# Towards Efficient Evaluation of Rule-based Permissions for Fine-grained Access Control in Collaborative Modeling[*]

Gábor Bergmann[1,2], Csaba Debreceni[1,2], István Ráth[1] and Dániel Varró[1,2,3]
[1]Budapest University of Technology and Economics, Hungary
[2]MTA-BME Lendület Research Group on Cyber-Physical Systems, Hungary
[3]McGill University of Montreal, Canada
{bergmann,debreceni,rath,varro}@mit.bme.hu

## ABSTRACT

In case of collaborative modeling, complex systems are developed by different stakeholders, in offline submissions or online sessions. To guarantee security, access control policies need to be enforced during the collaboration. Levels of required confidentiality and integrity may vary across modeling artifacts, and even features of a single model element.

Fine-grained rule-based access control was proposed for flexible and concise policies. Multiple rules in a policy are inherently subject to conflicts; we have previously shown how to interpret such conflicts in a consistent but also predictable way. However, in online collaboration scenarios, this interpretation has to be repeated upon each small change of the model, thus the computational cost can be prohibitive.

Now we present an improvement on the previous results allowing for incremental recomputation. Our approach is illustrated using a case study of the MONDO EU project.

## 1. INTRODUCTION

### 1.1 Background and Motivation

Model-based systems engineering has become an increasingly popular approach [32] in critical cyber-physical systems followed by many system integrators like airframers or car manufacturers to simultaneously enhance quality and productivity. An emerging industrial practice of such system integrators is to outsource the development of various components to subcontractors in an architecture-driven supply chain. Distributed teams of different stakeholders (system integrators, software engineers of component providers/suppliers, hardware engineers, specialists, certification authorities, etc.) may collaborate using models.

In an *offline collaboration* scenario, collaborators check out an artifact from a version control system (VCS) and commit local changes to the repository in an asynchronous long transaction. In *online collaboration*, engineers may simultaneously edit a model in short synchronous transactions which are immediately propagated to all other users (similarly to online collaborative tools like Google Docs). Several collaborative modeling frameworks exist (CDO [27], EMFStore [28], GenMyModel [2], etc.), but security management is unfortunately still in a preliminary phase.

In fact, collaborative scenarios introduce significant challenges for security management, both in terms of confidentiality and integrity. For instance, the detailed internal design of a specific component needs to be revealed to certification authorities to obtain certification credit but they need to be hidden from competitors who might supply a different component in the system. Furthermore, there may be critical aspects of the system model that may only be modified by domain experts having the appropriate qualifications.

Capturing security policies on the storage (file) level instead of the model level results in inflexible fragmentation of models in collaborative scenarios (although flexibility is key [25]); this can be solved by *fine-grained access control*, where each model element and its features can have its own set of permissions. On the other hand, large industrial models can have millions of model elements, thus explicitly assigning permissions for each of them, as well as maintaining the permissions after changes to the model, would be labor-intensive and error-prone, and would make it difficult to understand the system of privileges.

A *rule-based* approach for concisely defining fine-grained model access control policies has been proposed in [6]. A single rule may grant or deny *nominal permissions* for many elements in a model. Due to this implicit nature, it is possible that several rules would be in direct conflict with each other, assigning contradictory permissions for some model element. Indirect conflicts are also possible, if the fragment of the model revealed to a user is not consistent with itself, or with the write permissions. In [10], we *resolve conflicts* in a deterministic and customizable way to present a consistent and secure updateable view to each user.

The solution in [10] relies on an algorithm that can be executed for a given state of the model to derive the *effective permissions* from the nominal ones. However, this strategy might not scale up to larger system models in case of online collaboration, as the algorithm would have to re-process the permissions of all model elements after each small editing operation, leading to unacceptable response times.

[*]This paper is partially supported by the MTA-BME Lendület 2015 Research Group on Cyber-Physical Systems. The first author was supported by the János Bolyai Research Scholarship of the Hungarian Academy of Sciences.

© 2017 ACM. ISBN 978-1-4503-2138-9.
DOI: 10.1145/1235

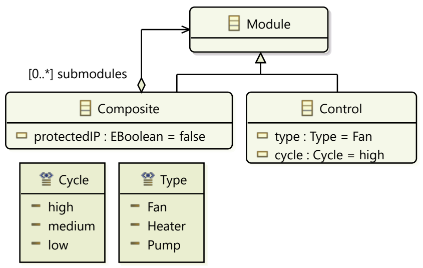

Figure 1: Simplified Metamodel of Offshore Wind Turbines

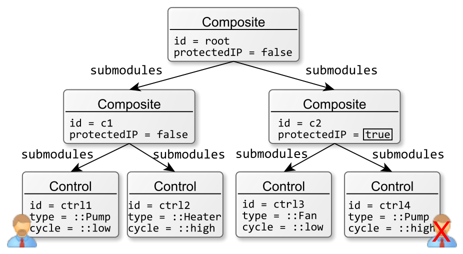

Figure 2: Sample Wind Turbine Instance Model

## 1.2 Goals and Contributions

In this paper, we investigate a possible way to *incrementalize* the conflict resolution schema, so that small changes within large models would incur a small maintenance computation only. Such an incremental resolution method would (a) supercede the limited-scope proof-of-concept incremental conflict resolver used in the online scenario experiments of [6], (b) inherit the beneficial features (flexibility, consistency, determinism) of the non-incremental resolver algorithm [10], and (c) complement the incremental policy rule evaluation and enforcement techniques presented in [6] to guarantee acceptable performance for online in addition to offline collaboration.

We achieve this goal by reformulating the original algorithm of [10] as a recursive Datalog [16] query on the system model (technically recursive VIATRA [5] graph patterns), for which incremental evaluation algorithms are known. We point out subtle difficulties with the evaluation of the resulting patterns, and suggest solutions for dealing with them.

## 2. CASE STUDY

## 2.1 Modeling Language

Several concepts will be illustrated using a simplified version of a modeling language for system integrators of offshore wind turbine controllers, which is one of the case studies of the MONDO EU FP7 project. The metamodel, defined in EMF [29] and depicted by Fig. 1, describes how the system is modeled as modules organized in a containment hierarchy of composite modules. Composite modules may express protected IP and they can ultimately contain control unit modules that are responsible for a given type of physical device (such as pumps, heaters or fans).

*Example 1.* A sample instance model containing a hierarchy of 3 `Composite` modules and 4 `Control` units is shown in Fig. 2 – boxes represent objects, entries within the box are attribute values, and edges are containment references.

### 2.1.1 Access Restrictions

We assume that each control unit type (`Pump`, etc. . . ) is associated with a specific person (referred to as *specialist*) who is responsible for maintaining the model of control unit modules of that specific type. Each such user is able to modify the control units that belong to them, they cannot access to any control unit contained directly or indirectly by composite module that express protected IP. For instance, the user responsible for pump controllers (the *Pump Control Engineer*) can modify the controller `ctrl1` but cannot see the object `ctrl4` as it is contained `c2` which describe that its content is protected. Finally, there is a *principal engineer* that oversees the entire module structure and has read and write access to the entire model.

### 2.1.2 Online Usage Scenario

The system integrator company is hosting the wind turbine control model on their collaboration server, where it is stored, versioned, etc. A group of users may participate in **online collaboration**, where they are continuously connected to the central repository via an appropriate client (e.g. web browser). Through their client, each user sees a live view of those parts of the model that they are allowed to access. The users can modify the model through their client, which will directly forward the change to the collaboration server. Then the server will decide whether the change is permitted under write access restrictions. If it is allowed, then the views of all connected users with appropriate read permissions will be updated transparently and immediately. However, changes introduced by a user may influence the accessibility of model parts for other users. Hence, the access control policy has to be reevaluated when the model is changed.

*Example 2.* After the *Principal Engineer* changes the protectedIP attribute of the composite module `c2` from `true` to `false`, the *Pump Control Engineer* shall be able to access the contained pump control module `ctrl4`.

## 3. SECURITY VOCABULARY

Here we briefly introduce a vocabulary for fine-grained access control; a more detailed treatment of these concepts can be found in our previous work [6] and especially [10].

## 3.1 Model Facts as Assets

In order to provide fine-grained access control, models have to be decomposed into smaller *assets* that can be separately protected. For simplicity, we will consider models as a set of elementary *model facts*. For example, EMF models (ignoring the multi-resource case) consist of the following kinds of model facts:

**Object facts** are pairs formed of a model element (EObject) with its exact type (EClass), for each model element object; e.g. obj(c1,Composite).

**Attribute facts** are triples formed of a source EObject, an attribute name (EAttribute) and the attribute value, for each (non-default) attribute value assignment; e.g. attr(ctrl3, cycle,::low).

**Reference facts** are triples formed of a source EObject, a reference type (EReference) and the referenced EObject, for each containment link and cross-link between objects; e.g. ref(c1,submodules,ctrl2).

Note that for multi-valued attributes and references, each of the multiple entries at a source EObject will be represented by a separate attribute or reference model fact.

In the following, we will highlight the case of object facts, but the same formal framework applies to all assets.

## 3.2 Permissions

The above mentioned model facts are the *assets* that the access control policy will protect against *operations* (typically *read* and *write*) that can be performed by the user. The goal of interpreting the security policy is to come up with an *effective permission function* that assigns a *permission level* to each combination of asset, user and operation.

The assigned permission level takes its value from a *permission lattice* characteristic to the operation. In the simplest case, this is the two-valued lattice $\{deny < allow\}$; in general it is possible to use a more refined lattice instead. In the rest of the paper, we will restrict ourselves to the case where the lattice is a *total order*, i.e. from any two levels, one is considered strictly more restrictive than the other.

For instance, it was previously suggested [10] to use $\{deny < obfuscate < allow\}$ as read permission levels, to express the case where an attribute of an otherwise visible object conveys confidential information that shall not be revealed to the given user, but the attribute can not be completely hidden (since it is used as e.g. an identifier or visual label).

A sample refinement of the write permission levels could be $\{deny < dangle < allow\}$, where cross-references with write permission level *dangle* can not be normally modified by the user, but they can be removed as the side effect of deleting the target object of the reference (if that deletion is permitted), even if the cross-link is not visible to the user. Imagine a traceability link that points from a hidden part of the model to a visible object; the difference between assigning *deny* or *dangle* is that the target object can not be deleted by the user in the former case, while its deletion would be allowed (with an invisible side-effect of removing the traceability link) in the latter case.

## 3.3 Consistency of Secure View

The view presented to a user consists of the set of assets for which they have read permissions (modulo obfuscation, see above). However, an arbitrary set of model facts does not necessarily constitute a valid model; there may be *internal consistency constraints* (also called referential integrity constraints) imposed on the facts by the modeling platform to ensure the integrity of the model representation and the ability to persist, read, and traverse models. A major goal stated in [6, 10] requires that secure models must be synthesized as a set of model facts compatible with all internal consistency rules.

Note that we distinguish these low-level internal consistency rules from high-level, language-specific *well-formedness constraints*. Violating the latter kind does not prevent a model from being processed and stored in the given modeling technology, as error markers can be placed on such violations. Thus only internal consistency is essential for access control.

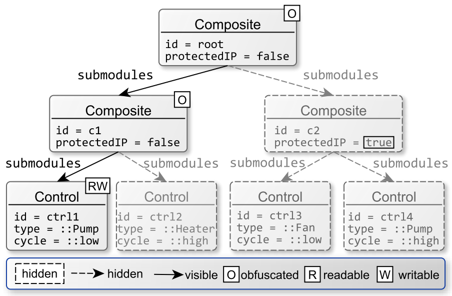

Figure 3: Effective Permissions for Pump Control Engineer

Internal consistency of the filtered view implies that the effective permissions must satisfy a number of constraints, each of which are expressed as a *strong dependency* relationship between two values of the effective permission function, a dependant and a dependee. According to the formalization in [10], this relationship is a *Galois connection*. This essentially means that if a lower bound is known for the permission level of the dependant, then a lower bound on the permission level of the dependee is inferred; and conversely, an upper bound on the dependee puts an upper bound on the dependant.

Examples of these constraints include the following: (a) if an asset is writable (write permission level *allow*), then it must also be visible (at least on the read permission level *allow*); (b) if an object is readable at least on the level *obfuscate*, then its containing object (if any) must also be visible (at least on the level *obfuscate*). All of these constraints have a converse, e.g. if an object is not visible (read permission level is at most *deny*), then any contained objects have their read permission level at *deny* as well.

Apart from the strong dependencies discussed above, the concept of *weak consequences* were also introduced in [10], to express defaults rather than strict implications. For example, all contained elements and attributes of a visible object should also be made visible by default - unless there is another reason to deny the read permission. Another good example would be the converse of the strong dependency in the previous paragraph: if the read permission for an object is known to be at most *obfuscate*, then its write permission must be at most *dangle*; here we can add a weak consequence that said write permission should be *deny* by default, unless specifically overridden.

*Example 3.* Fig. 3 depicts the effective permissions of the running example for the *Pump Control Engineer*. As `c2` expresses protected IP, read and write operations are denied for the composite `c2` and its contents, visualized by dashed borders in case of objects and dashed edges in case of references. On the other hand, the control module `ctrl1` needs to be modifiable, therefor also visible (*strong dependency*) denoted by RW in a square. As `ctrl1` needs to be visible, its container `c1` and also the root module `root` need to be visible at least in an obfuscated way (*strong dependency*) denoted by O in squares. There are no explicit restrictions related to the attributes of control module `ctrl1`, so they inherit visibility from the object asset (*weak consequence*).

**Listing 1** Queries for Access Control Rules

```
  pattern pumpControlPattern(ctrl:Control)
  { Control.type(ctrl,::Pump)}

  pattern propectedIPPattern(c:Composite)
  { Composite.protectedIP(c,true)}
```

**Listing 2** Access Control Rules of the Running Example

```
policy Example deny RW by default {
  rule accessModule allow W to PumpCtrlEng
  { query: pumpControlPattern } priority 1

  rule hideModule deny R to PumpCtrlEng
  { query: propectedIPPattern } priority 2
}
```

## 4. PREVIOUS WORK

Using the vocabulary introduced in Sec. 3, we now briefly revisit the rule-based access control solution and the non-incremental conflict resolver introduced by [10]. More detailed descriptions with concrete examples can be found in the referenced paper.

The design goals were to allow the policy engineer to declaratively, concisely but also flexibly express permissions for a large number of assets in the form of rules. This was achieved by providing a (terminating, deterministic) algorithm that interprets these rules and resolves conflicts among them (also taking into account the defaults), yielding an effective permission function that meets the intentions of the policy engineer and satisfies the dependencies among assets.

### 4.1 Rule-based Access Control

In [6], we introduced a rule-based access control approach, where the rules are defined by graph queries (model queries). Such a query is essentially a formula that can be evaluated on the model. When a query is evaluated, the results consist of a set of *pattern matches*. Rules grant or deny read and write permissions to the assets identified by the *pattern matches* of a query. Of course, a rule may only affect a subset of users, so a selection of users/groups is given as well. Finally, a priority class $\pi \in \Pi$ is given as well, according to which conflicts are resolved: the higher priority rule overrides the lower priority one according to the total order of $\Pi$. To unburden the author, the order of rules in the policy may be taken as a default assignment of priority classes.

To summarize, an access control rule in the policy consists of a set of affected users/groups, a model query to identify affected assets, a priority class $\pi \in \Pi$, an affected operation, and an upper or lower bound on the permission level of that operation.

We allow for compound rules as concise syntactic sugar, where the policy engineer can specify more than one rule (corresponding to the above structure) in one go. For instance, it shall be possible to set both read and write permissions to *deny*, or to set the upper and lower bound of a read permission to *obfuscate* simultaneously.

The conflict resolution process of one user is independent from any other users. Therefore in the following sections, we consider permissions from the perspective of a single user only, and omit the user identity from the subsequently introduced structures.

*Example 4.* Lst. 1 specifies the graph queries for access control rules in VIATRA syntax. Pattern `pumpControlPattern` selects all control module of type ::Pump while pattern `propectedIPPattern` queries the composite modules where the `protectedIP` is to true.

The policy described in Lst. 2 using the syntax introduced in [10] refers these graph pattern to specify access control rules. Rule `accessModule` allows the write operation of modules identified by pattern `pumpControlPattern` and rule `hideModule` denies the read operation of modules selected by pattern `propectedIPPattern`. Priorities of the rules are 1 and 2, respectively.

The policy specifies the global default that all the assets are unreadable and unmodifiable. Rule `accessModule` assigns read permissions to objects `ctrl1` and `ctrl2` which is the result set of pattern `pumpControlPattern`) while rule `hideModule` denies the read permission of `c2` the result set of pattern `propectedIPPattern`.

### 4.2 Judgments

During the process of conflict resolution, the algorithm maintains a set of (typically conflicting) direct and indirect consequences of the access control rules.

We define a *permission set* as a set of *judgments*, where each judgment $j$ is a tuple $j = \langle a, o, p, \psi, \pi \rangle$, where $a \in$ Assets is an asset, $o \in \{R, W\}$ is an operation (read or write) to be performed on the asset, $p \in \text{Levels}_o$ is a permission level associated with the operation, $\psi \in \Psi = \{>, <\}$ indicates whether the judgment puts that permission level as an upper respectively lower bound for the final permission decision, and $\pi \in \Pi$ is a priority class.

A valid permission set is *complete*, i.e. for each asset and operation, it must contain at least one upper bound and one lower bound judgment, and by the ordering of permission levels, the lowest (strictest) upper bound must not be higher than the highest (strictest) lower bound.

The *initial permission set* is obtained from rules and defaults. For all assets and operations, default permissions are included as a pair of judgments (one upper bound, one lower bound, both with the same permission level), with a priority class that is lower than the priority of any query-based rule; this ensures completeness. For each match of the queries of security rules that apply for the user, an upper or lower bound judgment is added; the asset is given by the pattern match, and the operation, priority class and permission level are determined by the rule header.

The goal is to derive the *resolved permission set*, where the highest (most permissive) lower bound is equal to the lowest upper bound for each asset and operation (thereby identifying a single permission level as the *effective permission*), and there are no conflicts (neither directly nor indirectly though dependencies) between the judgments.

*Example 5.* The *initial permission set* obtained from the evaluation of access control rules `accessModule` and `hideModule` is collected in Table 1. Global defaults deny all access with 4 judgments per asset (both bounds $(>, <)$ and operations (R,W) set to *deny*), e.g. the default judgments of the control module `ctrl1` are listed in Table 2.

Table 1: Initial Judgments of the example Rules

$\langle \mathsf{obj(ctrl1,Control)}, W, allow, <, 1\rangle$
$\langle \mathsf{obj(ctrl4,Control)}, W, allow, <, 1\rangle$
$\langle \mathsf{obj(c2,Composite)}, R, deny, >, 2\rangle$

Table 2: Global Defaults on Control Module `ctrl1`

$\langle \mathsf{obj(ctrl1,Control)}, R, deny, <, 0\rangle$
$\langle \mathsf{obj(ctrl1,Control)}, R, deny, >, 0\rangle$
$\langle \mathsf{obj(ctrl1,Control)}, W, deny, <, 0\rangle$
$\langle \mathsf{obj(ctrl1,Control)}, W, deny, >, 0\rangle$

### 4.3 Propagation of Consequences

In Sec. 3.3 we discussed how the requirement of internal consistency introduces dependencies between the permission levels of different assets and operations.

As a judgment may impose dependency constraints on other assets or operations, its consequences can be propagated and directly represented as additional judgments on foreign asset/operation pairs. Using the *propagated consequence* judgments, all indirect conflicts can be transformed into *local conflicts*. The latter means that the conflict is expressed as two directly contradicting judgments for the same asset and operation, as opposed to dependency-driven indirect conflicts between judgments at different assets of operations. It was discussed in [10] how consequence propagation makes focusing on local conflicts sufficient.

For judgment $j = \langle a, o, p, \psi, \pi\rangle$ of bound $\psi \in \Psi = \{>, <\}$, with a dependency on asset and operation $\langle a', o'\rangle$, the propagated strong consequence judgment is denoted as $j_{\langle a', o'\rangle} := \langle a', o', l, \psi, \pi\rangle$ where $l$ is the bound associated with $p$ by the Galois connection described in Sec. 3.3.

Extending the above notion, is also possible to propagate *weak consequences* that encapsulate a default effect of a given judgment, not a necessary condition. Unlike its strong counterpart, a weak consequence does not inherit the priority of the original judgment, but is assigned a lower priority instead, and may be overridden by more dominant judgments without conflicting with the original judgment. It can be formally captured as $j^*_{\langle a', o'\rangle} := \langle a', o', l^*, \psi, \pi^*\rangle$. It was proposed in [10] to assign a priority class to these weak consequences that is lower than the priority of any user-specified rule, but higher than the priority of global defaults that such a weak consequence may override.

*Example 6.* The judgment $\langle \mathsf{obj(c2,Composite)}, R, deny, >, 2\rangle$ obtained from the rule `hideModule` needs to be propagated to the contained objects of composite `c2` as *strong dependency.* Hence, new judgments appear with exactly the same priority, operation, permission level and bound, namely the judgments $\langle \mathsf{obj(ctrl3,Control)}, R, deny, >, 2\rangle$ and $\langle \mathsf{obj(ctrl4,Control)}, R, deny, >, 2\rangle$. However, the new judgment $\langle \mathsf{obj(ctrl4,Control)}, R, deny, >, 2\rangle$ is in local conflict with the judgment $\langle \mathsf{obj(ctrl4,Control)}, R, allow, <, 1\rangle$ obtained from the rule `accessModule` (see Table 1) as they refer to the same asset and operation and contradict on the permission bound (at least *allow* vs. at most *deny*).

### 4.4 Dominance and the Resolution Step

Take any two judgments that are in conflict. Without loss of generality, we may assume that one must *dominate* the other via its higher priority class (see [10] for the unimportant nuance of conflicts within the same priority class). Due to the consequence propagation introduced in Sec. 4.3, it is enough to consider local conflicts, as any conflict will manifest itself somewhere as a local conflict. Therefore we consider pairs of judgments on the same asset and operation, where one is an upper bound judgment, and the other is an incompatible lower bound judgment with different priority.

A *resolution step* takes two judgments that are in a local conflict, and modifies the dominated judgment to make it compatible with the dominant judgment. For locally conflicting judgments $j = \langle a, o, p, \psi, \pi\rangle$ and $j' = \langle a, o, p', \psi', \pi'\rangle$ with $\pi > \pi'$ the conflict resolution replaces $j'$ with $j'' = \langle a, o, p, \psi', \pi'\rangle$. Executing such a step transforms a permission set to a different one.

Observations on the resolution step: (a) $j''$ relaxes $j'$, i.e. upper bounds are raised when replaced, while lower bounds are lowered. (b) $j''$ is guaranteed by the construction to be non-conflicting with $j$. (c) The resolution step upholds the completeness of the permission set.

*Example 7.* In *Ex. 6*, a local conflict is introduced by propagation of strong dependency. Due to domination, the judgment $\langle \mathsf{obj(ctrl4,Control)}, R, deny, >, 2\rangle$ with higher priority relaxes the judgment $\langle \mathsf{obj(ctrl4,Control)}, R, allow, <, 1\rangle$ to $\langle \mathsf{obj(ctrl4,Control)}, R, deny, <, 1\rangle$.

### 4.5 Resolution Process

The non-incremental *resolution process* (see Alg. 1) proposed in [10] iterates over judgments in the order of dominance (i.e. highest priority first), to propagate their consequences (also the weak consequences, unless contradicted by a previously processed judgment) and resolve any local conflicts that are detected.

The analysis in [10] shows that this process always terminates, yields a deterministic (confluent) result regardless of the choice of unprocessed dominant judgment, and the result is a complete and consistent effective permission set.

*Example 8.* The output of the algorithm on the case study is presented in *Ex. 3* and visualized in Fig. 3.

## 5. DECLARATIVE REFORMULATION

### 5.1 Motivation

In Sec. 4, we presented the algorithm of [10] for obtaining the effective permissions from the nominal permissions explicitly assigned by the access control rules of the policy. The algorithm can be run on a state of the model to produce the effective permissions of all model elements.

This is an adequate solution for the offline collaboration scenario, which is characterized by infrequent commit actions at the end of long transactions, which may change an arbitrarily large part of the model. And even if the change themselves turn out to be relatively small, the complexity of processing the commit is at least proportional to the size of the model anyway, unless a sophisticated fine-grained diff solution is applied.

On the other hand, online collaboration is characterized by frequent and short transactions, each of which change only a small part of the model. In case of larger models, response times would suffer greatly if the entire model had to

**Algorithm 1** The resolution process of [10] in pseudocode

```
  ▷ The policy is assumed as an implicit global parameter
function GetEffectivePermissions(model, user)
    $permissionSet \leftarrow getInitialPermissions(model, user)$
    $processed \leftarrow \emptyset$
    while $permissionSet \not\subseteq processed$ do
        $j \leftarrow chooseDominant(permissionSet \setminus processed)$
        $processed \leftarrow processed \cup \{j\}$    ▷ mark as effective
        for all dependencies $\langle a', o' \rangle$ of $j$ do    ▷ propagate
            $conseq \leftarrow \{j_{\langle a', o' \rangle}\}$    ▷ strong consequence
            if $\nexists j' \in processed$ conflicting $j^*_{\langle a', o' \rangle}$ then
                $conseq \leftarrow conseq \cup \{j^*_{\langle a', o' \rangle}\}$    ▷ weak
            end if
            $permissionSet \leftarrow permissionSet \cup conseq$
        end for
        $conflicts \leftarrow localConflictsOf(j, permissionSet)$
        for all $j' \in conflicts$ do    ▷ resolve locally
            $j'' \leftarrow ResolutionStep(j, j')$    ▷ $j$ dominates
            $permissionSet \leftarrow permissionSet \cup \{j'\} \setminus \{j''\}$
        end for
    end while
    return $permissionSet$
end function
```

be re-processed upon each small change. Therefore, we aim to obtain the effective permission function by an incremental computation.

The evaluation of access control queries and the actual enforcement of permissions is also performed incrementally [6]. It would seem natural to try to express the conflict resolution and derivation of effective permissions as model queries, and use the same technology [31, 5] that already provides the incremental evaluation of access control queries.

In the following, we describe our efforts to reformulate the algorithmic conflict resolution process into a declarative computation. We use the formalism of recursive graph queries (Datalog [16]), specifically the Viatra syntax.

## 5.2 Naive Reformulation

For the sake of simplicity, the following discussion will focus on object assets only. The conflict resolver takes the following inputs (extensional relations in Datalog parlance):

- The explicit judgments that are immediately obtained by evaluating the access control rules in the policy; as mentioned above, each match of the associated model query of the rule will yield a judgment (in the initial permission set).
- Low-priority judgments corresponding to global defaults (can be merged with the former).
- Relations describing the connections between assets, in order to enable the propagation of consequent judgments. For instance, in case of object facts, a binary relation describing the containment hierarchy is necessary.

From these extensional relations, the following incrementally maintained queries (intensional relations) are derived (see also Lst. 3):

`judgment` matches all inferred judgments $j = \langle a, o, p, \psi, \pi \rangle$, computed as the union of explicit judgments, the strong and weak consequent judgments, and the relaxed forms of dominated judgments.

`relaxedJudgment` identifies judgments that are dominated (see next query), and computes their relaxed form (inheriting the bound of the dominant judgment).

`domination` identifies the judgments in local conflict (sufficient as per Sec. 4.3) and the dominant one among them; this query matches pairs of dominating and dominated judgments in local conflict, respectively $j = \langle a, o, p, \psi, \pi \rangle$ and $j' = \langle a, o, p', \psi', \pi' \rangle$, that are defined on the same asset and operation, with the dominating judgment having higher priority ($\pi > \pi'$) and the dominated bound contradicting it ($p' \psi p$ for $\psi \in \Psi = \{>, <\}$, as expressed by helper pattern `permissionOutOfBound`). For efficiency, only effective judgments (see next query) are considered for the role of the dominating judgment.

`effectiveJudgment` matches any judgment that is not dominated at all by any judgment, i.e. there is no corresponding match of the `domination` query. Note that this is a negative composition.

`strongConsequence` computes the propagated consequences of effective judgments; it matches pairs of judgments $j = \langle a, o, p, \psi, \pi \rangle$ and $j' = \langle a', o', p', \psi', \pi' \rangle$ where $j$ is an `effectiveJudgment` and $j' = j_{\langle a', o' \rangle}$. For strong consequences, $\pi = \pi'$ is also required. The results are computed as a union of cases, two for each kinds of dependency; one where a dominant lower bound on a dependant propagates a lower bound consequent judgment to the dependee asset, and one where a dominant upper bound on a dependee propagates an upper bound consequent judgment to the dependant asset. In case of object assets, such dependencies are found between the read and write operations of the same asset, and between contained and container objects (see Sec. 3.3).

`weakConsequence` is constructed similarly, except the consequent has a constant, low priority.

**Listing 3** Naive solution in Viatra syntax (extracts)

```
pattern judgment(user,asset,op,bound,dir,prio)
{ find explicitJudgment(user,asset,op,bound,dir,prio);
} or {
 find relaxedJudgment(user,asset,op,bound,dir,prio);
} or {
 find strongConsequence(user,asset,op,bound,dir,prio);
}
pattern relaxedJudgment(user,asset,op,bound,dir,prio)
{ find domination(user,asset,op,_,dir,prio,bound);
}
pattern effectiveJudgment(user,asset,op,bound,dir,prio)
{ find judgment(user,asset,op,bound,dir,prio);
  neg find domination(user,asset,op,bound,dir,prio,_);
}
pattern domination(user,asset,op,dBound,dDir,dPrio,eBound)
{ find judgment(user,asset,op,dBound,dDir,dPrio);
  find effectiveJudgment(user,asset,op,eBound,eDir,ePrio);
  find permissionOutOfBound(eDir,eBound,dBound);
  eDir!=dDir;
}
pattern strongConsequence(user,dAsset,dOp,dBound,dir,prio)
{ // Case 1: read vs write, AT_LEAST
  find effectiveJudgment(user,eAsset,eOp,eBound,dir,prio);
  dAsset == eAsset; dir == BoundDirection::AT_LEAST;
  eOp == SecurityOperation::WRITE;
  eBound == WriteLevels::ALLOW_WRITE;
  dOp == SecurityOperation::READ;
  dBound == ReadLevels::ALLOW_READ;
} or {...} // Other cases
```

Note that for a given asset and operation, in addition to the strictest effective upper or lower bound, there can additionally occur less strict, superfluous judgments. It is possible to eliminate them based on subsumption by stricter effective judgments. If subsumed judgments are then eliminated from the essential judgements, then their consequences won't be propagated, and some computational cost can be saved. The simplest solution is probably to adjust `domination` so that it does not check whether the bound direction of the dominated judgment is opposite to the dominant one (i.e. remove the last line of `domination` in Lst. 3), then in addition to actual dominations, it will also match most such subsumed judgments (except for the case where a judgment is subsumed by another one in the same priority class).

The effective permission function is finally obtained as the result set of the `effectiveJudgment` query, where the strictest lower and upper bounds will coincide (see proof in [10]). Let us now investigate the *correctness* of this solution.

*Proposition.* This group of queries yield the same effective permission set as Alg. 1.

*Proof sketch.* One has to consider the way of determining the set of effective judgments, as the rest follows fairly trivially. The original algorithm iteratively selected the highest-priority yet-unprocessed judgment and designated it as an effective judgment (by putting it into the `processed` set). By contrast, here judgments are considered effective when they are not dominated by any other judgment. It takes some thinking to see that the two conditions yield the same results. We use downwards induction by priority class; assume that the two versions of the effective permission set are equivalent above priority class $\pi$. When the original algorithm selects a dominant judgment $j$ of priority $\pi$ as the new effective judgment, the priority-based ordering ensures that there are no unprocessed judgments in the permission set that could dominate it, and of course already processed judgments are not in conflict with $j$ and thus do not dominate it (otherwise $j$ would have been relaxed and removed earlier when they were processed); consequently, the `effectiveJudgment` query would match $j$. Conversely, if the `effectiveJudgment` query matches the judgment $j$ of priority $\pi$, that means no other higher-priority effective judgment dominates it, therefore (by the induction hypothesis) the original algorithm never eliminates it from the permission set, and it is eventually selected as an effective judgment.

## 5.3 Non-Stratifiability

A significant flaw of the naive reformulation in Sec. 5.2 is revealed when we consider how the results of these queries are derived from each other.

Not counting extensional relations, the query `judgment` derives from `relaxedJudgment`, `strongConsequence` and `weakConsequence` as depicted in Fig. 4. The latter two also derive their results from `effectiveJudgment`. The `relaxedJudgment` references `domination`, while `domination` uses `judgment` and `effectiveJudgment`. Finally, `effectiveJudgment` is computed from `judgment`, `permissionOutOfBound` a negation of `domination`.

It is immediately obvious that this is a recursive query structure, e.g. `judgment` refers indirectly to `domination` which refers to `judgment` again. Fortunately, it is possible to evaluate recursive queries (there are even multiple incremental strategies [17]).

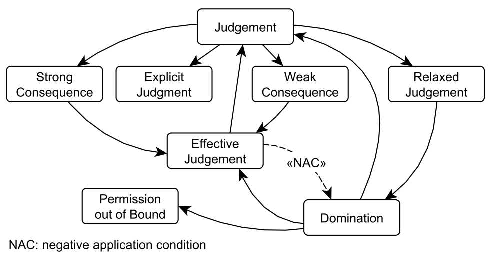

Figure 4: Dependency Graph of the Naive Queries

However, these strategies require a property of the query structure called *stratification* [16], which the reformulated conflict resolver does not exhibit. Informally, a Datalog program is stratified if there are no negations used in the cyclic references between predicates (i.e. relations or patterns). In our case, a counterexample would be `effectiveJudgment` negatively referring `domination`, which in turn uses `effectiveJudgment`.

Typical query engines only handle stratified query structures, as non-stratified ones do not even have a single accepted semantics [16], and 2-valued semantics are often not possible. See for instance the famous example of a non-stratified Datalog program "this sentence is a lie".

Fortunately, there are a few relaxations of the criteria for stratifiability. Recursions can still have definite (2-valued) semantics (and evaluation algorithms) if they are *locally* [21], *modularly* or at least *weakly* [24] stratifiable. Informally, these relaxed notions of stratification investigate the dependencies between individual tuples (so-called *ground atoms*) as opposed to whole relations; it is sufficient if there is no negation in circular references between tuples (in the *ground Datalog program*). The subtle differences between local, modular and weak stratification manifest themselves in the ways how already known facts are used to cull unnecessary (e.g. unsatisfiable) references before cycles are evaluated to be negation-free; the formal details are not explored here.

Although the query structure of Sec. 5.2 is neither stratifiable nor locally stratifiable, it is weakly and even modularly stratifiable. The reason is that the only occurrence of negation is between predicates `effectiveJudgment` and `domination`; however, a judgment may only be dominated by higher-priority judgments. After eliminating judgment pairs from `domination` for which the priority comparison constraints fail (which is allowed in modular, but not in local stratification), we are left with no tuple-level negative cycles.

Although modular stratifiability shows the existence of 2-valued semantics, it is not necessarily compatible with all evaluation strategies, especially incremental ones. Even if a given strategy is guaranteed to yield correct results, concrete tools such as VIATRA may pessimistically refuse to evaluate a query structure that is not stratifiable (as detecting e.g. local stratification may be undecidable [21]).

## 5.4 Performance Considerations

Beyond the question of stratification, the recursive query structure also introduces performance challenges.

Imagine a single-root containment hierarchy consisting of $n$ objects altogether. Suppose we define $k$ separate access control rules in $k$ different priority classes, each of which applies to the root object, such that every other rule (those in even priority classes) sets the read permission to *allow*, while the rest of the rules set it to *deny*. In this case, query evaluation performance depends greatly in the order of computations, even if the end result is guaranteed to be the same.

If the query engine happens to propagate tuples in the ascending order of the priorities of corresponding judgments, then first all $n$ objects are set to invisible by the lowest priority rule, then the next rule is found to dominate the previous one and all $n$ objects are made visible again, etc., until finally the effective permissions are obtained in $O(n*k)$ steps.

If, on the other hand, the query engine propagates tuples in the order of descending priority, then the final read permissions are set immediately in the highest priority class , and then the rest of the rules are merely dominated, but never propagated. This leads to $O(n+k)$ steps of computation.

Altogether, if the evaluation follows the original procedure in Alg. 1 and processes high-priority rules first, then (strong) consequences are never overruled, minimizing the amount of work to be done. Unfortunately, declarative queries can be executed with an arbitrary computation order chosen by the query engine. Ordering by rule priority is a highly application-specific optimization, and there is no standard way to make a query engine such as VIATRA aware of it.

## 5.5 Improved Query Structure

The previous sections identified two significant issues with the naive reformulation of the access control conflict resolution problem, stratifiability and performance. Still, we do not give up on using a general-purpose query tool to provide incremental evaluation; instead, we propose a way to modify the query structure such that it avoids both problems.

Fortunately, the solutions to both issues involve the same weak ordering of tuples based on priority class. We therefore propose to apply *partial grounding*, where given a policy of $k$ priority classes, we substitute all $k$ values into each priority variable of the query, blowing up each predicate into $k$ disjoint shard predicates ($k^2$ in case of `domination` where two judgments of different priority are involved). Priority comparisons can be statically evaluated in this case.

There is one corner case where negative recursion still persists, despite sharding the predicated according to priority: the `effectiveJudgment` query for a given priority class negatively calls the `domination` relation for the same priority (of the dominated judgment), which is in turn derived from dominated `judgment` matches of that priority, that on one branch rely on `strongConsequence` matches, which call `effectiveJudgment`, all of the same priority level. We propose the following solution: we replace `domination` with `dominationBy` that will no longer imply that a dominated judgment exists; it merely asserts that an effective judgment exists at a given priority, that would dominate any lower-priority judgment at the given asset and the given permission bound. This way the negative recursion is averted.

For priority class $\pi$, we define the following predicates (see also Lst. 4):

**$\texttt{judgment}_\pi$** matches all inferred judgments $j = \langle a, o, p, \psi, \pi \rangle$ of priority class $\pi$, computed as the union of explicit judgments in class $\pi$, the strong and weak consequent judgments in class $\pi$, and the relaxed forms of dominated judgments in class $\pi$.

**$\texttt{relaxedJudgment}_\pi$** identifies judgments of priority class $\pi$ that are dominated (see next query) by any priority $\pi' > \pi$, and computes their relaxed form (inheriting the bound of the dominant judgment).

**$\texttt{dominationBy}_\pi$** will match (independently of the existence of any dominated judgments) all effective judgments $j = \langle a, o, p, \psi, \pi \rangle \in \texttt{effectiveJudgment}_\pi$ of priority $\pi$ and all potential bounds $p'$ for the corresponding asset and operation that would be dominated by it in local conflict ($p'\psi p$ for $\psi \in \Psi = \{>, <\}$).

**$\texttt{effectiveJudgment}_\pi$** filters $\texttt{judgment}_\pi$ to those judgments that are not dominated, i.e. there is no corresponding match of $\texttt{dominationBy}_{\pi'}$ for any $\pi' > \pi$.

**$\texttt{strongConsequence}_\pi$** computes the propagated consequences of effective judgments at class $\pi$.

**$\texttt{weakConsequence}_\pi$** analogously.

In the above queries, the priority class comparisons "$\pi' > \pi$" are statically evaluated, and the generated disjunctions only include the clauses where the result is true.

With this solution, the query structure is now stratifiable, addressing the concerns of Sec. 5.3. Also, a query engine evaluating (or updating, in case of incrementality) it can process individual predicates in their topological order, which addresses the issue raised in Sec. 5.4.

Once stratification is achieved, let us take a look at what forms of (positive) recursion remain, keeping in mind that these no longer pose an obstacle to incremental matching. First, $\texttt{relaxedJudgment}_\pi$ contributes to $\texttt{judgment}_\pi$ and vice versa. Note that there is no circularity on the tuple level, as the original dominated judgments (as well as subsumed ones) are always farther from the effective judgment than their relaxed versions. Moreover, $\texttt{judgment}_\pi$ is partially expressed from $\texttt{strongConsequence}_\pi$, which finds consequences of $\texttt{effectiveJudgment}_\pi$ that of course uses $\texttt{judgment}_\pi$; in this case it depends on our internal consistency constraints whether consequences can cyclically justify each other (in the simple cases described in this paper, they can't). In application scenarios where we know that there are no tuple-level cycles, the fast default algorithm suffices to efficiently evaluate recursive queries; for other cases, VIATRA provides the more complex DRed algorithm [17] as well, guaranteeing correct incremental maintenance for any stratified query structure.

**Listing 4** Refined solution in VIATRA syntax (extracts)

```
pattern relaxedJudgement_at_0(user,asset,op,bound,dir){
 find judgement_at_0(user,asset,op,dBound,dir);
 find domination_by_1(user,asset,op,_dBound,bound);
} or {
 find judgement_at_0(user,asset,op,dBound,dir);
 find domination_by_2(user,asset,op,_dBound,bound);
} or {...}
pattern effectiveJudgement_at_0(user,asset,op,bound,dir){
 find judgement_at_0(user,asset,op,bound,dir);
 neg find domination_by_1(user,asset,op,bound,_);
 neg find domination_by_2(user,asset,op,bound,_);
 // ...
}
pattern domination_by_1(user,asset,op,dBound,dir){
 find effectiveJudgment_at_1(user,asset,op,eBound,dir);
 find permissionOutOfBound(dir,eBound,dBound);
}
// ...
```

Finally, a note on space and time cost: a straightforward upper bound on the size of the match sets can be obtained by taking the Cartesian product of the domains of query variables. In our example, there are 3 priority classes, 2 controlled operations, 3 permissions levels (both for read and write), etc.; in the end, for a fixed security policy, the upper bound is proportional to both the model size (number of assets) and the number of active (e.g. simultaneously connected) users. In the non-recursive case, incrementally maintaining the result of queries has time complexity proportional to the size of the change (in the inputs, in the results, and also in any internal cache structure), regardless of the size of the unchanged parts. For recursive queries, such a straightforward linear relationship can not always be established; performance analysis and improvements (beyond the considerations of Sec. 5.4) are left as future work.

## 6. RELATED WORK

**File-based Access Control.** Off-the-shelf file systems typically require resources (files and folders) to be explicitly labeled with permissions that take the form of an Access Control List (ACL), or the simplified form user/group/others. An ACL consists of entries (judgments) regarding which user/subject is granted or denied permission for a given operation. Conflict resolution is usually priority-based (first entry applies) within the access control list, and restrictive among type II and type III conflicts (e.g. contents of a hidden folder cannot be seen, regardless of ACLs inside).

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 judgments, which is difficult to manually maintain as the model evolves), and respect the modeling-specific challenges of consistency (such as permission dependencies of cross-references); all the while being more flexible in the conflict resolution method.

**Access Control for XML Documents.** A number of standards such as XACML [15] (OASIS standard) provide fine-grained access control for XML documents. These type of documents are similar to models in a way, that they consists of nodes with attributes that may contain other nodes. XACML provides several combining algorithms to select from contradicting policies. Similarly to our solution, it may use external and internal priorities together (*ordered-(deny/permit)-overrides*) or only internal priorities (*unordered-(deny/permit)-overrides*). In [12], 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, while our approach [10] also considers e.g. cross-references.

**Context-aware Access Control RDF Stores.** Models can be persisted into triples to store them in triple or quad stores (Neo4EMF[4], EMF Triple). Graph-based access control is a popular strategy for many RDF stores (4store [14], Virtuoso , IBM DB2) developed for storing large RDF data. In case of RDF, fine-grained specification of access control permissions are defined at triple level. In [1], a graph-based policy specification language proposed over SPARQL [22], but resolution of contradicting rules are not discussed. Amit Jain et. al. [18] propose an access control model for RDF and a two-level conflict resolution strategy that also takes inconsistencies into account similar to our solution. But, it uses only restrictive resolution without any configuration of default values or priorities between rules.

**Collaborative Modeling Environments.** Currently, fine-grained access control is not considered in the state of the art tools of MDE such as MetaEdit+[30], VirtualEMF[8], WebGME[19], EMFStore [28], GenMyModel [2], Obeo Designer Team [20], MDEForge [3] or the tools developed according to [13]. See also the broader survey in [23].

The generic framework CDO [27] (used e.g. in [20]) provides both online collaboration and role-based access control with type-specific (class, package and resource-level) permissions, but no facility for instance level access control policy specifications. However, there is a pluggable access control mechanism that can specify access on the object level; it should be possible to integrate fine-grained solutions such as the currently proposed system.

The collaborative hardware design platform VehicleFORGE stores their model in graph-based databases and has an access control scheme TrustForge [9] that uses an implementation of KeyNote [7] 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 [26] 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 [11].

## 7. SUMMARY

To support rule-based fine-grained access control under internal consistency constraints, we have taken a significant step towards efficient online collaboration by reformulating the permission assignment problem into a special class of Datalog programs that can be incrementally evaluated (taking advantage of the priority hierarchy). We have also sketched a proof of correctness for the reformulation. As a quick proof of concept [1], we have implemented and evaluated queries according to this scheme for an example policy.

In the future, we plan to (a) implement an automated translation from an arbitrary policy to such a query structure, (b) set up benchmarks to study (and, if necessary, improve) the performance of the solution in practice, (c) investigate incrementality with respect to policy changes.

## 8. REFERENCES

[1] Fabian Abel et al. Enabling adanced and context-dependent access control in RDF stores. In *The Semantic Web, 6th Int. Semantic Web Conf., 2nd Asian Semantic Web Conf.*, pages 1–14, 2007.

[2] Axellience. GenMyModel. http://www.genmymodel.com.

[3] Francesco Basciani, Juri Di Rocco, Davide Di Ruscio, Amleto Di Salle, Ludovico Iovino, and Alfonso

[1]Available at https://github.com/bergmanngabor/CommitMDE17reproduction

Pierantonio. MDEForge: an extensible web-based modeling platform. In *CloudMDE@MoDELS*, 2014.

[4] Amine Benelallam, Abel Gómez, Gerson Sunyé, Massimo Tisi, and David Launay. Neo4EMF, A scalable persistence layer for EMF models. In *Modelling Foundations and Applications - 10th European Conf., ECMFA 2014*, pages 230–241, 2014.

[5] Gábor Bergmann, István Dávid, Ábel Hegedüs, Ákos Horváth, István Ráth, Zoltán Ujhelyi, and Dániel Varró. Viatra 3: A reactive model transformation platform. In *Theory and Practice of Model Transformations - 8th International Conference, ICMT 2015, Held as Part of STAF 2015, L'Aquila, Italy, July 20-21, 2015. Proceedings*, pages 101–110, 2015.

[6] Gábor Bergmann, Csaba Debreceni, István Ráth, and Dániel Varró. Query-based access control for secure collaborative modeling using bidirectional transformations. In *Proceedings of the ACM/IEEE 19th International Conference on Model Driven Engineering Languages and Systems, Saint-Malo, France, October 2-7, 2016*, pages 351–361, 2016.

[7] Matt Blaze and Angelos D Keromytis. The keynote trust-management system version 2. 1999.

[8] Cauê Clasen, Frédéric Jouault, and Jordi Cabot. VirtualEMF: A model virtualization tool. In *Advances in Conceptual Modeling. Recent Developments and New Directions*, pages 332–335, 2011.

[9] Penn University DARPA VehicleFORGE. *TrustForge: Flexible Access Control for VehicleForge.mil Collaborative Environment*, 2012.

[10] Csaba Debreceni, Gábor Bergmann, István Ráth, and Dániel Varró. Deriving effective permissions for modeling artifacts from fine-grained access control rules. In *Proceedings of the 1st International Workshop on Collaborative Modelling in MDE (COMMitMDE 2016) co-located with ACM/IEEE 19th International Conference on Model Driven Engineering Languages and Systems (MoDELS 2016), St. Malo, France, October 4, 2016.*, pages 17–26, 2016.

[11] Matthias Farwick, Berthold Agreiter, Jules White, Simon Forster, Norbert Lanzanasto, and Ruth Breu. A web-based collaborative metamodeling environment with secure remote model access. In *Web Engineering, 10th International Conference, ICWE 2010, Vienna, Austria, July 5-9, 2010. Proceedings*, volume 6189 of *LNCS*, pages 278–291. Springer, 2010.

[12] Irini Fundulaki and Maarten Marx. Specifying access control policies for XML documents with XPath. In *9th ACM Symposium on Access Control Models and Technologies*, pages 61–69, 2004.

[13] Jesús Gallardo, Crescencio Bravo, and Miguel A. Redondo. A model-driven development method for collaborative modeling tools. *J. Network and Computer Applications*, 35(3):1086–1105, 2012.

[14] Garlik. 4store. http://4store.org/trac/wiki/GraphAccessControl.

[15] S. Godik and T. Moses (eds). eXtensible access control markup language (XACML) version 1.0. 02 2003.

[16] Todd J. Green, Shan Shan Huang, Boon Thau Loo, and Wenchao Zhou. Datalog and recursive query processing. *Found. Trends databases*, 5(2):105–195, November 2013.

[17] Ashish Gupta, Inderpal Singh Mumick, and V. S. Subrahmanian. Maintaining views incrementally (extended abstract). In *Proc. of the Int. Conf. on Management of Data, ACM*, pages 157–166, 1993.

[18] Amit Jain and Csilla Farkas. Secure resource description framework: an access control model. In *11th ACM Symposium on Access Control Models and Technologies*, pages 121–129, 2006.

[19] Miklos Maroti et al. Next Generation (Meta)Modeling: Web- and Cloud-based Collaborative Tool Infrastructure. In *8th Multi-Paradigm Modeling Workshop*, Valencia, Spain, 09/2014 2014.

[20] Obeo. Obeo designer team. https://www.obeodesigner.com/en/collaborative-features.

[21] Luigi Palopoli. Testing logic programs for local stratification. *Theoretical Computer Science*, 103(2):205 – 234, 1992.

[22] E. Prud'hommeaux and A. Seaborne. SPARQL query language for RDF. 07 2016.

[23] J. Di Rocco, D. Di Ruscio, L. Iovino, and A. Pierantonio. Collaborative repositories in model-driven engineering [software technology]. *IEEE Software*, 32(3):28–34, May 2015.

[24] Kenneth A. Ross. Modular stratification and magic sets for datalog programs with negation. *J. ACM*, 41(6):1216–1266, November 1994.

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

[26] Eugene Syriani, Hans Vangheluwe, Raphael Mannadiar, Conner Hansen, Van Mierlo, and Huseyin Ergin. AToMPM: A Web-based Modeling Environment. MODELS 2013 Demonstrations Track, 2013.

[27] The Eclipse Foundation. CDO. http://www.eclipse.org/cdo.

[28] The Eclipse Foundation. EMFStore. http://www.eclipse.org/emfstore.

[29] The Eclipse Project. Eclipse Modeling Framework. http://www.eclipse.org/emf/.

[30] Juha-Pekka Tolvanen. MetaEdit+: Domain-specific modeling and product generation environment. In *Software Product Lines, 11th Int. Conf. SPLC 2007, Kyoto, Japan*, pages 145–146, 2007.

[31] Zoltán Ujhelyi, Gábor Bergmann, Ábel Hegedüs, Ákos Horváth, Benedek Izsó, István Ráth, Zoltan Szatmári, and Dániel Varró. EMF-IncQuery: An integrated development environment for live model queries. *Science of Computer Programming*, (0):–, 2014.

[32] Jon Whittle, John Hutchinson, and Mark Rouncefield. The state of practice in model-driven engineering. *IEEE Software*, 31(3):79–85, 2014.