges with uniform distribution. However, special care was taken to ensure that all control unit types must occur at least once; this also implies $2M \geq K$.

The measurement was performed with $U \leq K$ specialist users and the system administrator being present (thus in total $U + 1$ front models).

Online case. To test the incremental behavior of the lens transformation addressed by *Q1*, we measured the time it took the system administrator to perform a complex model manipulation operation on his front model, and to propagate the changes to the front models of all users who can see it. The measured complex operation is a *signal reversal* (depicted in Fig. 16), which reverts the direction of a communication channels by changing the *provides* and *consumes* to the opposite.

We have selected this representative operation since (a) it involves adding and removing cross-references and

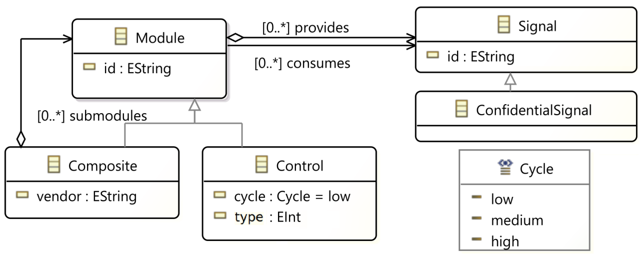


Fig. 13: Modified Metamodel of Wind Turbine

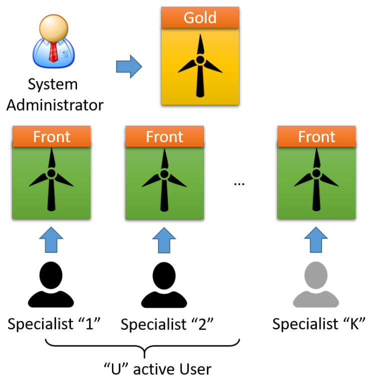


Fig. 14: Users and active user in the measurement setup

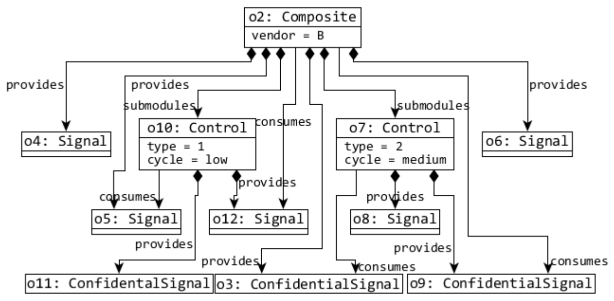


Fig. 15: Core structure of synthesized models

a rearrangement of the containment hierarchy; (b) it does not change the size of the model, thus introduces no bias of this kind; (c) the change is noticeable by all users that can see at least one of the involved modules in their front models; and (d) every access control rule in the policy (except for hiding the vendor attribute) plays a role in determining the impact of the change.

Offline case. The measurement focuses on the overhead of the collaboration framework required for propagating a change in the front model addressed by *Q2*. We measured the time it took a specific specialist to (1) propagate several number of complex model manipulation operation on her front model to the gold model and (2) from the gold model to the remaining front models of all users who can see the effect of the

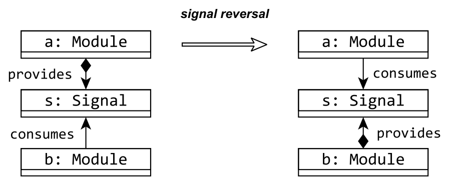


Fig. 16: Signal Reversal Operation

changes. The former describes the *response time* that a user has to wait for receiving the result (success/failure) of her commit while the latter is the *propogation time* to propogate changes to the other front repositories. The measured complex operation is a *signal addition* (depicted in Fig. 17), which adds a new signal under the root object.

We have selected this representative operation since (a) it demonstrates that any number of new changes can be introduced into the model; (b) it increases the model size but always with constant-size addition; (c) all the users can see the change in their front model; and (d) every access control rule in the policy (except for hiding the vendor attribute) plays a role in determining the impact of the change.

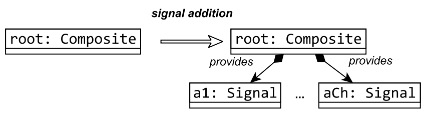


Fig. 17: Signal Addition Operation

Hardware Configuration

All the measurements[5] executed on a personal computer[6]. with maximum a 7GB of Java heap size.

7.1.2 Measurements of Online case

In the *model size scalability* series, we used

– fixed number of $K = 50$ control unit types and $U = 10$ present collaborators
– with increasing size of the model ranging from $M = 50$ to $M = 700$ (7701 objects, 10500 references)

In the *active users scalability* series, we used

– fixed number of $K = 100$ control unit types and model of size $M = 200$ (2301 objects, 3100 references),

[5] Raw data and reproduction instructions at `https://tinyurl.com/sosym-access-control`
[6] CPU: Intel Core i7-4700MQ@2.40GHz, MEM: 8GB

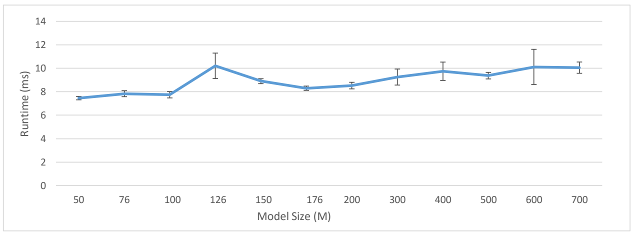


Fig. 18: Average execution time of an online signal reversal (increasing model size)

– with the increasing number of specialist collaborators joining the session ranging from $U = 2$ to $U = 100$.

For accuracy, 100 reversal operations were carried out and their execution times averaged in a single run; we have plotted the median execution time of 10 runs, excluding 2 warm-up runs, with 1 standard deviation error bars.
The results of the *model size scalability* series are shown in Fig. 18 addressing *Q1.1*. The cost of performing a single reversal model manipulation is low, and seems to be independent from the model size. This confirms that we have achieved incrementality where computation cost is dependent on the size of the change, but not on the size of the entire model.

The results of the *active users scalability* series are shown in Fig. 19 addressing *Q1.2*. It is apparent that when very few users join the session, most signal reversals are not visible to any user other than the principal engineer; but as more and more specialist users join the session, the number of active users starts to dominate the cost of model manipulation. Asymptotically, the cost of model manipulation is proportional to the average number of front models it is propagated to.

Note that it has only a small chance that users concurrently modify their front model as we mentioned in Sec. 6.2.3, but in that case the operations which arrive later to the server will be rejected. Hence, concurrent modifications have no additional effect on the performance.

Concurrent users are replaced with active users and a reason is introduced why concurrent users does not cause performance issues.

7.1.3 Measurements of Offline case

In the *model size scalability* series, we used

– fixed number of $K = 100$ control unit types, $Fr = 20$ front repositories and $Ch = 10$ changes,
– where the model is increased from $M = 100$ to $M = 6000$ (34001 objects, 45000 references).

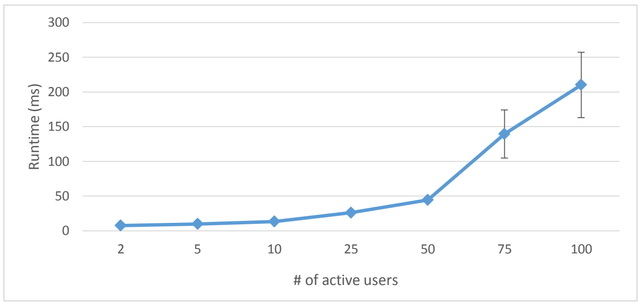


Fig. 19: Average execution time of an online signal reversal (increasing the number of active users)

In the *number of front repository scalability* series, we used

– fixed number of $K = 100$ control unit types, model of size $M = 800$ (8801 objects, 12000 references) and $Ch = 10$ changes,
– where the number of front repositories is increased from $Fr = 5$ to $Fr = 100$.

In the *change size scalability* series, we used

– fixed number of $K = 100$ control unit types, model of size $M = 400$ and $Fr = 20$ front repositories,
– where the number of introduced changes is increased from $Ch = 10$ to $Ch = 1000$.

The measurements were executed 10 times with 2 warm-up execution in separate JVM and the results show the median of the measured values.
The charts represent the entire transformation time including the following tasks: (1) loading the EMF models, (2)initializing the lens by building the correspondence tables, (3) loading the additional files such as rules and queries, (4) executing the transformation and (5) finally serializing the results as a committable new version of the models.

The lower part of the bars (denoted by checkered blue background) represents *response time* including the PutBack phase of the transformation. This is the delay experienced by committing users before they receive their response from the server so that they can continue their work. The upper part of the bars (in solid blue color) visualizes *propagation time* of the changes to synchronize with the rest of the front repositories (this happens asynchronously from the point of view of the committing user).

The results of *model size scalability* series are shown in Fig. 20 addressing *Q2.1*. In case of the largest model, users should wait at most 10 seconds to commit their changes in addition to the default execution time of a commit in the version control system. Response time

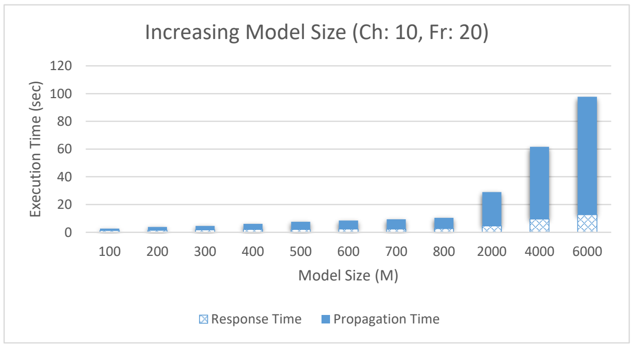


Fig. 20: Average execution time of an offline signal addition (increasing model size)

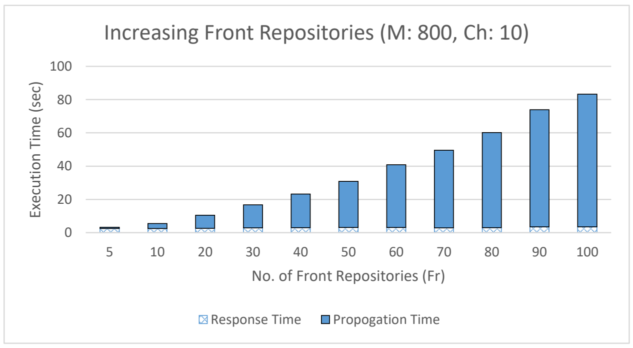


Fig. 21: Average execution time of an offline signal addition (increasing number of front repositories)

grows linearly with the size of the model while synchronization is non-linear to the model size.

The results of *number of front repository scalability* series are shown in Fig. 21 addressing *Q2.2*. In case of our special signal addition change, all front repositories had to be updated to propagate the changes and the same number of modification had to be executed on those front models. The results clearly show that the execution time grows linearly to the number of front repositories to which the change has to be propagated.

The results of the third series are shown in Fig. 22 addressing *Q2.3*. It shows that loading the models and building the correspondence table dominate the execution time in case of small changes. However, the execution time grows linearly with the size of changes in case of large commits (e.g. from 1000), for the sole reason that the resulting model size itself is increased due to the addition of so many new signals.

7.1.4 Discussion on Performance Findings

As seen from the measurements, the overhead on the commit time experienced by a committer in the offline

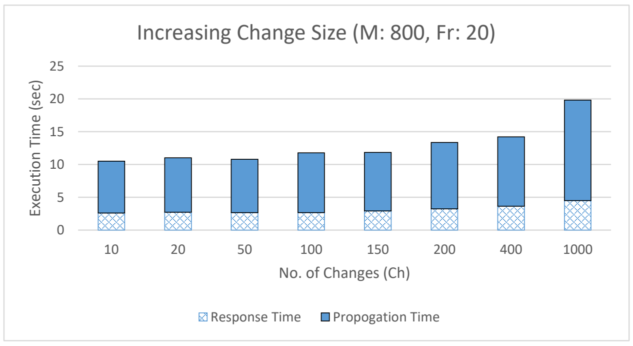


Fig. 22: Average execution time of an offline signal addition (increasing number of changes)

scenario is manageable, but can easily reach several seconds for larger models with tens of thousands of elements. This time is still significantly shorter than typical build and test execution times in continuous integration solutions, so our access control service is unlikely to form the bottleneck of developer productivity. Note that while the implementation of our prototype could certainly be improved, it will always have an overhead that is at least proportional to the model size, since the entire new model file has to be read and processed upon each commit (no matter how small the change within that model). This is a characteristic of file-based offline collaboration (required to meet goal G2).

One way to get around this limitation is to use online collaboration instead, where the execution time overhead on model modification is very low, even for a few dozen simultaneous collaborators. This can be seen as a space-time tradeoff, as online collaboration uses in-memory models, putting a limit on the amount of online sessions and participants that a given server can support. We therefore recommend the adaptation of online collaboration whenever possible for models that are currently under very active development, and using the file-based offline interface for other cases, such as accessing rarely updated models, old revisions, side branches, and of course for working on a disconnected computer.

8 Related Work

In this section, we collect the state-of-the-art approaches for specifying and enforcing fine-grained access control over various technological domains, and compare it to our solution.

8.1 Fine-grained Policies in Various Domains

8.1.1 File-based Access Control.

Traditional version control systems (like CVS, SVN) and file sharing technologies adopt file-level access policies, which are clearly insufficient for fine-grained access control specifications.

Off-the-shelf file systems typically require resources (files and folders) to be explicitly labeled with permissions that take the form of an Access Control List (ACL), or the simplified form user/group/other flags. An ACL consists of entries regarding which user/subject is granted or denied permission for a given operation.

File-based solutions can be directly applied to MDE, but cannot provide fine-grained access control, where different parts of a model file have different permissions. Our policies are fine-grained, use implicit rules (so that model elements do not have to be explicitly annotated with permission flags, which is difficult to manually maintain as the model evolves), and respect internal consistency (such as permission dependencies of cross-references); all the while being more flexible [20] in the conflict resolution method.

8.1.2 Access Control in RDF Triple/Quad stores.

Graph-based access control is a popular strategy for many triple and quad stores (4store [31], Virtuoso , IBM DB2) developed for storing large RDF data. User privileges can be granted to for each named graph while access control is actually checked when issuing a SPARQL query. Denial of access for a graph filters the query results obtained from this specific graph. Data access in AllegroGraph [28] can be controlled on the database or catalog level (coarse-grained) as well as on the graph and triple level (fine-grained) while Stardog only allows database-level access control.

Similarly to our approach, fine-grained access control is discussed in [22] using graph queries as preconditions of rules to select certain assets on which the permissions need to be enforced. The major difference is that we apply queries in an MDE environment (this has very important implications relative to RDF, see Sec. 3.1), and we also provide offline collaboration.

In the Oracle Database Semantic Technologies [42], access control is carried out by default on the model (graph) level. Furthermore, it can be configured on the triple (row) level, which is implemented by query rewriting. In this case, the definition of access control policies is based on so-called match and apply (graph) patterns, where the former identifies the type of access restriction while the latter injects access-control specific constraints to the query.

Another access control technique is called *label based security*, which offers (1) triple-level control using (a hierarchy of) sensitivity labels attached to each triple, and (2) RDF resource-level access control for subject/predicate/object. Explicit data access labels are implemented in [42] and are generalized into abstract tokens and operators in [43].

8.1.3 Access Control for XML Documents.

A number of standards such as XACML [33] (OASIS standard) provide fine-grained access control for XML documents. These type of documents are similar to models in a way, that they consists of nodes with attributes that may contain other nodes. XACML provides several combining algorithms to select from contradicting policies. In [29], fine-grained access control is formalized using XPath for XML documents, which claims that the visibility of a node depends on its ancestors, thus when a node is granted access, then access is also granted to its descendants. However, other dependencies are not discussed related to XML Documents.

Similarly to our approach, a dedicated policy language is used by [40], from which a lens is automatically generated to enforce access control for XML documents. In addition to the attributes and context of the assets (XML nodes), the XQuery-based policy can take into account external (subject or context) attributes as well. As it is not an MDE approach, there is no treatment of cross-references. There is no discussion of internal consistency either (see Sec. 3.2), except for the containment hierarchy, which is relevant for XML as well. Finally, there is no discussion of the challenges of online and offline collaboration.

8.1.4 Access control in Collaborative Modeling Environments.

Currently, fine-grained access control is not considered in the state of the art tools of MDE such as MetaEdit+ [53], VirtualEMF [16], WebGME [38], EMFStore [50], GenMyModel [6], Obeo Designer Team [41], MDE-Forge [9] or the tools developed according to [30]. See also the broader survey in [44].

The generic framework CDO [49] (used e.g. in [41]) provides both online collaboration and role-based access control with type-specific (class, package and resource-level) permissions, but no facility for instance level access control policy specifications. However, there is a pluggable access control mechanism that can specify access on the object level; it should be possible to integrate fine-grained solutions such as the currently proposed system.

The collaborative hardware design platform VehicleFORGE stores their model in graph-based databases and has an access control scheme TrustForge [19] that uses an implementation of KeyNote [13] trust management system. This system is responsible for evaluating the request addressed to the database, which can be configured in various ways. It supports unlimited permission levels and it is also able to handle consistency constraints by adding them as assertions. Conflict resolution strategies are not discussed. AToMPM [48] provides fine-grained role-based access control for online collaboration; no offline scenario or query-based security is supported, though. Access control is provided at elementary manipulation level (RESTful services) in the online collaboration solution of [25].

8.2 Access Control Enforcement

8.2.1 Access control using Model Transformation.

Closest to our approach, [39] uses model transformations to build infrastructures that can manage access control policies written in any policy language. It means that their approach takes a policy model as input and derives transformation rules to enforce read and write permissions. To compare it to our solution, we use elementary model transformation rules that take the effective permissions as input, instead of integrating the permissions into our rules.

8.2.2 Model-driven Security.

Model-based techniques have also been used for access control purposes. In [36], similarly to our solution, access control is enforced at runtime by program code that has been automatically generated from a model-based specification, which captures both system and security policy descriptions. This technique can provide runtime checks only on single entities by using the guarded object design pattern. A similar approach is suggested by [14], which specifies access control policies by OCL. Although this idea enables the formulation of queries that involve several objects, the efficient checking of these complex structural queries highly depends on the algorithmic experience of the system designer due to the fact that OCL handles model navigation in an imperative style, in contrast to declarative graph patterns, where several sophisticated pattern matching algorithms are readily available.

The book chapter [37] about Model-driven Security provides a detailed survey of a wide range of MDE approaches for designing secure systems, but does not cover the security of the MDE process itself.

8.2.3 Access Control using Bidirectional Programming.

Bidirectional Programming (BP) is an approach for defining lenses concisely, e.g. by only specifying one of GET and PUTBACK, and deriving the other. Such lenses can be directly applied for read filtering. However, [27] demonstrates that conventional BP is not sufficient for write access control. It also proposes such an integrity-preserving BP extension, focusing on string transformations (and therefore not directly applicable in MDE). There is no notion of access control policy either, so the security engineer has to develop their own lens transformation to implement access control.

9 Conclusion and Future Work

In this paper, we aimed to uniformly enhance secure collaborative modeling by using fine-grained access control policies uniformly for online and offline collaboration scenarios. Each collaborator can access a dedicated copy of the model in accordance with read permissions of the policy. Moreover, bidirectional transformations are used to synchronize changes between different collaborators and check that write permissions are also respected.

We illustrated our techniques in the context of a Wind Turbine case study from the MONDO European Project, which was also used to assess scalability with models of increasing size, increasing change introduced by collaborators and increasing number of collaborators. In case of online collaboration, the results were promising with close to instant propagation of changes and checking of write permissions. In case of offline collaboration, the results show that the response time is acceptable and the overhead is less than 10 additional seconds for the largest model).

As future work, we would like to (i) address the limitations presented in Sec. 6.3, (ii) investigate the possibilities of building correspondence relations between the original model and filtered copy of it dedicated to a certain collaborator, and (iii) realize our collaboration scheme with other frameworks (e.g. Git, GenMyModel) and with support for continuous integration and review / change request management systems.

10 Acknowledgment

This paper is partially supported by the EU Commission with project MONDO (FP7-ICT-2013-10), no. 611125., the MTA-BME Lendület 2015 Research Group on Cyber-Physical Systems and NSERC RGPIN-04573-16. The second author was supported by the János Bolyai Research Scholarship of the Hungarian Academy of Sciences.

References

1. CAESAR Research Project. `http://store.sae.org/caesar/`.
2. OMG Object Constraint Language, February 2014. `http://www.omg.org/spec/OCL/`.
3. The Cambridge Dictionary, 2017. `http://dictionary.cambridge.org/dictionary/english/obfuscate`.
4. Aerospace vehicle systems institute. SAVI Research Project. `http://http://savi.avsi.aero/`.
5. Apache. Subversion. 07 2017.
6. Axellience. GenMyModel. `http://www.genmymodel.com`.
7. A. Bagnato, E. Brosse, A. Sadovykh, P. Maló, S. Trujillo, X. Mendialdua, and X. De Carlos. Flexible and scalable modelling in the mondo project: Industrial case studies. In *XM@ MoDELS*, pages 42–51, 2014.
8. F. Bancilhon and N. Spyratos. Update semantics of relational views. *ACM Trans. Database Syst.*, 6(4):557–575, Dec. 1981.
9. F. Basciani, J. D. Rocco, D. D. Ruscio, A. D. Salle, L. Iovino, and A. Pierantonio. MDEForge: an extensible web-based modeling platform. In *CloudMDE@MoDELS*, 2014.
10. G. Bergmann, I. Dávid, Á. Hegedüs, Á. Horváth, I. Ráth, Z. Ujhelyi, and D. Varró. VIATRA 3: a reactive model transformation platform. In *International Conference on Theory and Practice of Model Transformations*, pages 101–110. Springer, 2015.
11. G. Bergmann, C. Debreceni, I. Ráth, and D. Varró. Query-based Access Control for Secure Collaborative Modeling using Bidirectional Transformations. In *ACM/IEEE 19th Int. Conf. on MODELS*, 2016.
12. G. Bergmann, C. Debreceni, I. Ráth, and D. Varró. Towards efficient evaluation of rule-based permissions for fine-grained access control in collaborative modeling. In *2st International Workshop on Collaborative Modelling in MDE*, Austin Texas, USA, In Press. ACM.
13. M. Blaze and A. D. Keromytis. The keynote trust-management system version 2. 1999.
14. R. Breu, G. Popp, and M. Alam. Model based development of access policies. *International Journal on Software Tools for Technology Transfer*, 9(5):457–470, 2007.
15. M. Chechik, F. Dalpiaz, C. Debreceni, J. Horkoff, I. Ráth, R. Salay, and D. Varró. Property-based methods for collaborative model development. In *Joint Proc. of the 3rd Int. Workshop on the Glob. Of Modeling Lang. and the 9th Int. Workshop on Multi-Paradigm Modeling*, pages 1–7. Citeseer, 2015.
16. C. Clasen, F. Jouault, and J. Cabot. VirtualEMF: A model virtualization tool. In *Advances in Conceptual Modeling. Recent Developments and New Directions*, pages 332–335, 2011.
17. N. Conner. *Google Apps: The Missing Manual: The Missing Manual.* " O'Reilly Media, Inc.", 2008.
18. K. Czarnecki and S. Helsen. Feature-based survey of model transformation approaches. *IBM Systems Journal*, 45(3):621–645, 2006.
19. P. U. DARPA VehicleFORGE. *TrustForge: Flexible Access Control for VehicleForge.mil Collaborative Environment*, 2012.
20. C. Debreceni, G. Bergmann, I. Ráth, and D. Varró. Deriving effective permissions for modeling artifacts from fine-grained access control rules. In *1st International Workshop on Collaborative Modelling in MDE*, Saint Malo, France, 06 2016. ACM.
21. C. Debreceni, I. Ráth, D. Varró, X. De Carlos, X. Mendialdua, and S. Trujillo. Automated model merge by design space exploration. In *International Conference on Fundamental Approaches to Software Engineering*, pages 104–121. Springer, 2016.
22. S. Dietzold and S. Auer. S.: Access control on RDF triple stores from a semantic wiki perspective. In *In: Scripting for the Semantic Web Workshop at 3rd European Semantic Web Conference (ESWC*, 2006.
23. Z. Diskin. Algebraic models for bidirectional model synchronization. In *MoDELS*, pages 21–36, 2008.
24. H. Ehrig, K. Ehrig, U. Prange, and G. Taentzer. *Fundamentals of Algebraic Graph Transformation (Monographs in Theoretical Computer Science. An EATCS Series).* Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2006.
25. M. Farwick, B. Agreiter, J. White, S. Forster, N. Lanzanasto, and R. Breu. A web-based collaborative metamodeling environment with secure remote model access. In *Web Engineering, 10th International Conference, ICWE 2010, Vienna, Austria, July 5-9, 2010. Proceedings*, volume 6189 of *LNCS*, pages 278–291. Springer, 2010.
26. K. F. Fogel and M. Bar. *Open source development with CVS.* Coriolis Group Books, 2001.
27. J. N. Foster, B. C. Pierce, and S. Zdancewic. Updatable security views. In *Proceedings of the 2009 22Nd IEEE Computer Security Foundations Symposium*, CSF '09, pages 60–74, Washington, DC, USA, 2009. IEEE Computer Society.
28. I. Franz. AllegroGraph. `http://franz.com/agraph/allegrograph/doc/security.html`.
29. I. Fundulaki and M. Marx. Specifying access control policies for XML documents with XPath. In *9th ACM Symposium on Access Control Models and Technologies*, pages 61–69, 2004.
30. J. Gallardo, C. Bravo, and M. A. Redondo. A model-driven development method for collaborative modeling tools. *J. Network and Computer Applications*, 35(3):1086–1105, 2012.
31. Garlik. 4store. `http://4store.org/trac/wiki/GraphAccessControl`.
32. T. Gibson-Robinson, P. Armstrong, A. Boulgakov, and A. Roscoe. FDR3 — A Modern Refinement Checker for CSP. In E. Ábrahám and K. Havelund, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, volume 8413 of *Lecture Notes in Computer Science*, pages 187–201, 2014.
33. S. Godik and T. M. (eds). eXtensible access control markup language (XACML) version 1.0. 02 2003.
34. Int. Organization for Standardization. *ISO 16739:2013: Industry Foundation Classes (IFC) for data sharing in the construction and facility management industries*, 04 2013.

35. R. Jaeschke. Encrypting c source for distribution. *Journal of C Language Translation*, 2(1):71–80, 1990.
36. J. Jürjens. Model-based run-time checking of security permissions using guarded objects. In M. Leucker, editor, *Proc. of the 8th International Workshop on Runtime Verification*, volume 5289 of *LNCS*, pages 36–50, Budapest, Hungary, 2008. Springer.
37. L. Lucio, Q. Zhang, P. H. Nguyen, M. Amrani, J. Klein, H. Vangheluwe, and Y. L. Traon. Advances in model-driven security. *Advances in Computers*, 93:103–152, 2014.
38. M. Maroti et al. Next Generation (Meta)Modeling: Web- and Cloud-based Collaborative Tool Infrastructure. In *8th Multi-Paradigm Modeling Workshop*, Valencia, Spain, 09/2014 2014.
39. S. Martínez, J. García, and J. Cabot. Runtime support for rule-based access-control evaluation through model-transformation. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Software Language Engineering*, pages 57–69. ACM, 2016.
40. L. Montrieux and Z. Hu. Towards attribute-based authorisation for bidirectional programming. In *Proceedings of the 20th ACM Symposium on Access Control Models and Technologies*, SACMAT '15, pages 185–196, New York, NY, USA, 2015. ACM.
41. Obeo. Obeo designer team. `https://www.obeodesigner.com/en/collaborative-features`.
42. Oracle. Database Semantic Technologies. `http://docs.oracle.com/cd/E11882_01/appdev.112/e11828/fine_grained_acc.htm`.
43. V. Papakonstantinou, M. Michou, I. Fundulaki, G. Flouris, and G. Antoniou. Access control for RDF graphs using abstract models. In *17th ACM Symposium on Access Control Models and Technologies, SACMAT '12, Newark, NJ, USA - June 20 - 22, 2012*, pages 103–112. ACM, 2012.
44. J. D. Rocco, D. D. Ruscio, L. Iovino, and A. Pierantonio. Collaborative repositories in model-driven engineering [software technology]. *IEEE Software*, 32(3):28–34, May 2015.
45. A. W. Roscoe. *Understanding concurrent systems*. Springer Science & Business Media, 2010.
46. B. Roscoe. The theory and practice of concurrency. 1998.
47. P. Stevens. Bidirectional model transformations in QVT: semantic issues and open questions. *Software & Systems Modeling*, 9(1):7–20, 2008.
48. E. Syriani, H. Vangheluwe, R. Mannadiar, C. Hansen, V. Mierlo, and H. Ergin. AToMPM: A Web-based Modeling Environment. MODELS 2013 Demonstrations Track, 2013.
49. The Eclipse Foundation. CDO. `http://www.eclipse.org/cdo`.
50. The Eclipse Foundation. EMFStore. `http://www.eclipse.org/emfstore`.
51. The Eclipse Foundation. RAP. `http://www.eclipse.org/rap/`.
52. The Eclipse Project. Eclipse Modeling Framework. `http://www.eclipse.org/emf/`.
53. J. Tolvanen. MetaEdit+: Domain-specific modeling and product generation environment. In *Software Product Lines, 11th Int. Conf. SPLC 2007, Kyoto, Japan*, pages 145–146, 2007.
54. D. Varró, G. Bergmann, Á. Hegedüs, Á. Horváth, I. Ráth, and Z. Ujhelyi. Road to a reactive and incremental model transformation platform: three generations of the viatra framework. *Software & Systems Modeling*, 15(3):609–629, 05/2016 2016.
55. J. Whittle, J. E. Hutchinson, and M. Rouncefield. The state of practice in model-driven engineering. *IEEE Software*, 31(3):79–85, 2014.

A Get Transformation Rules

rule	Subtractive GET Reference	**Priority**	1
Precondition :: $(M_G(OA_G, RA_G, AA_G), M_F(OA_F, RA_F, AA_F), permission_{\mathrm{Eff}}) \rightarrow \{ReferenceAsset\}$			
$\{ReferenceAsset(s_F, ref, t_F) \mid ReferenceAsset(s_F, ref, t_F) \in RA_F,$			
$\exists ObjectAsset(s_F, t_s) \in OA_F,\ ObjectAsset(t_F, t_T) \in OA_F : trace(ObjectAsset(s_G, t_s)) = ObjectAsset(s_F, t_s),$			
$trace(ObjectAsset(t_G, t_t)) = ObjectAsset(t_F, t_t),\ \nexists ReferenceAsset(s_G, ref, t_G) : ReferenceAsset(s_G, ref, t_G) \in RA_G$			
$permission_{\mathrm{Eff}}(ReferenceAsset(s_G, ref, t_G), read) \neq deny\}$			
Action			
$RA_F := RA_F \setminus \{ReferenceAsset(s_F, ref, t_F)\}$			

rule	Subtractive GET Attribute	**Priority**	2
Precondition :: $(M_G(OA_G, RA_G, AA_G), M_F(OA_F, RA_F, AA_F), permission_{\mathrm{Eff}}) \rightarrow \{AttributeAsset\}$			
$\{AttributeAsset(o_F, attr, v') \mid AttributeAsset(o_F, attr, v') \in AA_F, \exists ObjectAsset(o_F, t) : ObjectAsset(o_F, t) \in OA_F$			
$\exists ObjectAsset(o_G, t) : ObjectAsset(o_G, t) \in OA_G, trace(ObjectAsset(o_G, t)) = ObjectAsset(o_F, t),\ \nexists AttributeAsset(o_G, attr, v) :$			
$v' = \begin{cases} v, permission_{\mathrm{Eff}}(AttributeAsset(o_G, attr, v), read) = allow \\ obf(v), permission_{\mathrm{Eff}}(AttributeAsset(o_G, attr, v), read) = obfuscate \end{cases} , AttributeAsset(o_G, attr, v) \in AA_G\}$			
Action			
$AA_F := AA_F \setminus \{AttributeAsset(o_F, attr, v')\}$			

rule	Subtractive GET Object	**Priority**	3
Precondition :: $(M_G(OA_G, RA_G, AA_G), M_F(OA_F, RA_F, AA_F), permission_{\mathrm{Eff}}) \rightarrow \{ObjectAsset\}$			
$\{ObjectAsset(o_F, t) \mid ObjectAsset(o_F, t) \in OA_F,\ \nexists ObjectAsset(o_G, t') : trace(ObjectAsset(o_G, t')) = ObjectAsset(o_F, t), t = t'$			
$permission_{\mathrm{Eff}}(ObjectAsset(o_G, t'), read) \neq deny\}$			
Action			
$OA_F := OA_F \setminus \{ObjectAsset(o_F, t)\}, trace \setminus ObjectAsset(o_G, t') \mid trace(ObjectAsset(o_G, t')) = ObjectAsset(o_F, t)$			

rule	Additive GET Object	**Priority**	4
Precondition :: $(M_G(OA_G, RA_G, AA_G), M_F(OA_F, RA_F, AA_F), permission_{\mathrm{Eff}}) \rightarrow \{ObjectAsset\}^2$			
$\{ObjectAsset(o_G, t), ObjectAsset(o_F, t') \mid ObjectAsset(o_G, t) \in OA_G, permission_{\mathrm{Eff}}(ObjectAsset(o_G, t), \mathrm{read}) \neq \mathrm{deny},$			
$\nexists ObjectAsset(o_F, t') \in OA_F : trace(ObjectAsset(o_G, t)) = ObjectAsset(o_F, t'), t = t'\}$			
Action			
$OA_F := OA_F \cup \{ObjectAsset(o_F, t')\}, trace(ObjectAsset(o_G, t)) := ObjectAsset(o_F, t')$			

rule	Additive GET Attribute	**Priority**	5
Precondition :: $(M_G(OA_G, RA_G, AA_G), M_F(OA_F, RA_F, AA_F), permission_{\mathrm{Eff}}) \rightarrow \{AttributeAsset\}$			
$\{AttributeAsset(o_F, attr, v') \mid AttributeAsset(o_G, attr, v) \in AA_G, permission_{\mathrm{Eff}}(AttributeAsset(o_G, attr, v), read) \neq deny,$			
$\exists ObjectAsset(o_F, t) : ObjectAsset(o_F, t) \in AA_F, trace(ObjectAsset(o_G, t)) = ObjectAsset(o_F, t),\ \nexists AttributeAsset(o_F, attr, v') :$			
$v' = \begin{cases} v, permission_{\mathrm{Eff}}(AttributeAsset(o_G, attr, v), read) = allow \\ obf(v), permission_{\mathrm{Eff}}(AttributeAsset(o_G, attr, v), read) = obfuscate \end{cases} , AttributeAsset(o_F, attr, v') \in AA_F\}$			
Action			
$AA_F := AA_F \cup \{AttributeAsset(o_F, attr, v')\}$			

rule	Additive GET Reference	**Priority**	6
Precondition :: $(M_G(OA_G, RA_G, AA_G), M_F(OA_F, RA_F, AA_F), permission_{\mathrm{Eff}}) \rightarrow \{ReferenceAsset\}$			
$\{ReferenceAsset(s_F, ref, t_F) \mid ReferenceAsset(s_G, ref, t_G) \in RA_G, permission_{\mathrm{Eff}}(ReferenceAsset(s_G, ref, t_G), read) \neq deny,$			
$\exists ObjectAsset(s_G, t_s), ObjectAsset(t_G, t_t) : trace(ObjectAsset(s_G, t_s)) = ObjectAsset(s_F, t_s),$			
$trace(ObjectAsset(t_G, t_t)) = ObjectAsset(t_F, t_t),\ \nexists ReferenceAsset(s_F, ref, t_F) : ReferenceAsset(s_F, ref, t_F) \in RA_F\}$			
Action			
$RA_F := RA_F \cup \{ReferenceAsset(s_F, ref, t_F)\}$			

B PutBack Transformation Rules

rule	Subtractive PutBack Reference	**Priority**	1
Precondition :: $(M_G(OA_G, RA_G, AA_G), M_F(OA_F, RA_F, AA_F), \mathit{permission}_{\mathrm{Eff}}) \rightarrow \{\mathit{ReferenceAsset}\}$			
$\{\mathit{ReferenceAsset}(s_G, ref, t_G) \mid \mathit{ReferenceAsset}(s_G, ref, t_G) \in RA_G, \mathit{permission}_{\mathrm{Eff}}(\mathit{ReferenceAsset}(s_G, ref, t_G), read) \neq deny,$ $\exists \mathit{ObjectAsset}(s_G, t_s), \mathit{ObjectAsset}(t_G, t_t) : trace(\mathit{ObjectAsset}(s_G, t_s)) = \mathit{ObjectAsset}(s_F, t_s),$ $trace(\mathit{ObjectAsset}(t_G, t_t)) = \mathit{ObjectAsset}(t_F, t_t), \nexists \mathit{ReferenceAsset}(s_F, ref, t_F) : \mathit{ReferenceAsset}(s_F, ref, t_F) \in RA_F\}$			
Action			
If $\mathit{permission}_{\mathrm{Eff}}(\mathit{ReferenceAsset}(s_G, ref, t_G), write) \neq deny$ then $RA_G := RA_G \setminus \mathit{ReferenceAsset}(s_G, ref, t_G)$ else ⤨			

rule	Subtractive PutBack Attribute	**Priority**	2
Precondition :: $(M_G(OA_G, RA_G, AA_G), M_F(OA_F, RA_F, AA_F), \mathit{permission}_{\mathrm{Eff}}) \rightarrow \{\mathit{AttributeAsset}\}$			
$\{\mathit{AttributeAsset}(o_G, attr, v) \mid \nexists \mathit{AttributeAsset}(o_F, attr, v') :$ $\exists \mathit{ObjectAsset}(o_F, t) : \mathit{ObjectAsset}(o_F, t) \in OA_F, trace(\mathit{ObjectAsset}(o_G, t)) = \mathit{ObjectAsset}(o_F, t),$ $v = \begin{cases} v', \mathit{permission}_{\mathrm{Eff}}(\mathit{AttributeAsset}(o_G, attr, v), read) = allow \\ obf^{-1}(v'), \mathit{permission}_{\mathrm{Eff}}(\mathit{AttributeAsset}(o_G, attr, v), read) = obfuscate \end{cases}, \mathit{AttributeAsset}(o_G, attr, v) \in AA_G\}$			
Action			
If $\mathit{permission}_{\mathrm{Eff}}(\mathit{AttributeAsset}(o_G, attr, v), write) \neq deny$ then $AA_G := AA_G \setminus \mathit{AttributeAsset}(o_G, attr, v)$ else ⤨			

rule	Subtractive PutBack Object	**Priority**	3
Precondition :: $(M_G(OA_G, RA_G, AA_G), M_F(OA_F, RA_F, AA_F), \mathit{permission}_{\mathrm{Eff}}) \rightarrow \{\mathit{ObjectAsset}\}$			
$\{\mathit{ObjectAsset}(o_G, t) \mid \mathit{ObjectAsset}(o_G, t) \in OA_G, \mathit{permission}_{\mathrm{Eff}}(\mathit{ObjectAsset}(o_G, t), \mathrm{read}) \neq \mathrm{deny},$ $\nexists \mathit{ObjectAsset}(o_F, t') : trace(\mathit{ObjectAsset}(o_G, t)) = \mathit{ObjectAsset}(o_F, t'), t = t'\}$			
Action			
If $\mathit{permission}_{\mathrm{Eff}}(\mathit{ObjectAsset}(o_G, type), write) \neq deny$ then $OA_G := OA_G \setminus \mathit{ObjectAsset}(o_G, t), trace \setminus \mathit{ObjectAsset}(o_G, t) \mid trace(\mathit{ObjectAsset}(o_G, t)) = \mathit{ObjectAsset}(o_F, t)$ else ⤨			

rule	Additive PutBack Object	**Priority**	4
Precondition :: $(M_G(OA_G, RA_G, AA_G), M_F(OA_F, RA_F, AA_F), \mathit{permission}_{\mathrm{Eff}}) \rightarrow \{\mathit{ObjectAsset}\}^2$			
$\{\mathit{ObjectAsset}(o_F, t), \mathit{ObjectAsset}(o_G, t) \mid \mathit{ObjectAsset}(o_F, t) \in OA_F, \nexists \mathit{ObjectAsset}(o_G, t) : trace(\mathit{ObjectAsset}(o_G, t)) = \mathit{ObjectAsset}(o_F, t)\}$			
Action			
$OA_G := OA_G \cup \{\mathit{ObjectAsset}(o_G, t)\}$ If $\mathit{permission}_{\mathrm{Eff}}(\mathit{ObjectAsset}(o_G, t), write) \neq deny$ then $trace(\mathit{ObjectAsset}(o_G, t)) := \mathit{ObjectAsset}(o_F, t)$ else ⤨			

rule	Additive PutBack Attribute	**Priority**	5
Precondition :: $(M_G(OA_G, RA_G, AA_G), M_F(OA_F, RA_F, AA_F), \mathit{permission}_{\mathrm{Eff}}) \rightarrow \{\mathit{AttributeAsset}\}$			
$\{\mathit{AttributeAsset}(o_G, attr, v) \mid \mathit{AttributeAsset}(o_F, attr, v') \in AA_F, \exists \mathit{ObjectAsset}(o_F, t) : \mathit{ObjectAsset}(o_F, t) \in AA_F$ $\exists \mathit{ObjectAsset}(o_G, t) : \mathit{ObjectAsset}(o_G, t) \in OA_G, trace(\mathit{ObjectAsset}(o_G, t)) = \mathit{ObjectAsset}(o_F, t), \nexists \mathit{AttributeAsset}(o_G, attr, v) :$ $v = \begin{cases} v', \mathit{permission}_{\mathrm{Eff}}(\mathit{AttributeAsset}(o_G, attr, v), read) = allow \\ obf^{-1}(v'), \mathit{permission}_{\mathrm{Eff}}(\mathit{AttributeAsset}(o_G, attr, v), read) = obfuscate \end{cases}, \mathit{AttributeAsset}(o_G, attr, v) \in AA_G\}$			
Action			
$AA_G := AA_G \cup \{\mathit{AttributeAsset}(o_G, att, v)\}$ If $(\mathit{permission}_{\mathrm{Eff}}(\mathit{AttributeAsset}(o_G, att, v), write) = \mathrm{deny})$ then ⤨			

rule	Additive PutBack Reference	**Priority**	6
Precondition :: $(M_G(OA_G, RA_G, AA_G), M_F(OA_F, RA_F, AA_F), \mathit{permission}_{\mathrm{Eff}}) \rightarrow \{\mathit{ReferenceAsset}\}$			
$\{\mathit{ReferenceAsset}(s_G, ref, t_G) \mid \mathit{ReferenceAsset}(s_F, ref, t_F) \in RA_F,$ $\exists \mathit{ObjectAsset}(s_F, t_s), \mathit{ObjectAsset}(t_F, t_t) \in OA_F : trace(\mathit{ObjectAsset}(s_G, t_s)) = \mathit{ObjectAsset}(s_F, t_s),$ $trace(\mathit{ObjectAsset}(t_G, t_t)) = \mathit{ObjectAsset}(t_F, t_t), \nexists \mathit{ReferenceAsset}(s_G, ref, t_G) : \mathit{ReferenceAsset}(s_G, ref, t_G) \in RA_G\}$			
Action			
$RA_G := RA_G \cup \{\mathit{ReferenceAsset}(s_G, ref, t_G)\}$ If $\mathit{permission}_{\mathrm{Eff}}(\mathit{ReferenceAsset}(s_G, ref, t_G), write) = deny$ then ⤨			

C Proof sketches for the transformation properties listed in Sec. 4.3.1

Proof for Prop. 1: Termination

In the following, by *source model* we will mean the front model in case of PutBack, and the readable part of the gold model in case of Get. Note that in both cases, the set of assets contained in the source model, defined this way, is not modified during a transformation run.

Then we prove that for any given asset, either the additive or the subtractive rule may fire during a transformation run, but not both. This follows from the fact that the precondition of the additive rule requires the presence of the asset in the source model, while the subtractive rule requires its absence, and the source model does not change.

Note that the main action of each rule directly invalidates its precondition: additive rules always check for the non-existence of the asset to be created in the target model, while subtractive rules always check for the existence of the asset to be removed in the target model. Also, this main effect could only be possibly undone by the opposite rule (additive for subtractive and vice versa), which was shown above not to fire during the same transformation run. Therefore a rule, once fired for a given precondition match, will not fire again for the same match in the same transformation run.

Finally, on a source model of finite size, the preconditions of the rules have a finite number of matches, thus all the rule executions during a transformation run may be characterized by a finite number of distinct matches. As each precondition match is fired at most once, the transformation run must consist of a finite number of rule firings. It is trivial to see that individual rule actions are terminating.

Note that this proof, unlike the following ones, does not rely on Asm. 1; so the transformations can at least be known to terminate even when the conditions of regularity are not met.

Proof for Prop. 2: Confluence

Here we follow a proof strategy and terminology common in rewriting systems (see e.g. [24] for graph transformations). The proof will rely on both the above result for termination as well as Asm. 1.

First of all, our rules are *sequentially independent*, meaning that if there are two valid transformation runs that differ only by swapping two subsequent rule executions, then the end result of the two runs are the same. This is easily seen by examining the actions of the rules. Note in our case, because of the priority-based total ordering of rules, it is sufficient to check different precondition matches of the same rule, as no other rules are on the same priority level, making them unswappable.

Next, we also establish that the rules are *parallel independent*, meaning that if there are two distinct preconditions matches of the same priority (i.e. the same rule in our case), then firing the rule for one will never disable the other. For Get, this is also easily checked by inspecting the rules. For PutBack, however, we must take into account that with changes to the gold model, the effective permissions may also change. This is indeed a point where confluence could fail; but Asm. 1 guarantees that this does not happen. A further violation of parallel independence is caused by PutBack failing and stopping execution when a write permission check fails; in this case if we can pretend that this failure is registered, the transformation continues, and then only rejects the modification (rolling back all effects) once it has terminated.

Parallel independence guarantees that if a rule application is enabled at some point, it will stay enabled until it is fired, and therefore it will eventually be fired in a comlpete transformation run. The two kinds of independence together prove that any two complete transformation runs (from the same starting model) terminate with equivalent results, as in [24].

We have shown that the transformations are terminating and confluent rule systems; from here on, we rely on this observation, and treat the transformations induced by the rules as deterministic Get and PutBack functions.

Proof for Prop. 3: Confidentiality

The rule-based formalisation and the partitioning of assets based on permissions (made possible due to Asm. 1) makes it very straightforward to prove security properties.

It is easy to check that Get rules create assets in the front model only if they are readable, and will definitely remove them if they exist but are unreadable. A slight clarification needs to be made for obfuscated assets, that are not copied verbatim, but rather undergo an obfuscating transformation in the appropriate rules.

PutBack does not modify the front model.

Proof for Prop. 4: Integrity

A successful PutBack run only creates or removes assets in the gold model if they are writeable. Get does not modify the gold model, neither does a failed PutBack.

Proof for Prop. 5: GetPut

At the point where Get terminates, there are no matches for any of its preconditions. The goal is to prove that at this state, PutBack has no matches either.

As shown for Prop. 3, the front model at this point contains exactly those gold assets that are readable by the user (modulo obfuscation). It is easy to check that subtractive PutBack preconditions will only match if there is a readable asset that is in the gold model but not in the front model, while (almost, but not perfectly symmetrically) additive PutBack preconditions will only match if there is an asset in the front model that is not in the gold model; none of which can be satisfied in such a state.

Proof for Prop. 6: PutGet

At the point where a successful PutBackterminates, there are no matches for any of its preconditions. The goal is to prove that at this state, Get has no matches either.

A quick review of PutBack rules reveal that the front and gold models must agree (at least) on all readable assets at this point (modulo obfuscation), otherwise there would be precondition matches. It then follows trivially that Get preconditions cannot be satisfied in such a state.

Proof for Prop. 7: PutPut

As discussed before, this law is very restrictive, and it might have to be given up in order to allow for certain sensible extensions. Nevertheless, here we will prove that this property holds for the system as presented in this paper.

Thanks to Asm. 1, assets can be partitioned into a writeable part and its complement, in line with the *constant complement* [8] approach. It can be easily checked that all PutBack rules ensure that the changed asset is writeable; none of them affects the complement (this is the part that no longer holds once *dangle* permissions are introduced).

We have also established above that after a successful PutBack, the front and gold models must agree (at least) on all readable assets; consequently they also agree on all writeable assets, which is a subset of the former.

Since PutBack leaves the complement constant and overwrites the writeable part of the gold model with the front model, it follows that in a sequence of successful PutBack runs, only the final one matters.

D Collaboration Scheme Formalized as Communicating Sequential Processes

Algorithm 1 Collaboration Scheme Formalized as Communicating Sequential Processes

```
Range = 1..N
channel commit, update, checkout, needToUpdate, accessDenied, otherCommitUnderExecution, policyViolated, upToDate, failure, success, lock, unlock, putback, get: Range

$\textbf{Checkout}(x)$ = checkout.x $\rightarrow$ (success.x $\rightarrow$ SKIP $\square$ accessDenied.x $\rightarrow$ failure.x $\rightarrow$ SKIP)
$\textbf{Update}(x)$ = update.x $\rightarrow$ (success.x $\rightarrow$ SKIP $\square$ accessDenied.x $\rightarrow$ failure.x $\rightarrow$ SKIP $\square$ upToDate.x $\rightarrow$ failure.x $\rightarrow$ SKIP)
$\textbf{Pre-commit}_{\text{succ}}(x)$ = commit.x $\rightarrow$ lock.x $\rightarrow$ SKIP
$\textbf{Pre-commit}_{\text{fail}}(x)$ = commit.x $\rightarrow$ (accessDenied.x $\rightarrow$ failure.x $\rightarrow$ SKIP $\square$ needToUpdate.x $\rightarrow$ failure.x $\rightarrow$ SKIP)
$\textbf{Pre-commit}_{\text{reject}}(x)$ = commit.x $\rightarrow$ otherCommitUnderExecution.x $\rightarrow$ SKIP
$\textbf{Commit}_{\text{succ}}(x)$ = putback.x $\rightarrow$ success.x SKIP
$\textbf{Commit}_{\text{fail}}(x)$ = policyViolated.x $\rightarrow$ failure.x $\rightarrow$ SKIP
$\textbf{Post-commit}_{\text{sync}}(x, z)$ = if x $\neq$ z get.x $\rightarrow$ SKIP else SKIP
$\textbf{Post-commit}_{\text{succ}}(x)$ = unlock.x $\rightarrow$ SKIP
$\textbf{Post-commit}_{\text{fail}}(x)$ = unlock.x $\rightarrow$ SKIP

$\textbf{Server}_{\text{idle}}()$ = $\square$ y:Range @ $\textbf{Checkout}$(y); $\textbf{Server}_{\text{idle}}()$
        $\square$ $\textbf{Update}$(y); $\textbf{Server}_{\text{idle}}()$
        $\square$ $\textbf{Pre-commit}_{\text{succ}}(y)$; $\textbf{Server}_{\text{locked}}(y)$
        $\square$ $\textbf{Pre-commit}_{\text{fail}}(y)$ ; $\textbf{Server}_{\text{idle}}()$

$\textbf{Server}_{\text{locked}}(x)$ = $\square$ y:Range @ $\textbf{Checkout}$(y); $\textbf{Server}_{\text{locked}}(x)$
        $\square$ $\textbf{Update}$(y); $\textbf{Server}_{\text{locked}}(x)$
        $\square$ $\textbf{Pre-commit}_{\text{reject}}(y)$; $\textbf{Server}_{\text{locked}}(x)$
        $\square$ $\textbf{Commit}_{\text{succ}}(x)$; $\textbf{Server}^{\text{sync}}_{\text{unlocked}}(x, 1)$
        $\square$ $\textbf{Commit}_{\text{fail}}(x)$ ; $\textbf{Server}_{\text{locked}}(x)$

$\textbf{Server}^{\text{sync}}_{\text{unlocked}}(x, z)$ = $\square$ y:Range @ $\textbf{Checkout}$(y); $\textbf{Server}^{\text{sync}}_{\text{unlocked}}(x)$
        $\square$ $\textbf{Update}$(y); $\textbf{Server}^{\text{sync}}_{\text{unlocked}}(x)$
        $\square$ $\textbf{Pre-commit}_{\text{reject}}(y)$; $\textbf{Server}^{\text{sync}}_{\text{unlocked}}(x)$
        $\square$ $\textbf{Post-commit}_{\text{sync}}(x, z)$; if z $\neq$ N then $\textbf{Server}^{\text{sync}}_{\text{unlocked}}(x, z + 1)$ else $\textbf{Server}^{\text{succ}}_{\text{unlocked}}(x)$

$\textbf{Server}^{\text{succ}}_{\text{unlocked}}(x)$ = $\square$ y:Range @ $\textbf{Checkout}$(y); $\textbf{Server}^{\text{succ}}_{\text{unlocked}}(x)$
        $\square$ $\textbf{Update}$(y); $\textbf{Server}^{\text{succ}}_{\text{unlocked}}(x)$
        $\square$ $\textbf{Pre-commit}_{\text{reject}}(y)$; $\textbf{Server}^{\text{succ}}_{\text{unlocked}}(x)$
        $\square$ $\textbf{Post-commit}_{\text{succ}}(x)$; $\textbf{Server}_{\text{idle}}()$

$\textbf{Server}^{\text{fail}}_{\text{unlocked}}(x)$ = $\square$ y:Range @ $\textbf{Checkout}$(y); $\textbf{Server}^{\text{fail}}_{\text{unlocked}}(x)$
        $\square$ $\textbf{Update}$(y); $\textbf{Server}^{\text{fail}}_{\text{unlocked}}(x)$
        $\square$ $\textbf{Pre-commit}_{\text{reject}}(y)$; $\textbf{Server}^{\text{fail}}_{\text{unlocked}}(x)$
        $\square$ $\textbf{Post-commit}_{\text{fail}}(x)$ ; $\textbf{Server}_{\text{idle}}(x)$

$\textbf{Client}_{\text{checkout}}(x)$ = checkout.$x$ $\rightarrow$ (success.$x$ $\rightarrow$ $\textbf{Client}_{\text{idle}}(x)$$\square$accessDenied.$x$ $\rightarrow$ failure.$x$ $\rightarrow$ $\textbf{Client}_{\text{checkout}}(x)$)
$\textbf{Client}_{\text{update}}(x)$ = update.$x$ $\rightarrow$ (success.$x$ $\rightarrow$ $\textbf{Client}_{\text{idle}}(x)$
        $\square$$\textbf{AccessDenied}(x)$; $\textbf{Client}_{\text{update}}(x)$)
        $\square$upToDate.$x$ $\rightarrow$ success.$x$ $\rightarrow$ $\textbf{Client}_{\text{idle}}(x)$)

$\textbf{Client}_{\text{commit}}(x)$ = commit.$x$ $\rightarrow$ (success.$x$ $\rightarrow$ $\textbf{Client}_{\text{idle}}(x)$
        $\square$accessDenied.$x$ $\rightarrow$ failure.$x$ $\rightarrow$ $\textbf{Client}_{\text{idle}}(x)$)
        $\square$needToUpdate.$x$ $\rightarrow$ failure.$x$ $\rightarrow$ $\textbf{Client}_{\text{update}}(x)$)
        $\square$policyViolated.$x$ $\rightarrow$ failure.$x$ $\rightarrow$ $\textbf{Client}_{\text{update}}(x)$)
        $\square$otherCommitUnderExecution.$x$ $\rightarrow$ failure.$x$ $\rightarrow$ $\textbf{Client}_{\text{update}}(x)$)

$\textbf{Client}_{\text{idle}}(x)$ = $\textbf{Client}_{\text{commit}}(x)$$\square$$\textbf{Client}_{\text{update}}(x)$

$\textit{SyncEvents}$ = {|commit,update,checkout,accessDenied,policyViolated,needToUpdate,failure,success|}

$\textbf{Server}$ = $\textbf{Server}_{\text{idle}}()$

$\textbf{Clients}$ = ||| y : Range @ $\textbf{Client}_{\text{checkout}}(y)$
$\textbf{Collaboration}$ = $\textbf{Server}$ $||_{\textit{SyncEvents}}$ $\textbf{Clients}$
```