

Enforcing fine-grained access control for secure collaborative modelling using bidirectional transformations

Csaba Debrececi^{1,2}  · Gábor Bergmann^{1,2} · István Ráth¹ · Dániel Varró^{1,2,3}

Received: 18 March 2017 / Revised: 27 September 2017 / Accepted: 5 November 2017
© The Author(s) 2017. This article is an open access publication

Abstract Large-scale model-driven system engineering projects are carried out collaboratively. Engineering artefacts 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 artefacts need to be protected in accordance with access control policies. We propose a secure collaborative modelling 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 modelling · Secured views · Access control · Online collaboration · Offline collaboration · Bidirectional model transformation

1 Introduction

1.1 Collaborative modelling 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 artefacts (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 artefacts to subcontractors in an architecture-driven supply chain.

Collaboration scenarios include traditional *offline collaborations* with asynchronous long transactions (i.e. to check out an artefact from a version control system and commit local changes afterwards) as well as *online collaborations*

Communicated by Dr. Jörg Kienzle and Alexander Pretschner.

The original version of this article was revised due to a retrospective Open Access order.

✉ Csaba Debrececi
debrececi@mit.bme.hu

Gábor Bergmann
bergmann@mit.bme.hu

István Ráth
rath@mit.bme.hu

Dániel Varró
varro@mit.bme.hu

¹ Department of Measurement and Information Systems, Budapest University of Technology and Economics, Magyar tudósok krt. 2, Budapest 1117, Hungary

² MTA-BME Lendület Research Group on Cyber-Physical Systems, Budapest, Hungary

³ Department of Electrical and Computer Engineering, McGill University, Montreal, Canada

with short and synchronous transactions (e.g. when a group of collaborators simultaneously edit a model, similarly to well-known online document/spreadsheet editors). Several collaborative modelling frameworks (like CDO [49], EMF-Store [50]) 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 [12] or SAVI [1]) address certain collaborative aspects of the design process (e.g. by assuming multiple sub-contractors), 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 modelling artefacts.

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 [2], CVS [24]). *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 Provider₁* and *HW Supplier₁* having full control over their model (fragment). Now if *HW Supplier₁* intends to share part of their model with *SW Provider₁*, 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 Provider₂*, the same argument applies; in the end, a collaboratively developed system model would end up being split into several fragments.

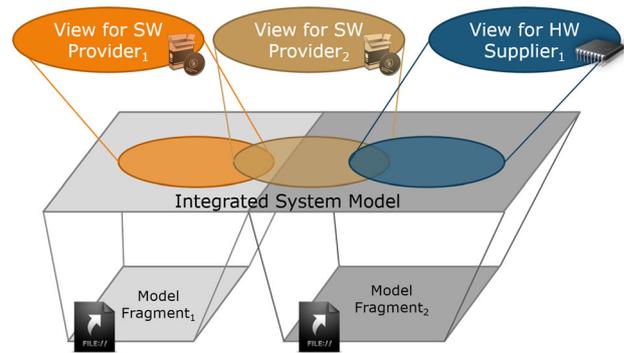


Fig. 1 Problem with file-level access control

Even in the simple case depicted in Fig. 1, the model needs to be split into two files (*Model Fragment₁* and *Model Fragment₂*) and access needs to be granted separately for each file when a *SW Provider₁* and a *HW Supplier₁* collaborates. When a new collaborator, *SW Provider₂*, 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 Provider₁* \Leftrightarrow *SW Provider₂*, one accessible to *HW Supplier₂* \Leftrightarrow *SW Provider₂* 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.

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 modelling 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 Sect. 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 modelling* 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 [30]. 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 [8, 18] 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 collabora-

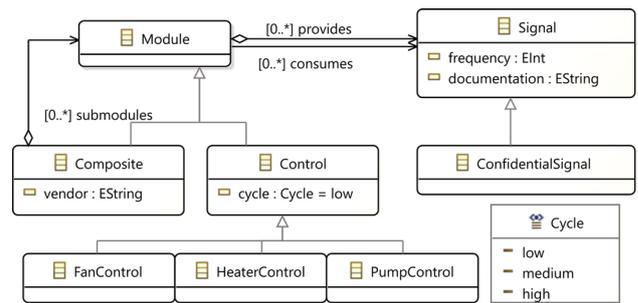


Fig. 2 Simplified metamodel of wind turbine controllers

rative 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 Sect. 2. Section 3 defines how models can be decomposed into individual assets and introduces the rules that assign read and write permissions to assets. In Sect. 4, we overview our bidirectional model transformation for access control, while Sect. 5 describes our secure collaboration schema and proves its correctness. In Sect. 6, we give a brief overview on how to adapt this collaborative modelling schema to online and offline scenarios. Section 7 describes the evaluation of our approach, and related work is overviewed in Sect. 8. Finally, Sect. 9 concludes our paper.

2 Case study

2.1 Modelling language

Our approach will be illustrated using a simplified version of a modelling language for system integrators of offshore wind turbine controllers, which served as one of the case studies of the MONDO EU FP7 project [4]. 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 respon-

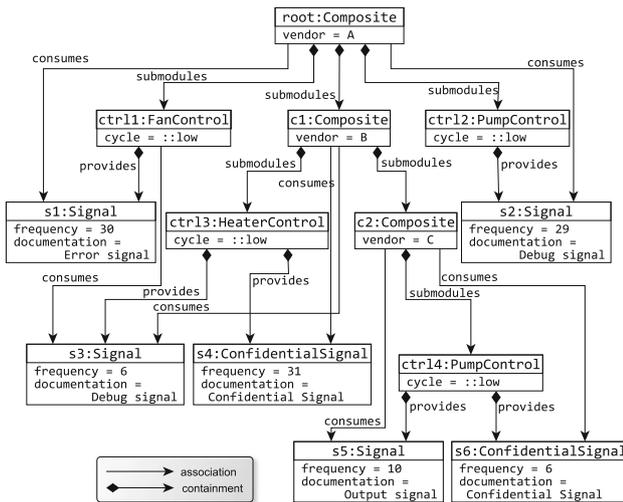


Fig. 3 Sample wind turbine instance model

sibilities. 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 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 in 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 Sect. 1.3, we identify the following challenges.

C1 *Fine-grained Access Control of Model Artefacts.*

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 offline 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 modelling tools.

C2.2 *Offline Models.*

To meet **G2** in an offline 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 online 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 fulfilment.

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 Modelling preliminaries

In order to tackle challenges **C1.1** and **C2.1**, we first analyse 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;

$$\text{ObjectAsset} = \langle \text{object}, \text{type} \rangle$$

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;

$$\text{ReferenceAsset} = \langle \text{object}_s, \text{reference}, \text{object}_t \rangle$$

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

$$\text{AttributeAsset} = \langle \text{object}, \text{attribute}, \text{value} \rangle$$

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

$$M = \langle \{ \text{ObjectAsset} \}, \{ \text{ReferenceAsset} \}, \{ \text{AttributeAsset} \} \rangle$$

Note that there can be multi-valued attributes and references in certain modelling 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 modelling 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 modelling 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 modelling 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. (Modelling 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.

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 types of constraints are often described using OCL [40]. The difference between the two concepts is that violating the lat-

ter kind does not prevent a model from being processed and stored in a given modelling 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” [48]. 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 [33].

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 describe 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 *judgements* (allow, obfuscate, deny) to specify the privileges of a certain *user* $\in Users$ for an *oper-*

ation type (read or write) on a given subset of *assets*.

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

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

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

Definition 7 An *access control policy* defines an effective permission function ($permission_{Eff}$) derived from a list of *access control rules* that applies *judgements* (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. Hence, the effective permission function ($permission_{Eff}$) needs to derive a consistent and conflict-free set of judgements. Our previous work [9, 18] 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 override 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 modelling 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 modelling 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 Sect. 2.2. A possible permission function ($permission_{Eff}$) is visualized in Fig. 4. For instance, *Pump Control Engineers* 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 (ctrl12, ctrl14); 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

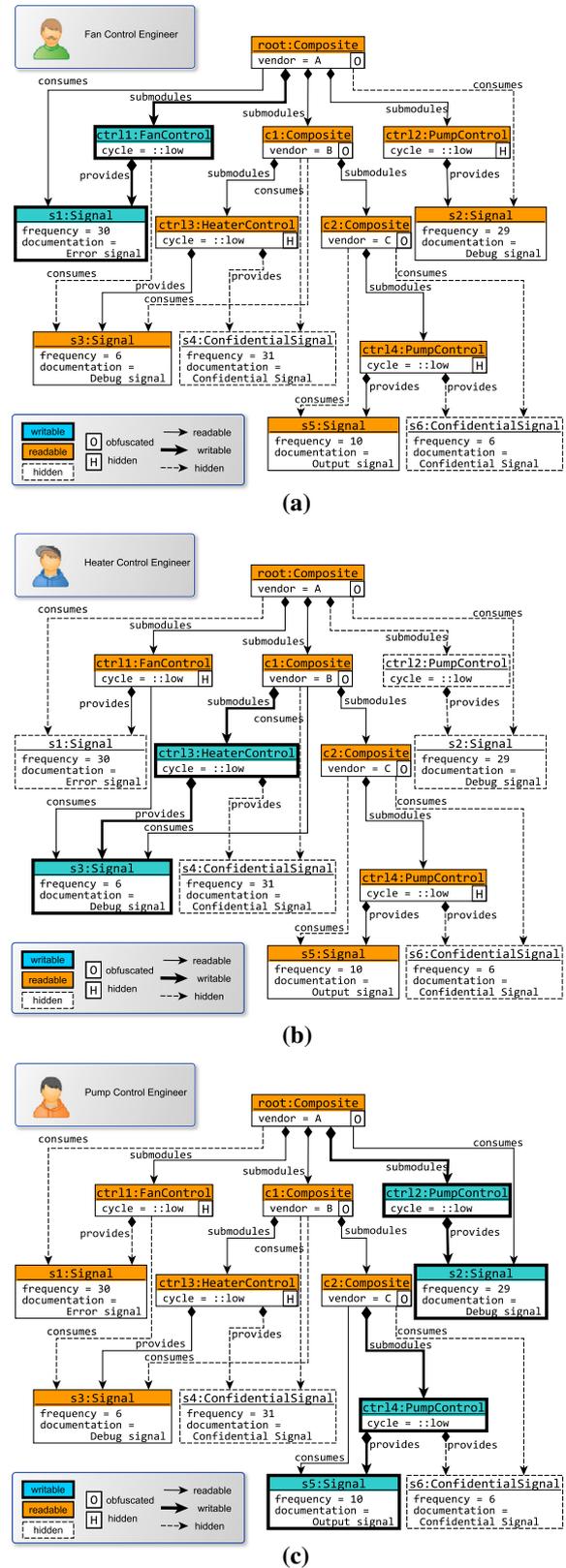


Fig. 4 Effective permissions of the example, a fan control engineer, b heater control engineer, c pump control engineer

and write operation performed by the user. However, challenge C2.1 requires users to access their front models using standard modelling 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*.

In the literature of bidirectional transformations [21], 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.

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

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.

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

The lens concept is illustrated in 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

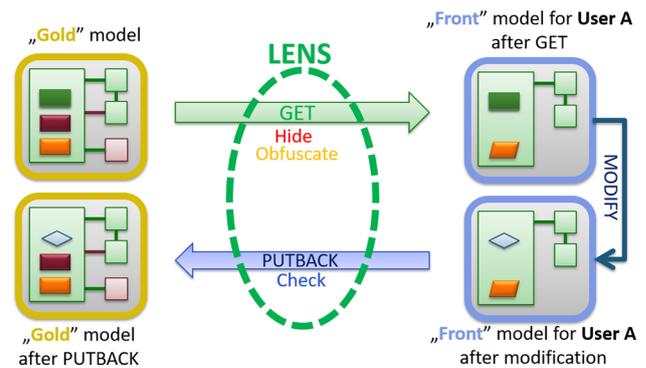


Fig. 5 Secure access control by bidirectional lenses

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 need 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.

4.2 Transformation design

Both GET and PUTBACK are designed as rule-based model transformations [16]. 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 [7] follow an *event-driven behaviour* 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 [16] 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, and 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.

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

Definition 11 A transformation \mathcal{T} consists of a set of transformation rules ($\{rule_1, rule_2 \dots rule_n\}$) 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 .

$$\mathcal{T} = \{rule_1, rule_2 \dots rule_n\}$$

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

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}_{GET} and $\mathcal{T}_{PUTBACK}$ 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 Sect. 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:

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

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

(*rule*_{ADDITIVE GET OBJECT})

The *additive* rule of \mathcal{T}_{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 *ObjectAsset*(o_G, t) and *ObjectAsset*(o_F, t) as follows: an *ObjectAsset*(o_G, t) in the gold model that has no corresponding *ObjectAsset*(o_F, t) in the front model, but it should be readable according to the *permission*_{EFF}. The action part will create a new *ObjectAsset*(o_F, t) and establish a correspondence relation between these two objects.

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 *ctrl3*. This change

Table 1 Additive GET and Subtractive PUTBACK rules

rule	Additive GET Object	Priority	4
Precondition	$(M_G(OA_G, RA_G, AA_G), M_F(OA_F, RA_F, AA_F), permission_{\text{Eff}}) \rightarrow \{ObjectAsset\}^2$ $\{ObjectAsset(o_G, t), ObjectAsset(o_F, t') \mid ObjectAsset(o_G, t) \in OA_G, permission_{\text{Eff}}(ObjectAsset(o_G, t), read) \neq deny,$ $\nexists ObjectAsset(o_F, t') \in OA_F : trace(ObjectAsset(o_G, t)) = ObjectAsset(o_F, t'), t = t'\}$		
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	$(M_G(OA_G, RA_G, AA_G), M_F(OA_F, RA_F, AA_F), permission_{\text{Eff}}) \rightarrow \{AttributeAsset\}$ $\{AttributeAsset(o_F, attr, v') \mid AttributeAsset(o_G, attr, v) \in AA_G, permission_{\text{Eff}}(AttributeAsset(o_G, attr, v), read) \neq deny,$ $\exists ObjectAsset(o_F, t) : ObjectAsset(o_F, t) \in AA_F, trace(ObjectAsset(o_G, t)) = ObjectAsset(o_F, t), \nexists AttributeAsset(o_F, attr, v') :$ $v' = \begin{cases} v, permission_{\text{Eff}}(AttributeAsset(o_G, attr, v), read) = allow \\ obf(v), permission_{\text{Eff}}(AttributeAsset(o_G, attr, v), read) = obfuscate \end{cases}, AttributeAsset(o_F, attr, v') \in AA_F\}$		
Action	$AA_F := AA_F \cup \{AttributeAsset(o_F, attr, v')\}$		
rule	Subtractive PUTBACK Object	Priority	3
Precondition	$(M_G(OA_G, RA_G, AA_G), M_F(OA_F, RA_F, AA_F), permission_{\text{Eff}}) \rightarrow \{ObjectAsset\}$ $\{ObjectAsset(o_G, t) \mid ObjectAsset(o_G, t) \in OA_G, permission_{\text{Eff}}(ObjectAsset(o_G, t), read) \neq deny,$ $\nexists ObjectAsset(o_F, t') : trace(ObjectAsset(o_G, t)) = ObjectAsset(o_F, t'), t = t'\}$		
Action	If $permission_{\text{Eff}}(ObjectAsset(o_G, type), write) \neq deny$ 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 \times_{Δ}		

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}_{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 $rule_{\text{ADDITIVE GET OBJECT}}$ selects the $ObjectAsset(sN_G, \text{Signal})$ as it has no corresponding $ObjectAsset(sN_F, \text{Signal})$ in the front model. The action part creates $ObjectAsset(sN_F, \text{Signal})$ and traces it back to $ObjectAsset(sN_G, \text{Signal})$. Exactly the same sequence happens in case of the front model of *Fan Control Engineer* M_{fan_F} (depicted in Fig. 4a).¹

ADDITIVE GET ATTRIBUTE RULE

($rule_{\text{ADDITIVE GET ATTRIBUTE}}$)

The *additive* rule of \mathcal{T}_{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 $rule_{\text{ADDITIVE GET ATTRIBUTE}}$ selects $AttributeAsset(o_G, attr, v)$ in the gold model that has no corresponding $AttributeAsset(o_F, attr, v')$ in the front model, but it should be readable according to the $permission_{\text{Eff}}$. The value of v' is calculated in accordance with its read permission (potentially in an obfuscated form).

¹ $rule_{\text{ADDITIVE GET REFERENCE}}$ takes care of the containment reference between sN_G and $ctrl3$.

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}_{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 $rule_{\text{ADDITIVE GET ATTRIBUTE}}$ selects the $s1_F$ object from the front model attribute *frequency* and value 15 as $AttributeAsset(s1_F, frequency, 15)$ does not exist, but it should be readable in F_{pump} . The action part adds the $AttributeAsset(s1_F, frequency, 15)$ to M_F^{pump} .²
- 2) \mathcal{T}_{GET} will be executed between M_G and the front model of *Fan Control Engineer* M_F^{fan} (depicted in Fig. 4a). But now, the precondition has no match as $s1$ is not readable in M_F^{fan} .

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

² Similarly, $rule_{\text{SUBTRACTIVE GET ATTRIBUTE}}$ will handle the removal of the previous attribute asset $AttributeAsset(s1_F, frequency, 30)$ before the addition.

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, $\mathcal{T}_{PUTBACK}$ 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 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

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

First, in Sect. 4.3.1, we state the properties that the lens transformations are expected to exhibit. In Sect. 4.3.2, we state and discuss an important assumption that will be vital to proving the aforementioned properties in “Appendix C”. Finally, in Sect. 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 (Properties 1, 2) are generally expected of most rule-based model transformations, in order to define an actual deterministic transformation function. Then Properties 3 and 4 state the security-specific requirements.

Next, Properties 5 and 6 pertain specifically to bidirectional transformations and are widely promoted (see e.g. [21, 46], also [25] 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, Property 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 [9], a possible sample refinement of the write permission levels could be $\{deny < dangle < allow\}$. Cross-references with write permission level *dangle* cannot 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 cannot 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) Property 7 will not hold, as undoability is lost when dangling links/attributes/objects are removed.

4.3.2 Regularity of policy

Constant complement [5], 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, for example, to verify whether PUTBACK (translator in the terminology of [5]) may possibly change the complement (which would violate Property 7). Similarly, correctness proofs can benefit from applying a second kind of partitioning [25] 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 Assumption 1 does not require that the actual effective permissions (as evaluated following the policy by the appropriate algorithms [9, 18]) 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 exist 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 Assumption 1 holds trivially. For PUTBACK, however, it is possible to come up with scenarios where Assumption 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 Sect. 4.3.1, we consider Assumption 1 to hold and apply partitioning-based arguments partly similar to those in [5, 25]. This limitation to the case with regularity is, on one hand, necessary to prove the properties stated earlier (the exception is Property 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 “Appendices A and B”; the proof sketches are found in “Appendix C”.

4.3.3 Realization in EMF

We realized the presented lens transformation in the Eclipse Modelling 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 Sect. 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, and 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 Sect. 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 Sect. 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

Figures 6 and 7 describe the behaviour 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 behaviour of a given state by defining its own state machine where only one state can be active.

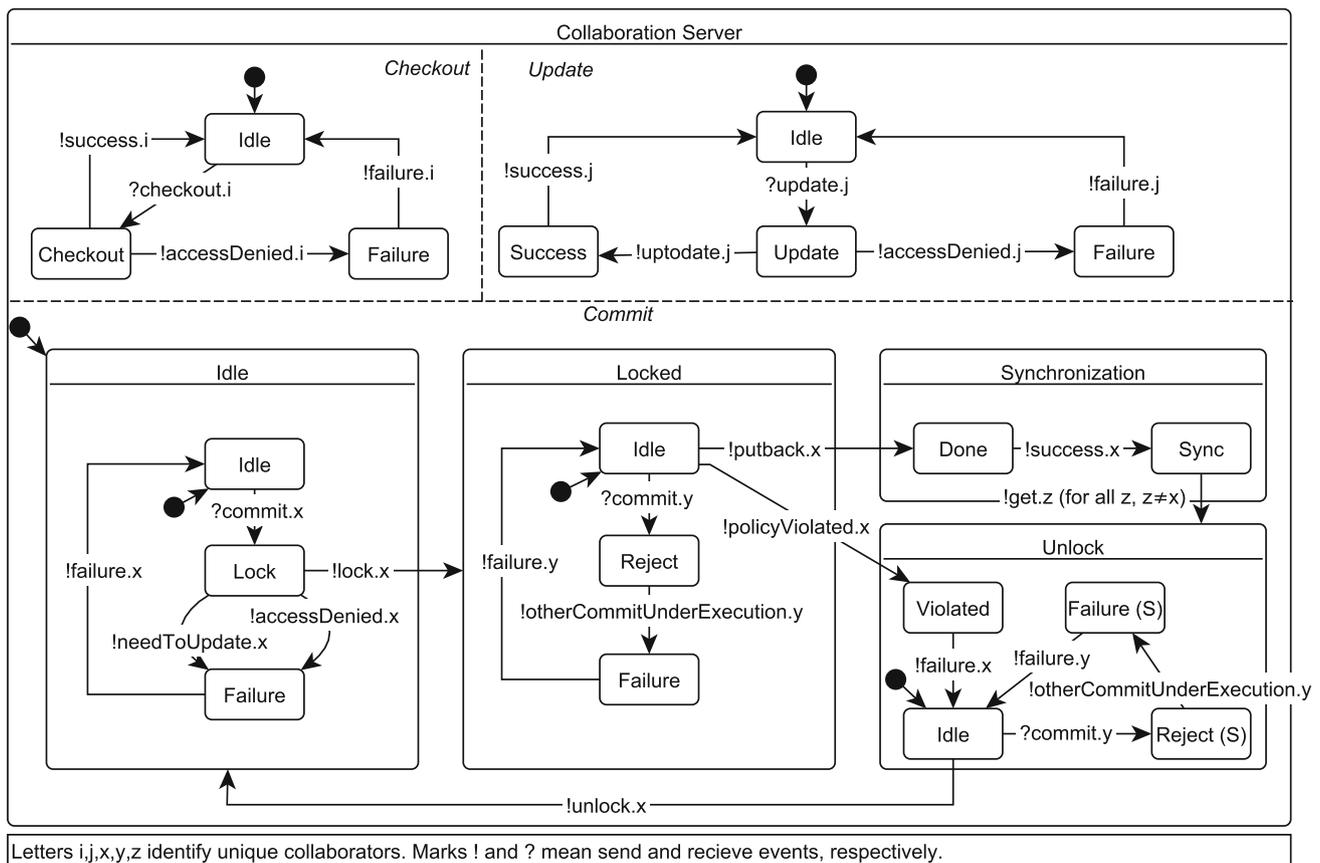


Fig. 6 State machine of the collaboration server

Orthogonal regions (divided by dashed borders) separate the behaviour 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 itself³ by sending an *accessDenied* event followed by a *failure* event. Otherwise, a *success* event is sent.

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

³ 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.

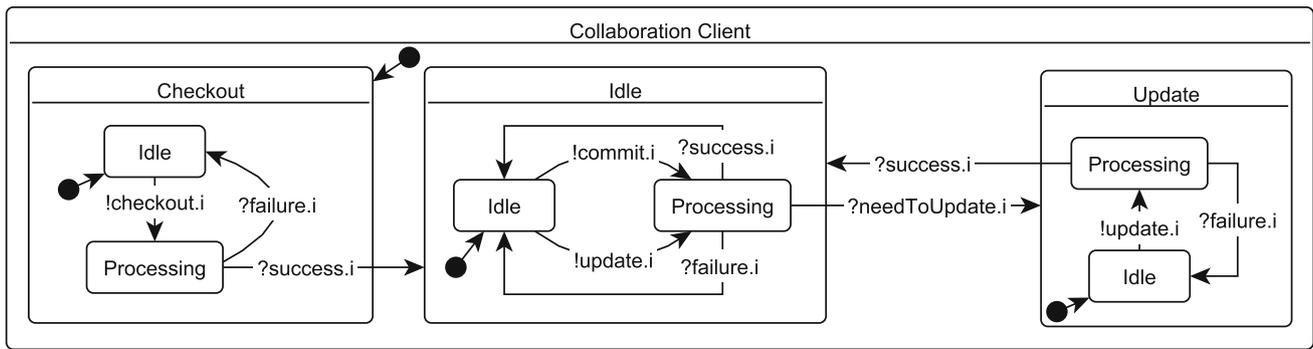


Fig. 7 State machine of the collaboration client

executing T_{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 move 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 behaviour 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)* [44, 45] described in “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 behaviour 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*}.

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

For the analysis, we used the *FDR4 tool* [30] to evaluate *assertions* over certain properties of the processes.

CRITERIA 1. and 2. require 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.

$$\text{assert } \mathbf{Collaboration} : [deadlock\ free] \quad (1)$$

$$\text{assert } \mathbf{Collaboration} : [divergence\ free] \quad (2)$$

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

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

$\mathbf{P} \mathbb{F} : \langle a, b, c \rangle$ 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 \setminus 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. $\mathbf{P} \setminus (\mathcal{E} \cap \{a, b, c\})$ 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.

$$\text{assert } \mathbf{Server} \setminus (\mathcal{E} \cap \{unlock, putback\}) \quad (3)$$

$$\neg \mathbb{T} : \langle putback.x, putback.y \rangle$$

$$x, y \in Int, x \neq y$$

CRITERION 4. requires to execute \mathcal{T}_{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 .

$$\text{assert } \mathbf{Server} \setminus (\mathcal{E} \cap \{get.n, putback, unlock\}) \quad (4)$$

$$\mathbb{T} : \langle putback.x, get.n, unlock.x \rangle$$

$$x, N \in Int, x \neq n$$

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:

Table 2 Results of the assertion in *FDR4 tool*

Assertions	States	Transitions	time (s)
1	89,233	227,591	0.44
2	89,233	227,591	0.51
3	452	1121	0.13
4	417	1071	0.42
5	10	11	0.12
6	10	11	0.12

$$\text{assert } \mathbf{Clients} \setminus (\mathcal{E} \cap \{commit, update, needToUpdate\}) \quad (5)$$

$$\neg \mathbb{T} : \langle commit.x, needToUpdate.x, commit.x \rangle$$

$$\text{assert } \mathbf{Clients} \setminus (\mathcal{E} \cap \{commit, update, needToUpdate\}) \quad (6)$$

$$\mathbb{F} : \langle commit.x, needToUpdate.x, update.x \rangle$$

$$x \in Int$$

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 assertions⁴ 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 90,000 states and 230,000 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 Scheme⁵

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 Sect. 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

⁴ The complete formal specification is available at: <http://goo.gl/pJzIX1>.

⁵ Source codes and more details are at <https://tinyurl.com/sosym-access-control-source>.

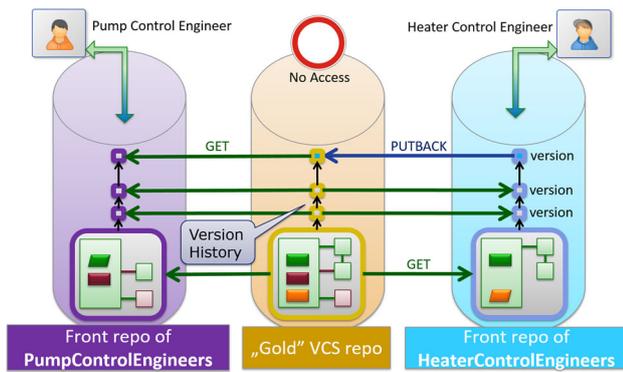


Fig. 8 The MONDO offline collaboration server: architecture

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 *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 modelling tools. Nevertheless, optional client-side collaboration tools may still be used to improve user experience, e.g. for smart model merging [19], user-friendly lock management [13] 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* [2]. Subversion provides features of the collaboration scheme by default:

FILE-LEVEL ACCESS CONTROL

is responsible for sending *accessDenied* event to the 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}_{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.

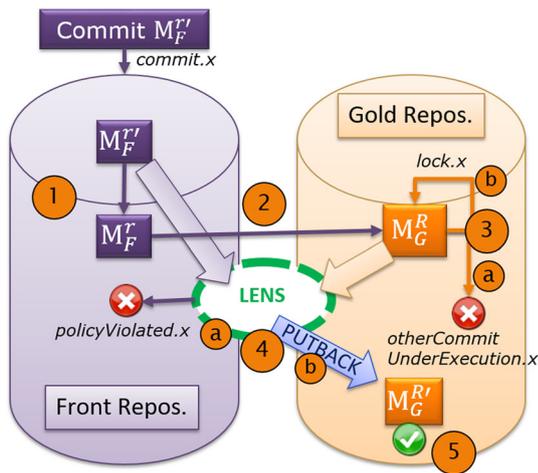


Fig. 9 Pre-commit hook at the front repository

- (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.x* 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).
 5. Finally, the hook finishes successfully and let the VCS server to handle the request.

Post-Commit Hook. \mathcal{T}_{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.

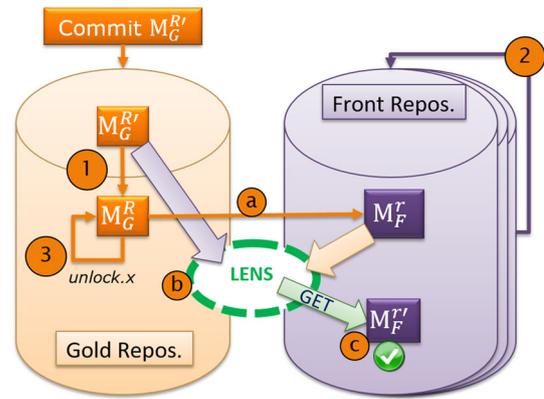


Fig. 10 Post-commit hook at the gold repository

- (b) \mathcal{T}_{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 committed 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.x*.
 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 modelling platforms (e.g. IFC [32]) automatically provide such permanent identifiers.

While requiring permanent identifiers is a limitation of the approach, it is only relevant for modelling 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 modelling 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 modelling tool where changes are propagated immediately to other users during *collaborative modelling 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.

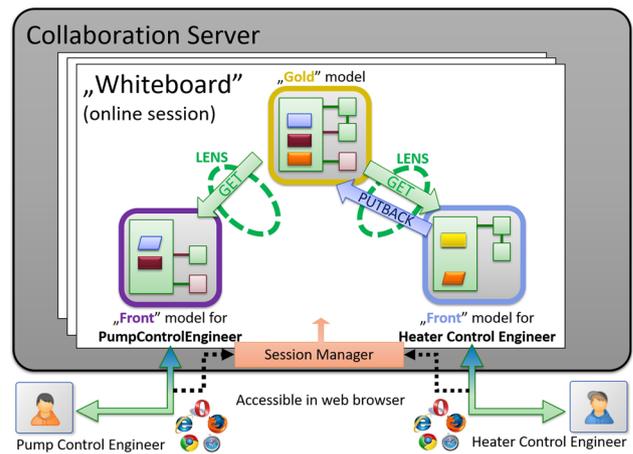


Fig. 11 Overview of online collaboration

6.2.1 Technical realization

During a collaborative modelling 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* [16]; thus, users always see an up-to-date view of the model during the editing session.

Similarly to modern collaborative editing tools (such as Google Sheets [15]), 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 disc.

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}_{GET} that achieves the front model on which the new user can work.

- 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.
- 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 arises, 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 metamodels⁶ which also provides similar modelling 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 GenMy-Model [3, 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}_{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).

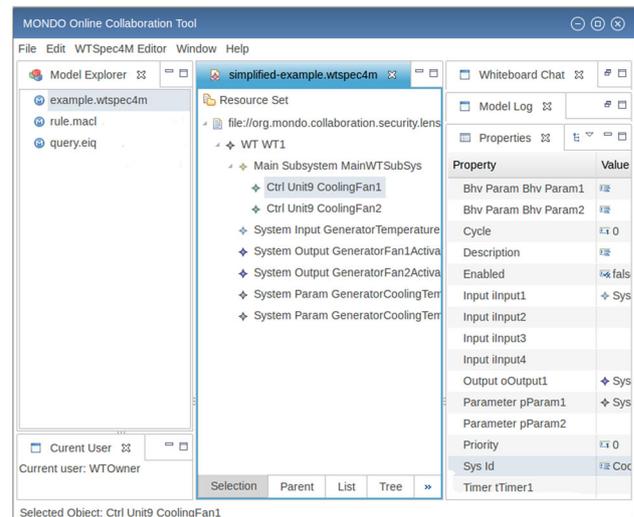


Fig. 12 User interface for online collaboration prototype

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.

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 besides 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 modelling tools may even incorporate this information into their model notation, e.g. to visually show read-only parts of the model as frozen.

⁶ EMF and RAP integration: https://wiki.eclipse.org/RAP/EMF_Integration.

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 [13].

6.3.2 Ordered lists

In EMF, some multi-valued references and attributes are *ordered* lists. Model assets introduced in Sect. 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 Property 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, for example, 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 [21] 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 [4] 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*?

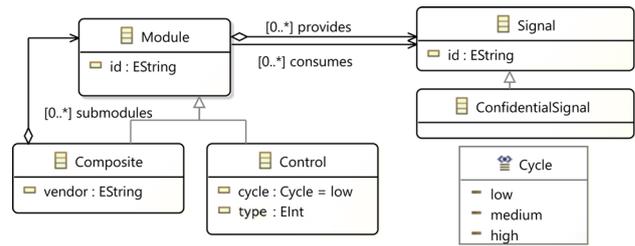


Fig. 13 Modified metamodel of wind turbine

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, Sect. 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 Sect. 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` and `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 $2M \geq K$.

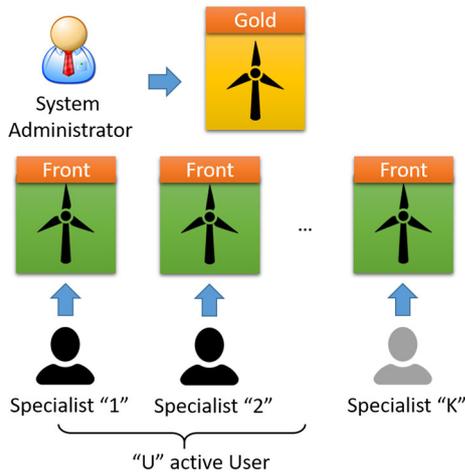


Fig. 14 Users and active user in the measurement setup

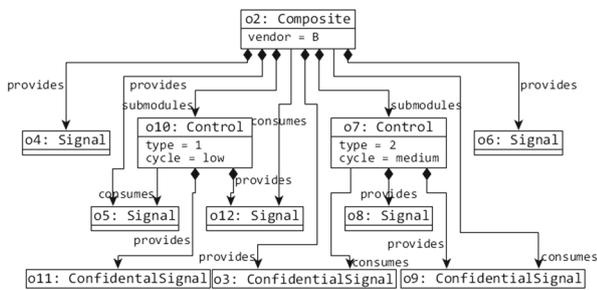


Fig. 15 Core structure of synthesized models

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 behaviour 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 a rearrangement of the containment hierarchy; (b) it does not change the size of the model and 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

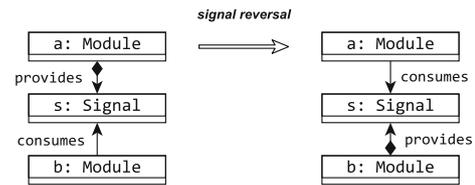


Fig. 16 Signal reversal operation

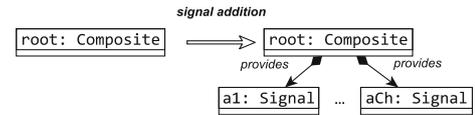


Fig. 17 Signal addition operation

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 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 *propagation time* to propagate 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.

Hardware Configuration All the measurements⁷ executed on a personal computer⁸ 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, 10,500 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),

⁷ Raw data and reproduction instructions at <https://tinyurl.com/sosym-access-control>.

⁸ CPU: Intel Core i7-4700MQ@2.40GHz, MEM: 8GB.

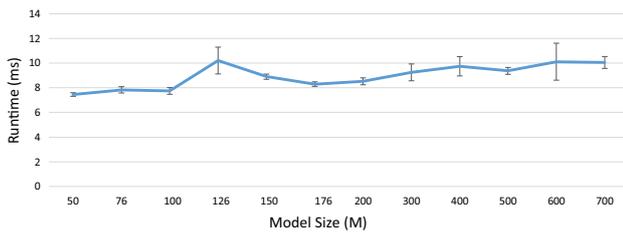


Fig. 18 Average execution time of an online signal reversal (increasing model size)

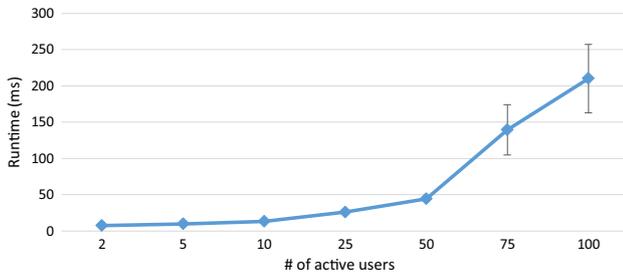


Fig. 19 Average execution time of an online signal reversal (increasing the number of active users)

- 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 Sect. 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.

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 $M = 6000$ (34,001 objects, 45,000 references).

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, 12,000 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 PUT-BACK 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 colour) 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 grows linearly with the size of the model, while synchronization is nonlinear 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

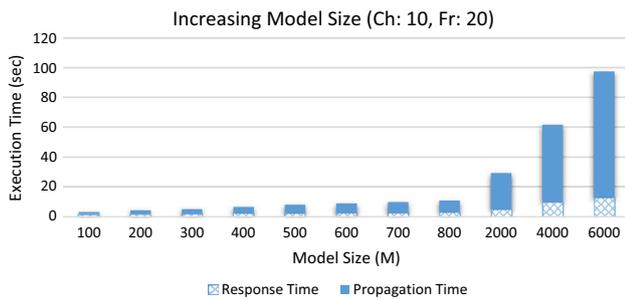


Fig. 20 Average execution time of an offline signal addition (increasing model size)

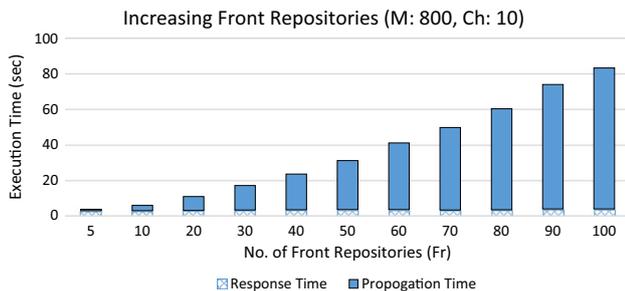


Fig. 21 Average execution time of an offline signal addition (increasing number of front repositories)

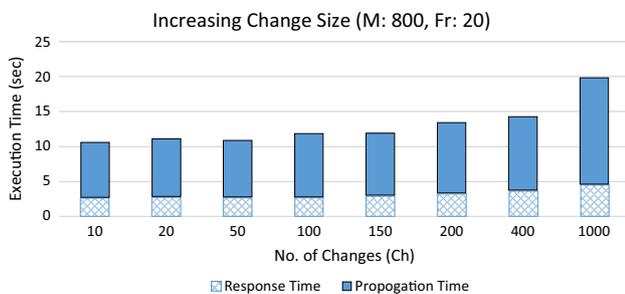


Fig. 22 Average execution time of an offline signal addition (increasing number of changes)

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 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 trade-off, 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 labelled 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 [18] 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 [29], 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 [26] can be controlled on the database or catalogue 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 [20] 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 Sect. 3.1), and we also provide offline collaboration.

In the Oracle Database Semantic Technologies [41], 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 [41] and are generalized into abstract tokens and operators in [42].

8.1.3 Access control for XML documents

A number of standards such as XACML [31] (OASIS standard) provide fine-grained access control for XML documents. These types of documents are similar to models in a way that they consist of nodes with attributes that may contain other nodes. XACML provides several combining algorithms to select from contradicting policies. In [27], 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 [38], 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 Sect. 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 modelling environments

Currently, fine-grained access control is not considered in the state-of-the-art tools of MDE such as MetaEdit+[53], VirtualEMF [14], WebGME [36], EMFStore [50], GenMy-Model [3], Obeo Designer Team [39], MDEForge [6] or the tools developed according to [28]. See also the broader survey in [43].

The generic framework CDO [49] (used, for example, in [39]) 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 Vehicle-FORGE stores their model in graph-based databases and has an access control scheme TrustForge [17] that uses an implementation of KeyNote [10] 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 [47] 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 [23].

8.2 Access control enforcement

8.2.1 Access control using model transformation

Closest to our approach, [37] 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 [34], 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 [11], 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 [35] 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, [25] 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 modelling 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 Sect. 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.

Acknowledgements 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 (LP-2015-12) and NSERC RGPIN-04573-16. The second author was supported by the János Bolyai Research Scholarship of the Hungarian Academy of Sciences.

Open Access This article is distributed under the terms of the Creative Commons Attribution 4.0 International License (<http://creativecommons.org/licenses/by/4.0/>), which permits unrestricted use, distribution, and reproduction in any medium, provided you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license, and indicate if changes were made.

A GET Transformation rules

<i>rule</i>	Subtractive GET Reference	Priority	1
-------------	---------------------------	-----------------	---

Precondition :: $(M_G(OA_G, RA_G, AA_G), M_F(OA_F, RA_F, AA_F), permission_{\text{Efl}}) \rightarrow \{ReferenceAsset\}$

$\{ReferenceAsset(s_F, ref, t_F)\} | ReferenceAsset(s_G, ref, t_G) \in RA_F,$

$\exists ObjectAsset(s_F, t_S) \in OA_F, ObjectAsset(t_F, t_T) \in OA_F : trace(ObjectAsset(s_G, t_S)) = ObjectAsset(s_F, t_S),$

$trace(ObjectAsset(t_G, t_t)) = ObjectAsset(t_F, t_t), \nexists ReferenceAsset(s_G, ref, t_G) : ReferenceAsset(s_G, ref, t_G) \in RA_G$

$permission_{\text{Efl}}(ReferenceAsset(s_G, ref, t_G), read) \neq deny\}$

Action

$RA_F := RA_F \setminus \{ReferenceAsset(s_F, ref, t_F)\}$

<i>rule</i>	Subtractive GET Attribute	Priority	2
-------------	---------------------------	-----------------	---

Precondition :: $(M_G(OA_G, RA_G, AA_G), M_F(OA_F, RA_F, AA_F), permission_{\text{Efl}}) \rightarrow \{AttributeAsset\}$

$\{AttributeAsset(o_F, attr, v')\} | AttributeAsset(o_F, attr, v') \in AA_F, \exists ObjectAsset(o_F, t) : ObjectAsset(o_F, t) \in OA_F$

$\exists ObjectAsset(o_G, t) : ObjectAsset(o_G, t) \in OA_G, trace(ObjectAsset(o_G, t)) = ObjectAsset(o_F, t), \nexists AttributeAsset(o_G, attr, v) :$

$v' = \begin{cases} v, permission_{\text{Efl}}(AttributeAsset(o_G, attr, v), read) = allow \\ obf(v), permission_{\text{Efl}}(AttributeAsset(o_G, attr, v), read) = obfuscate \end{cases}, AttributeAsset(o_G, attr, v) \in AA_G\}$

Action

$AA_F := AA_F \setminus \{AttributeAsset(o_F, attr, v')\}$

<i>rule</i>	Subtractive GET Object	Priority	3
-------------	------------------------	-----------------	---

Precondition :: $(M_G(OA_G, RA_G, AA_G), M_F(OA_F, RA_F, AA_F), permission_{\text{Efl}}) \rightarrow \{ObjectAsset\}$

$\{ObjectAsset(o_F, t)\} | ObjectAsset(o_F, t) \in OA_F, \nexists ObjectAsset(o_G, t') : trace(ObjectAsset(o_G, t')) = ObjectAsset(o_F, t), t = t'$

$permission_{\text{Efl}}(ObjectAsset(o_G, t'), read) \neq deny\}$

Action

$OA_F := OA_F \setminus \{ObjectAsset(o_F, t)\}, trace \setminus ObjectAsset(o_G, t') | trace(ObjectAsset(o_G, t')) = ObjectAsset(o_F, t)$

<i>rule</i>	Additive GET Object	Priority	4
-------------	---------------------	-----------------	---

Precondition :: $(M_G(OA_G, RA_G, AA_G), M_F(OA_F, RA_F, AA_F), permission_{\text{Efl}}) \rightarrow \{ObjectAsset\}^2$

$\{ObjectAsset(o_G, t), ObjectAsset(o_F, t')\} | ObjectAsset(o_G, t) \in OA_G, permission_{\text{Efl}}(ObjectAsset(o_G, t), read) \neq deny,$

$\nexists ObjectAsset(o_F, t') \in OA_F : trace(ObjectAsset(o_G, t)) = ObjectAsset(o_F, t'), t = t'\}$

Action

$OA_F := OA_F \cup \{ObjectAsset(o_F, t')\}, trace(ObjectAsset(o_G, t)) := ObjectAsset(o_F, t')$

<i>rule</i>	Additive GET Attribute	Priority	5
-------------	------------------------	-----------------	---

Precondition :: $(M_G(OA_G, RA_G, AA_G), M_F(OA_F, RA_F, AA_F), permission_{\text{Efl}}) \rightarrow \{AttributeAsset\}$

$\{AttributeAsset(o_F, attr, v')\} | AttributeAsset(o_G, attr, v) \in AA_G, permission_{\text{Efl}}(AttributeAsset(o_G, attr, v), read) \neq deny,$

$\exists ObjectAsset(o_F, t) : ObjectAsset(o_F, t) \in AA_F, trace(ObjectAsset(o_G, t)) = ObjectAsset(o_F, t), \nexists AttributeAsset(o_F, attr, v') :$

$v' = \begin{cases} v, permission_{\text{Efl}}(AttributeAsset(o_G, attr, v), read) = allow \\ obf(v), permission_{\text{Efl}}(AttributeAsset(o_G, attr, v), read) = obfuscate \end{cases}, AttributeAsset(o_F, attr, v') \in AA_F\}$

Action

$AA_F := AA_F \cup \{AttributeAsset(o_F, attr, v')\}$

<i>rule</i>	Additive GET Reference	Priority	6
-------------	------------------------	-----------------	---

Precondition :: $(M_G(OA_G, RA_G, AA_G), M_F(OA_F, RA_F, AA_F), permission_{\text{Efl}}) \rightarrow \{ReferenceAsset\}$

$\{ReferenceAsset(s_F, ref, t_F)\} | ReferenceAsset(s_G, ref, t_G) \in RA_G, permission_{\text{Efl}}(ReferenceAsset(s_G, ref, t_G), read) \neq deny,$

$\exists ObjectAsset(s_G, t_S), ObjectAsset(t_G, t_t) : trace(ObjectAsset(s_G, t_S)) = ObjectAsset(s_F, t_S),$

$trace(ObjectAsset(t_G, t_t)) = ObjectAsset(t_F, t_t), \nexists ReferenceAsset(s_F, ref, t_F) : ReferenceAsset(s_F, ref, t_F) \in RA_F\}$

Action

$RA_F := RA_F \cup \{ReferenceAsset(s_F, ref, t_F)\}$

B PUTBACK Transformation rules

rule	Subtractive PUTBACK Reference	Priority	1
Precondition	$(M_G(OA_G, RA_G, AA_G), M_F(OA_F, RA_F, AA_F), permission_{\text{Eff}}) \rightarrow \{ReferenceAsset\}$ $\{ReferenceAsset(s_G, ref, t_G) ReferenceAsset(s_G, ref, t_G) \in RA_G, permission_{\text{Eff}}(ReferenceAsset(s_G, ref, t_G), read) \neq deny,$ $\exists ObjectAsset(s_G, t_s), ObjectAsset(t_G, t_t) : trace(ObjectAsset(s_G, t_s)) = ObjectAsset(s_F, t_s),$ $trace(ObjectAsset(t_G, t_t)) = ObjectAsset(t_F, t_t), \nexists ReferenceAsset(s_F, ref, t_F) : ReferenceAsset(s_F, ref, t_F) \in RA_F\}$		
Action	If $permission_{\text{Eff}}(ReferenceAsset(s_G, ref, t_G), write) \neq deny$ then $RA_G := RA_G \setminus ReferenceAsset(s_G, ref, t_G)$ else \times		
rule	Subtractive PUTBACK Attribute	Priority	2
Precondition	$(M_G(OA_G, RA_G, AA_G), M_F(OA_F, RA_F, AA_F), permission_{\text{Eff}}) \rightarrow \{AttributeAsset\}$ $\{AttributeAsset(o_G, attr, v) \nexists AttributeAsset(o_F, attr, v') :$ $\exists ObjectAsset(o_F, t) : ObjectAsset(o_F, t) \in OA_F, trace(ObjectAsset(o_G, t)) = ObjectAsset(o_F, t),$ $v = \begin{cases} v', permission_{\text{Eff}}(AttributeAsset(o_G, attr, v), read) = allow \\ obf^{-1}(v'), permission_{\text{Eff}}(AttributeAsset(o_G, attr, v), read) = obfuscate \end{cases}, AttributeAsset(o_G, attr, v) \in AA_G\}$		
Action	If $permission_{\text{Eff}}(AttributeAsset(o_G, attr, v), write) \neq deny$ then $AA_G := AA_G \setminus AttributeAsset(o_G, attr, v)$ else \times		
rule	Subtractive PUTBACK Object	Priority	3
Precondition	$(M_G(OA_G, RA_G, AA_G), M_F(OA_F, RA_F, AA_F), permission_{\text{Eff}}) \rightarrow \{ObjectAsset\}$ $\{ObjectAsset(o_G, t) ObjectAsset(o_G, t) \in OA_G, permission_{\text{Eff}}(ObjectAsset(o_G, t), read) \neq deny,$ $\nexists ObjectAsset(o_F, t') : trace(ObjectAsset(o_G, t)) = ObjectAsset(o_F, t'), t = t'\}$		
Action	If $permission_{\text{Eff}}(ObjectAsset(o_G, type), write) \neq deny$ then $OA_G := OA_G \setminus ObjectAsset(o_G, t), trace \setminus ObjectAsset(o_G, t) trace(ObjectAsset(o_G, t)) = ObjectAsset(o_F, t)$ else \times		
rule	Additive PUTBACK Object	Priority	4
Precondition	$(M_G(OA_G, RA_G, AA_G), M_F(OA_F, RA_F, AA_F), permission_{\text{Eff}}) \rightarrow \{ObjectAsset\}^2$ $\{ObjectAsset(o_F, t), ObjectAsset(o_G, t) ObjectAsset(o_F, t) \in OA_F, \nexists ObjectAsset(o_G, t) : trace(ObjectAsset(o_G, t)) = ObjectAsset(o_F, t)\}$		
Action	$OA_G := OA_G \cup \{ObjectAsset(o_G, t)\}$ If $permission_{\text{Eff}}(ObjectAsset(o_G, t), write) \neq deny$ then $trace(ObjectAsset(o_G, t)) := ObjectAsset(o_F, t)$ else \times		
rule	Additive PUTBACK Attribute	Priority	5
Precondition	$(M_G(OA_G, RA_G, AA_G), M_F(OA_F, RA_F, AA_F), permission_{\text{Eff}}) \rightarrow \{AttributeAsset\}$ $\{AttributeAsset(o_G, attr, v) AttributeAsset(o_F, attr, v') \in AA_F, \exists ObjectAsset(o_F, t) : ObjectAsset(o_F, t) \in AA_F$ $\exists ObjectAsset(o_G, t) : ObjectAsset(o_G, t) \in OA_G, trace(ObjectAsset(o_G, t)) = ObjectAsset(o_F, t), \nexists AttributeAsset(o_G, attr, v) :$ $v = \begin{cases} v', permission_{\text{Eff}}(AttributeAsset(o_G, attr, v), read) = allow \\ obf^{-1}(v'), permission_{\text{Eff}}(AttributeAsset(o_G, attr, v), read) = obfuscate \end{cases}, AttributeAsset(o_G, attr, v) \in AA_G\}$		
Action	$AA_G := AA_G \cup \{AttributeAsset(o_G, attr, v)\}$ If $(permission_{\text{Eff}}(AttributeAsset(o_G, attr, v), write) = deny)$ then \times		
rule	Additive PUTBACK Reference	Priority	6
Precondition	$(M_G(OA_G, RA_G, AA_G), M_F(OA_F, RA_F, AA_F), permission_{\text{Eff}}) \rightarrow \{ReferenceAsset\}$ $\{ReferenceAsset(s_G, ref, t_G) ReferenceAsset(s_F, ref, t_F) \in RA_F,$ $\exists ObjectAsset(s_F, t_s), ObjectAsset(t_F, t_t) \in OA_F : trace(ObjectAsset(s_G, t_s)) = ObjectAsset(s_F, t_s),$ $trace(ObjectAsset(t_G, t_t)) = ObjectAsset(t_F, t_t), \nexists ReferenceAsset(s_G, ref, t_G) : ReferenceAsset(s_G, ref, t_G) \in RA_G\}$		
Action	$RA_G := RA_G \cup \{ReferenceAsset(s_G, ref, t_G)\}$ If $permission_{\text{Eff}}(ReferenceAsset(s_G, ref, t_G), write) = deny$ then \times		

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

Proof for Property 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 Assumption 1; so the transformations can at least be known to terminate even when the conditions of regularity are not met. \square

Proof for Property 2 Confluence

Here we follow a proof strategy and terminology common in rewriting systems (see e.g. [22] for graph transformations). The proof will rely on both the above result for termination as well as Assumption 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 Assumption 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 complete 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 [22].

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. \square

Proof for Property 3 Confidentiality

The rule-based formalization and the partitioning of assets based on permissions (made possible due to Assumption 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. \square

Proof for Property 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 Property 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 Property 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. \square

Proof for Property 6 PutGet

At the point where a successful PUTBACK terminates, 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 Property 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 Assumption 1, assets can be partitioned into a writeable part and its complement, in line with the *constant complement* [5] 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. \square

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

Checkout(x) = checkout. x → (success. x → SKIP □ accessDenied. x → failure. x → SKIP)
Update(x) = update. x → (success. x → SKIP □ accessDenied. x → failure. x → SKIP □ upToDate. x → failure. x → SKIP)
Pre-commit_{succ}(x) = commit. x → lock. x → SKIP
Pre-commit_{fail}(x) = commit. x → (accessDenied. x → failure. x → SKIP □ needToUpdate. x → failure. x → SKIP)
Pre-commit_{reject}(x) = commit. x → otherCommitUnderExecution. x → SKIP
Commit_{succ}(x) = putback. x → success. x SKIP
Commit_{fail}(x) = policyViolated. x → failure. x → SKIP
Post-commit_{sync}(x, z) = if $x \neq z$ get. x → SKIP else SKIP
Post-commit_{succ}(x) = unlock. x → SKIP
Post-commit_{fail}(x) = unlock. x → SKIP

Server_{idle}() = □ y :Range @ **Checkout**(y); **Server**_{idle}()
 □ **Update**(y); **Server**_{idle}()
 □ **Pre-commit**_{succ}(y); **Server**_{locked}(y)
 □ **Pre-commit**_{fail}(y); **Server**_{idle}()

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

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

Server_{unlocked}^{succ}(x) = □ y :Range @ **Checkout**(y); **Server**_{unlocked}^{succ}(x)
 □ **Update**(y); **Server**_{unlocked}^{succ}(x)
 □ **Pre-commit**_{reject}(y); **Server**_{unlocked}^{succ}(x)
 □ **Post-commit**_{succ}(x); **Server**_{idle}()

Server_{unlocked}^{fail}(x) = □ y :Range @ **Checkout**(y); **Server**_{unlocked}^{fail}(x)
 □ **Update**(y); **Server**_{unlocked}^{fail}(x)
 □ **Pre-commit**_{reject}(y); **Server**_{unlocked}^{fail}(x)
 □ **Post-commit**_{fail}(x); **Server**_{idle}(x)

Client_{checkout}(x) = checkout. x → (success. x → **Client**_{idle}(x) □ accessDenied. x → failure. x → **Client**_{checkout}(x))

Client_{update}(x) = update. x → (success. x → **Client**_{idle}(x)
 □ **AccessDenied**(x); **Client**_{update}(x)
 □ upToDate. x → success. x → **Client**_{idle}(x))

Client_{commit}(x) = commit. x → (success. x → **Client**_{idle}(x)
 □ accessDenied. x → failure. x → **Client**_{idle}(x)
 □ needToUpdate. x → failure. x → **Client**_{update}(x)
 □ policyViolated. x → failure. x → **Client**_{update}(x)
 □ otherCommitUnderExecution. x → failure. x → **Client**_{update}(x))

Client_{idle}(x) = **Client**_{commit}(x) □ **Client**_{update}(x)

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

Server = **Server**_{idle}()

Clients = ||| y : Range @ **Client**_{checkout}(y)

Collaboration = **Server** || _{$SyncEvents$} **Clients**

References

1. Aerospace Vehicle Systems Institute. SAVI Research Project. <http://savi.avsi.aero/>
2. Apache. Subversion. 07 (2017)
3. Axellence. GenMyModel. <http://www.genmymodel.com>
4. Bagnato, A., Brosse, E., Sadovykh, A., Maló, P., Trujillo, S., Mendialdua, X., De Carlos, X.: Flexible and scalable modelling in the mondo project: Industrial case studies. In: XM@ MoDELS, pp. 42–51 (2014)
5. Bancilhon, F., Spyrtos, N.: Update semantics of relational views. *ACM Trans. Database Syst.* **6**(4), 557–575 (1981)
6. Basciani, F., Rocco, J.D., Ruscio, D.D., Salle, A.D., Iovino, L., Pierantonio, A.: MDEForge: an extensible web-based modeling platform. In: CloudMDE@MoDELS (2014)
7. Bergmann, G., Dávid, I., Hegedüs, Á., Horváth, Á., Ráth, I., Ujhelyi, Z., Varró, D.: VIATRA 3: a reactive model transformation platform. In: International Conference on Theory and Practice of Model Transformations, pp. 101–110. Springer (2015)
8. Bergmann, G., Debrececi, C., Ráth, I., Varró, D.: Query-based access control for secure collaborative modeling using bidirectional transformations. In: ACM/IEEE 19th International Conference on MODELS (2016)
9. Bergmann, G., Debrececi, C., Ráth, I., Varró, D.: Towards efficient evaluation of rule-based permissions for fine-grained access control in collaborative modeling. In: 2nd International Workshop on Collaborative Modelling in MDE, Austin Texas, USA. ACM (in press)
10. Blaze, M., Keromytis, A.D.: The keynote trust-management system version 2 (1999)
11. Breu, R., Popp, G., Alam, M.: Model based development of access policies. *Int. J. Softw. Tools Technol. Transf.* **9**(5), 457–470 (2007)
12. CAESAR Research Project. <http://store.sae.org/caesar/>
13. Chechik, M., Dalpiaz, F., Debrececi, C., Horkoff, J., Ráth, I., Salay, R., Varró, D.: Property-based methods for collaborative model development. In: Joint Proceedings of the 3rd International Workshop on the Glob. of Modeling Lang. and the 9th International Workshop on Multi-Paradigm Modeling. Citeseer, pp. 1–7 (2015)
14. Clasen, C., Jouault, F., Cabot, J.: VirtualEMF: A model virtualization tool. In: Advances in Conceptual Modeling. Recent Developments and New Directions, pp. 332–335 (2011)
15. Conner, N.: Google Apps: The Missing Manual: The Missing Manual. O'Reilly Media Inc, Sebastopol (2008)
16. Czarnecki, K., Helsen, S.: Feature-based survey of model transformation approaches. *IBM Syst. J.* **45**(3), 621–645 (2006)
17. DARPA VehicleFORGE, P.U.: TrustForge: Flexible Access Control for VehicleForge.mil Collaborative Environment, (2012)
18. Debrececi, C., Bergmann, G., Ráth, I., Varró, D.: Deriving effective permissions for modeling artifacts from fine-grained access control rules. In: 1st International Workshop on Collaborative Modelling in MDE, Saint Malo, France. ACM (2016)
19. Debrececi, C., Ráth, I., Varró, D., De Carlos, X., Mendialdua, X., Trujillo, S.: Automated model merge by design space exploration. In: International Conference on Fundamental Approaches to Software Engineering. Springer, pp. 104–121 (2016)
20. Dietzold, S., Auer, S., S.: Access control on RDF triple stores from a semantic wiki perspective. In: Scripting for the Semantic Web Workshop at 3rd European Semantic Web Conference (ESWC) (2006)
21. Diskin, Z.: Algebraic models for bidirectional model synchronization. In: MoDELS, pp. 21–36 (2008)
22. Ehrig, H., Ehrig, K., Prange, U., Taentzer, G.: Fundamentals of Algebraic Graph Transformation (Monographs in Theoretical Computer Science. An EATCS Series). Springer, New York (2006)
23. Farwick, M., Agreiter, B., White, J., Forster, S., Lanzanasto, N., Breu, R.: 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, pp. 278–291. Springer (2010)
24. Fogel, K.F., Bar, M.: Open Source Development with CVS. Coriolis Group Books, London (2001)
25. Foster, J.N., Pierce, B.C., Zdancewic, S.: Updatable security views. In: Proceedings of the 2009 22nd IEEE Computer Security Foundations Symposium, CSF'09, pp. 60–74. IEEE Computer Society, Washington (2009)
26. Franz, I.: AllegroGraph. <http://franz.com/agraph/allegrograph/doc/security.html>
27. Fundulaki, I., Marx, M.: Specifying access control policies for XML documents with XPath. In: 9th ACM Symposium on Access Control Models and Technologies, pp. 61–69 (2004)
28. Gallardo, J., Bravo, C., Redondo, M.A.: A model-driven development method for collaborative modeling tools. *J. Netw. Comput. Appl.* **35**(3), 1086–1105 (2012)
29. Garlik. 4store. <http://4store.org/trac/wiki/GraphAccessControl>
30. Gibson-Robinson, T., Armstrong, P., Boulgakov, A., Roscoe, A.: FDR3—A Modern Refinement Checker for CSP. In: Ábrahám, E., Havelund, K. (eds.) Tools and Algorithms for the Construction and Analysis of Systems, Volume 8413 of Lecture Notes in Computer Science, pp. 187–201 (2014)
31. Godik, S., Moses, T. (eds.). eXtensible access control markup language (XACML) version 1.0. 02 (2003)
32. International Organization for Standardization. ISO 16739:2013: Industry Foundation Classes (IFC) for data sharing in the construction and facility management industries (2013)
33. Jaeschke, R.: Encrypting C source for distribution. *J. C Lang. Transl.* **2**(1), 71–80 (1990)
34. Jürjens, J.: Model-based run-time checking of security permissions using guarded objects. In: Leucker, M. (ed.) Proceedings of the 8th International Workshop on Runtime Verification, Volume 5289 of LNCS, pp. 36–50. Springer, Budapest (2008)
35. Lucio, L., Zhang, Q., Nguyen, P.H., Amrani, M., Klein, J., Vangheluwe, H., Traon, Y.L.: Advances in model-driven security. *Adv. Comput.* **93**, 103–152 (2014)
36. Maroti, M., et al.: Next generation (meta)modeling: web- and cloud-based collaborative tool infrastructure. In: 8th Multi-Paradigm Modeling Workshop, Valencia, Spain (2014)
37. Martínez, S., García, J., Cabot, J.: Runtime support for rule-based access-control evaluation through model-transformation. In: Proceedings of the 2016 ACM SIGPLAN International Conference on Software Language Engineering, pp. 57–69. ACM (2016)
38. Montrieux, L., Hu, Z.: Towards attribute-based authorisation for bidirectional programming. In: Proceedings of the 20th ACM Symposium on Access Control Models and Technologies, SACMAT'15, pp. 185–196. ACM, New York (2015)
39. Obeo. Obeo designer team. <https://www.obeodesigner.com/en/collaborative-features>
40. OMG Object Constraint Language. <http://www.omg.org/spec/OCL/> (2014)
41. Oracle. Database Semantic Technologies. http://docs.oracle.com/cd/E11882_01/appdev.112/e11828/fine_grained_acc.htm
42. Papakonstantinou, V., Michou, M., Fundulaki, I., Flouris, G., Antoniou, G.: 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, pp. 103–112. ACM (2012)
43. Rocco, J.D., Ruscio, D.D., Iovino, L., Pierantonio, A.: Collaborative repositories in model-driven engineering [software technology]. *IEEE Softw.* **32**(3), 28–34 (2015)
44. Roscoe, A.W.: Understanding Concurrent Systems. Springer, Berlin (2010)

45. Roscoe, B.: The theory and practice of concurrency (1998)
46. Stevens, P.: Bidirectional model transformations in QVT: semantic issues and open questions. *Softw. Syst. Model.* **9**(1), 7–20 (2008)
47. Syriani, E., Vangheluwe, H., Mannadiar, R., Hansen, C., Mierlo, V., Ergin, H.: AToMPM: A Web-based Modeling Environment. *MODELS 2013 Demonstrations Track* (2013)
48. The Cambridge Dictionary. <http://dictionary.cambridge.org/dictionary/english/obfuscate> (2017)
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. Tolvanen, J.: MetaEdit+: Domain-specific modeling and product generation environment. In: 11th International Conference on Software Product Lines, SPLC 2007, Kyoto, Japan, pp. 145–146 (2007)
54. Varró, D., Bergmann, G., Hegedüs, Á., Horváth, Á., Ráth, I., Ujhelyi, Z.: Road to a reactive and incremental model transformation platform: three generations of the VIATRA framework. *Softw. Syst. Model.* **15**(3), 609–629 (2016). 05/2016
55. Whittle, J., Hutchinson, J.E., Rouncefield, M.: The state of practice in model-driven engineering. *IEEE Softw.* **31**(3), 79–85 (2014)



the MODELS 2017 conference.

Csaba Debreceni obtained his Master's Degrees in Computer Engineering from the Budapest University of Technology and Economics. He is currently a Research Assistant at the Fault-Tolerant Systems Research Group and the MTA-BME Lendület Research Group on Cyber-Physical Systems. His main research interest is model-driven development especially on collaborative modeling. Recently, he received 1st prize in the ACM Student Research Competition at



Labs, where he is a core engineer of the open source framework.

Gábor Bergmann has a Ph.D. in Software Engineering from Budapest University of Technology and Economics (BME) where he is currently an assistant professor. His main research interest is model-driven systems development with a special focus on incremental model queries. He published over 40 scientific papers and has participated in multiple academic research programs including EU FP7 projects SecureChange and MONDO. He is also a co-founder of IncQuery



co-authored more than 50 peer-reviewed papers which received multiple Best Paper Awards. In 2010, he was a visiting scholar at the University of Waterloo. István is a long-time contributor of Eclipse open source projects, serving as the chief technological architect of the VIATRA model transformation framework.

Dr. István Ráth is a co-founder and the lead of innovations at IncQuery Labs. He received his Ph.D. in Software Engineering from the Budapest University of Technology and Economics (BME). His main interest is model-driven systems development with a special focus on incremental and distributed model queries. Since 2006, he is a regular participant of European Union research projects such as SENSORIA, MOGENTES, SecureChange and MONDO. He



FASE 2013, ICMT 2014 and SLE 2016 conferences. He delivered a keynote talk at the IEEE CSMR 2012 and the SOFSEM 2016 conferences and at various international workshops and at the DSM-TP international summer school. He is a co-founder of the VIATRA model query and transformation framework.

Dániel Varró is a full professor of software engineering. His main research interest is model-driven software and systems engineering in the context of smart and safe cyber-physical systems. He is a co-author of more than 100 scientific papers with five Distinguished Paper Awards, and two Most Influential Paper Award. He regularly serves on the program committee of various international conferences in the field and serves on the editorial board of the *Software and Systems Modeling* journal (Springer). He was a program committee co-chair of