Deriving Effective Permissions for Modeling Artifacts from Fine-grained Access Control Rules*

Csaba Debreceni[1,2], Gábor Bergmann[1,2], István Ráth[1] and Dániel Varró[1,2]
[1]Budapest University of Technology and Economics,
Department of Measurement and Information Systems,
1117 Budapest, Magyar tudósok krt. 2.
[2]MTA-BME Lendület Research Group on Cyber-Physical Systems
{debreceni,bergmann,rath,varro}@mit.bme.hu

ABSTRACT

In case of collaborative modeling, complex systems are developed by different stakeholders. 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 to meet the needs of flexible and concise access control. Rule-based policies are inherently subject to conflicts between the rules; these conflicts should be interpreted in a consistent but also predictable way that caters to the preferences of the policy engineer.

We propose a deterministic, parameterizable resolution strategy between conflicting rules to calculate effective access permissions for each fact in the model. Our approach is illustrated using a case study of the MONDO EU project.

1. INTRODUCTION

1.1 Background and Motivation

The adoption of model driven engineering (MDE) by system integrators (like airframers or car manufacturers) has been steadily increasing in the recent years [23]. The use of models also intensifies collaboration between distributed teams of different stakeholders (system integrators, software engineers of component providers/suppliers, hardware engineers, specialists, certification authorities, etc.) via model repositories, which significantly enhances productivity and reduces time to market. An emerging industrial practice of system integrators is to outsource the development of various design artifacts to subcontractors in an architecture-driven supply chain.

*This paper is partially supported by the EU Commission with project MONDO (FP7-ICT-2013-10), no. 611125. and the MTA-BME Lendület 2015 Research Group on Cyber-Physical Systems.

Collaboration scenarios include traditional *offline collaborations* with asynchronous long transactions (i.e. to check out an artifact from a version control system and commit local changes afterwards) as well as *online collaborations* with short and synchronous transactions (e.g. when a group of collaborators simultaneously edit a model, similarly to well-known on-line document / spreadsheet editors). Several collaborative modeling frameworks (like CDO [8], EMFStore [9], etc.) exist to support such scenarios.

However, such 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 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 certification proofs and credits. On the other hand, there are highly critical aspects of the model that may only be modified by (or with approval from) specialists 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; 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 [4]. A single rule may grant or deny permissions for a large number of assets in a model.

However, due to this implicit nature, it is possible that several rules would be in direct conflict with each other, assigning contradictory *nominal permissions* for some model element. It is also possible for rules to conflict indirectly, if the fragment of the model revealed to a user in not consistent with itself, or with the allowed write operations.

In either case, the access control mechanism must *resolve conflicts* in a deterministic way by deriving the *effective permissions* from the nominal ones, to present a consistent and secure updateable view to each user.

1.2 Goals and Contributions

The main objective of the paper is to propose a conflict resolution technique to support secure collaboration based

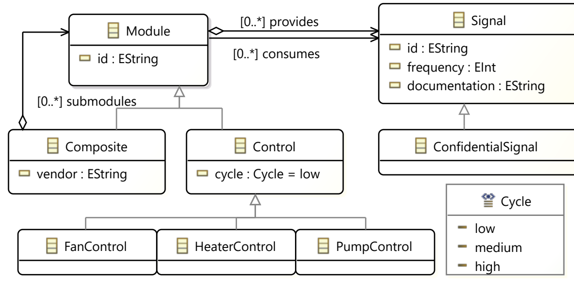

Figure 1: Simplified Metamodel of Wind Turbine Controllers

on a rule-based fine-grained access control policy, as proposed in [4]. In particular, we aim to address the following goals:

G1 *Deterministic Conflict Resolution*
the solution must define a deterministic application of the access control rules to obtain the same effective permissions after every execution;

G2 *Consistency of Effective Permissions*
the solution must synthesize secure models that are compatible with all internal consistency rules;

G3 *Flexible Adaptation to the Intention of the Policy Owner*
the solution must provide the concept of a parameterizable policy language that is able to fine-tune the resolution of conflicting rules.

In this paper, we propose a configurable, multi-stage conflict resolution approach that considers internal and external propertiesof rules, as well as dependencies among the assets they apply to, when determining their precedence. We demonstrate the applicability of the proposed approach on a case study.

2. CASE STUDY

2.1 Modeling Language

Our conflict resolution 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 [2]. The metamodel, defined in Ecore [19] and depicted by Fig. 1, describes how the system is modeled as modules providing and consuming signals that send messages after a specific amount of time defined by the frequency attribute. Modules are organized in a containment hierarchy of composite modules shipped by external vendors, and ultimately containing control unit modules responsible for a given type of physical device (such as pumps, heaters or fans) with specific cycle priorities. For external engineers, a documentation is attached to each signal to clarify its responsibilities. Some of the signals are treated as confidential intellectual property.

A sample instance model containing a hierarchy of 3 `Composite` modules with 4 `Control` units as submodules, providing 6 `Signals` altogether where two of them are `Confidential Signals`, is shown on Fig. 2. Boxes represent objects (with attribute values as entries within the box). Arrows with diamonds represent containment edges, while arrows without diamonds represent cross-references. Different type of lines, boundaries and letters in squares are related to the granted permissions that will be discussed later during the paper.

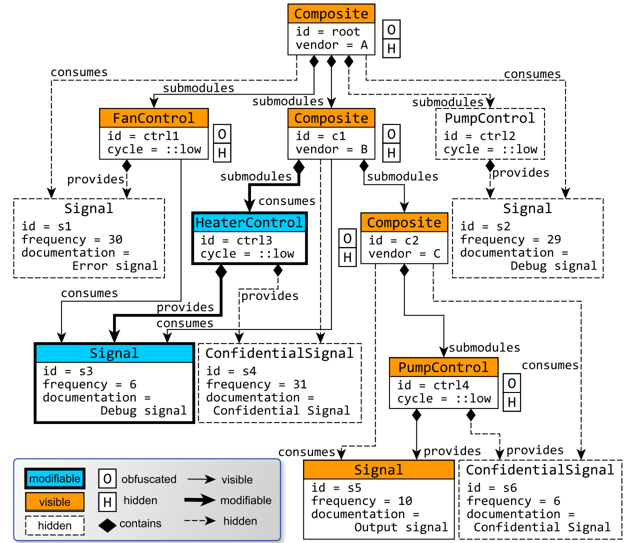

Figure 2: Sample Wind Turbine Instance Model

2.2 Security Requirements

Designing wind turbine control units requires specialized knowledge. There are 3 kinds of control units, and each kind can only be modified by specialist users with the appropriate qualification: *heater*, *pump* and *fan control engineers*. Specialist are not allowed to modify (and in some cases, read) parts of the model that would require a different kind of qualification. For this purpose, the following security requirements are stated in relation with control unit specialists:

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

R2 Specialists must see only those signals that are in 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 (incl. the owned control unit) contained *transitively* in such a composite.

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

R4 Specialists must see which modules consume signals provided by their owned control units.

R5 Specialists must not able to see confidential signals.

3. PRELIMINARIES

3.1 Model Facts as Assets

In our previous work [4], models are decomposed to a set of elementary *model facts*. For example, EMF models 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(c1, vendor,B).

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,consumes,s4).

Note that there are multi-valued attributes and references, where an EObject is allowed to host multiple attribute values (or reference endpoints) for that property. For such properties, each of these multiple entries at a source EObject will be represented by a separate attribute (or reference) model fact. Moreover, there are *opposite* references defined as a pair of references where the existence of a relation depends on its pair. For example, the opposite of a containment reference *eContainment* is the container reference *eContainer*.

These model facts are the *assets* that the access control policy will protect; though there are a few deviations from one-to-one correspondence, such as a reference and its opposite reference (if exists) are considered a single asset.

3.2 Model Obfuscation

Obfuscation is defined as the process of *"making something less clear and harder to understand, especially intentionally"*. The first purpose of obfuscation in programming was to distribute C sources in an encrypted way to prevent the access to confidential intellectual property in the code [14]. In a modeling environment, the same concept applies.

A model obfuscator such as VIATRA Model Obfuscator [22] obfuscates structured graph-like models (e.g. XML documents, EMF-based models) by altering data values (such as names, identifiers or other strings) in a way that the structure of the model remains the same. Two data values that were identical before the obfuscation will also be identical after it, but the obfuscated value computed based on an input obfuscation string will be completely different (e.g. "root" may become "oA3DD43CF5").

Several publications [13, 18] discuss implementation of the obfuscation function which is out of scope of this paper. We assumes that an obfuscated identifier remains unique and it is possible to revert it by the original owner of the model using a private key.

In the context of access control, obfuscated data describes the existence of a model fact (e.g. a value is assigned to the attribute of an object), but the meaning of that fact remains secret. Additionally, when only the existence of object needs to be presented, its identifiers would be obfuscated while the other features are hidden (if the object is identified by a set of its attribute values e.g. in our example language Fig. 1, where each object is identified by an *id* attribute). Hence, the correspondences between the objects of the original model and its secure view remains.

3.3 Definition and Applicaton of Rules

In [4], 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.

For example, Lst. 1 describes security rule `permitControl` to fulfill security requirement **R1**. Here `objectControl` represents a graph pattern in EMF-IncQuery [21] syntax that specifies a query against a wind turbine model, while `permitControl` is the rule that grants read and write operations to a heater control engineer on model facts identified by the query. In this case, a match consists of a single object `<ctrl>` of type `HeaterControl`. In the instance model of Fig. 2, the pattern has only one match: `<ctrl3>`. The rule permits the modification of the asset `<ctrl3>` by the engineer `HeaterCtrlEng`.

Listing 1 Query and Rule Definition

```
1  pattern objectControl(ctrl:Control)
2  { HeaterControl(ctrl);  }
3
4  rule permitControl allow RW to HeaterCtrlEng
5  { query: objectControl }
```

Lst. 2 introduces rules that fulfill the remaining security requirements without a detailed description of the queries as it is out of the scope of this paper. Security requirement **R2** is covered by rule `viewSignal` that would grant read permission for signals `s3`, `s4`, `s5` and `s6`. Rule `editSignal` would allow the editing of signals `s3` and `s4` described in security requirement **R3**. Rule `viewConsume` would permit the readability of references ref(ctrl1,consumes,s3), ref(c1,consumes,s3) and ref(c1,consumes,s4) to satisfy security requirement **R4**. Finally, rule `denyConfSignal` denies the visibility of confidential signals `s4` and `s6` required by security requirement **R3**.

Listing 2 Rule Definitions for the Example

```
1  rule viewSignal allow R to HeaterCtrlEng
2  { query: ... }
3  rule editSignal allow W to HeaterCtrlEng
4  { query: ... }
5  rule viewConsume allow R to HeaterCtrlEng
6  { query: ... }
7  rule denyConfSignal deny R to HeaterCtrlEng
8  { query: ... }
```

3.4 Consistency of Secure View

An arbitrary set of model facts does not necessarily constitute a valid model; there may be *internal consistency 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. Goal **G2** requires that secure models must be synthesized as a set of model facts compatible with all internal consistency rules.

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 required for access control.

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 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.

Opposite Features For reference types having an opposite, reference facts of the two types come in symmetric pairs.

Multiplicity Constraints Number of a specific references of an object needs to satisfy the multiplicity constraints.

3.5 Secure View of the Running Example

Fig. 2 contains markers to describe the filtered model presented to the heater control engineer. The user can modify the two objects marked with blue label and bold outline (`ctrl3` and `s3`), and can additionally see the signal with orange label and dashed outline (`s5`). Objects with white label and dashed outline are hidden (`ctrl2`, `s1`, `s2`, `s4` and `s6`) which implies that their attributes are unreadable and unmodifiable. Objects (`root`, `ctrl1`, `c1`, `c2`, `ctrl4`) that contain other visible objects, but must themselves be unreadable, appear with identifiers obfuscated *id* (marked with an O in a square), and their other attributes hidden (marked with an H in a square). Dashed edges are hidden references, thin edges are readable, while thick edges are writable.

4. PARAMETERIZABLE POLICIES

To meet goal G3, the policy language provides parameters to fine-tune the conflict resolution.

Default Read When none of the rules specify the read access of an asset, the *default read* value will be used. It can be *allow*, *obfuscate*, *deny*.

Default Write When none of the rules specify the write access of an asset, the *default write* value will be used. It can be *allow*, *deny*.

Conflict Resolution For conflict resolution, the strategy should decide between a *permissive* or a *restrictive* resolution. In case of *permissive* resolution, rule that allows an operation takes precendence over a denial rule. But, the *restrictive* resolution prioritizes denying rules over allowing ones.

Moreover, these values shall be specified on *global*, *user* and *root* levels to define different type of rules for each user and model. *Root* level settings are the most specific settings that associate the values with the root objects of a model. *User* level settings are connected to the current user and are applied when there are no root level settings for a root object. Finally, *global* level settings are the most general ones independent from the users and root objects and applied if there are no default values are specified. This means, *global* settings are mandatory, but *user* and *root* are optional.

For the running example, *user* specific default values are set as follows: read and write operations are denied by default and a restrictive conflict resolution will be used.

5. ANALYSIS OF CONFLICTS

5.1 Conflict types

Several rules can state contradict permission levels for the same assets. Moreover, the read and write operations of an asset may depend on other assets by consistency rules. These conflicts have to be resolved in order to obtain the conflict-free *effective permissions* which are actually enforced.

We classified the conflict types into 3 category represented in Table 1 based on the asset on which the rules are applied and the operation (*read,write*) they want to grant or deny.

Type I. A conflict of *Type I.* appears when two or more rules apply to the same asset and specify different permission levels for the same operation. For example, rule `viewSignal` allows the read operation of confidential signal `s4`, but rule `denyConfSignal` denies it.

Type II. A conflict of *Type II.* appears when two or more rules apply to the same asset but on a different operation. For example, rule `editSignal` allows the write operation on confidential signal `s4`, but rule `denyConfSignal` denies the read operation on confidential signal `s4`.

Type III. A conflict of *Type III.* appears when two or more rules apply to different assets that depend on each other (see Sec. 3.4). For example, rule `viewConsume` allows to see the consumes reference from composite module `c1` to confidential signal `s4`, but security requirement `denyConfSignal` denies the visibility of confidential signal `s4`.

Table 1: Conflict Categories

	Asset	Operation
Type I.	same	same
Type II.	same	different
Type III	different	-

Type I. Same Asset and Operation.

An asset cannot both be readable and hidden; neither can it be both writeable and unmodifiable at the same time. Naturally, when two or more different permission levels are applied to the same operation type (read/write) of the same asset, a conflict occurs.

Example. In our running example, rule `viewSignal` allows to read signal `s4`, but rule `denyConfSignal` denies to read it (depicted in Fig. 3).

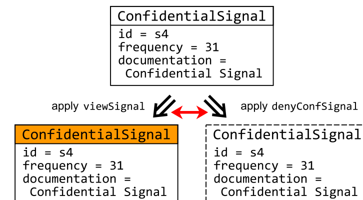

Figure 3: Type I. Conflict in the Example

Type II. Same Asset, Different Operation.

In case of different operations, we have to discuss the conflicting combinations of read and write permissions. As Table 2 shows, most of the cases are valid. On the other hand, if an asset is writable then it has to be readable while an unreadable asset cannot be writable. Moreover, giving write permission to an obfuscated asset is questionable. It should

mean that the users are only allowed to set obfuscated values which is not common in practice. Our approach works well regardless whether this combination is allowed or not.

Table 2: Compatibility matrix of access and operations types

		Read		
Write		Deny	Obfuscate	Allow
	Deny	✓	✓	✓
	Allow	✗	?	✓

Example. Back to our example, rule `editSignal` allows to write object `s4`, but `denyConfSignal` denies to read it (depicted in Fig. 4). Note that when the former rule dominates both the read and write permissions are granted.

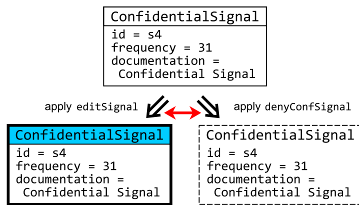

Figure 4: Type II. Conflict in the Example

Type III. Different Assets.

An *object* asset can be in conflict with its *attributes* and *references*. Moreover, we refine the list of assets introduced in Sec. 3.3 with *opposite*, *containment* and *identifier* assets.

An *opposite* asset is a pair of reference facts that are each others opposites. Both of them must have the same permissions. *Containment* asset is also a special reference asset defining parent and child relation between two objects. A chain of containment assets describes a containment hierarchy. *Identifier* asset is a set of attribute facts that identifies an object.

Read Dependency Readability of an object can be in conflict with one of its attributes when the attribute is readable/obfuscated but the object is unreadable.

In case of a reference, its source and target objects have to be taken into account. When a reference has to be readable, both of its source and target objects have to be readable too. Conversely, if either of the source and target object are unreadable references between them have to be invisible.

When an object asset o is readable its parent and the container reference have to be visible as well. This means that all the objects have to be visible in the containment hierarchy until o. Conversely, when a containment reference asset is invisible, none of its children should be visible.

In addition, if an object is visible all of its identifiers have to be visible, and conversely, if any of the identifiers is not visible, the object has to be invisible. Moreover, if an object is present in the filtered model solely to satisfy the dependencies of other assets (contained objects, reference links) and not because an explicit rule, it should be obfuscated.

If an object is explicitly obfuscated by the rules, or due to dependencies as explained above, its identifier attributes shall be obfuscated and other attributes need to remain hidden by default.

Finally, in case of a readable object, all of its features are also readable by default. However, this is merely a weak consequence: rules can specifically override this default for each feature.

Example. Application of rule `viewConsume` makes the *consumes* reference between control unit `ctrl1` and signal `s3` visible (depicted in Fig. 5). Signal `s3` is readable because of the application of rule `viewSignal`, but control unit `ctrl1` which is the source of the reference is invisible by default. Internal consistency constraints demand to make `ctrl1` visible. It also implies that the composite module `root` which is the container of `ctrl1` needs to be visible.

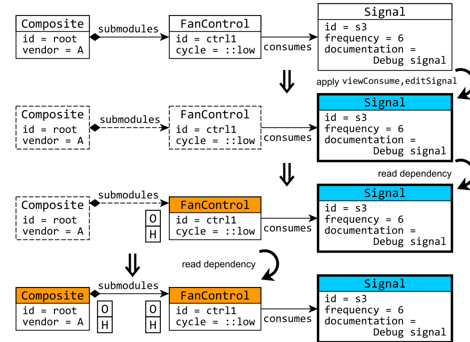

Figure 5: Type III. Conflicts - Read dependencies

Write Dependency Write permission means that the asset can be modified. In case of a writable/unwritable object, all/none of its features are writable by default. Once again, rules can specifically override this default for each feature.

When an object asset is writable it does not mean that it can be deleted. For that behavior, the containment reference that holds the object must also be modifiable. When a containment reference is writable, all of its children can be deleted (if they themselves are writeable) or moved to under another containment reference (if the other containment reference is also writable).

If an identifier is modified, it is equivalent to an object being deleted and a new one created at the same place. This also means that a containment reference is deleted and a new one is created. Thus, if any of the identifiers is writable the containment reference has to be writable. And if a containment reference asset is unmodifiable, none of the identifiers are writable.

Read vs. Write Dependency Writable assets have to be visible. Hence, we investigate conflicts between readable assets associated with the same object asset. Thus read and write dependent conflicts are special cases of resolving read dependencies.

Multiplicity Constraints The presented conflict resolutions handle the following internal consistency constraints: *object existence*, *containment hierarchy*, *opposite features* described in Sec. 3.3. Now, we investigate the *multiplicity constraints*. Multiplicity of a feature type defines the number of possible target values or objects and consists of a lower and upper limit ([*lower*]..[*upper*]). However, not all type of multiplicity constraints are supported by our approach.

0..* If there are no multiplicity restrictions, the presented approach works well without further considerations.

0..m This type of multiplicity constraint defines an unsettable feature (attribute or reference) may have up to m values per object, where m is a finite integer (typically 1). Applying read permissions may only decrease the number of model facts in the filtered model, thus this multiplicity constraint is met in the secure view regardless whether the value is readable, hidden or obfuscated. However, for a given object with one or more feature values that are invisible to the user (the security rules deny the read permission to the asset formed by the specific feature value), write access control must prevent the user from adding new values to the feature (even if the security rules would allow write permissions for model fact formed by the new value) if, together with the hidden values, they would exceed the upper limit in the unfiltered model.

1..1 This type of multiplicity constraint defines a feature where a value must always be set. If the user is not allowed to read the associated model fact, then an obfuscated (and unmodifiable) value must be provided so that the secure view is a consistent model. For instance, the identifier of a visible object cannot be hidden, only obfuscated.

k..m This rare type of multiplicity constraint defines a feature with at least $k \geq 1$ and at most $m > 1$ values per object, where m can be unlimited. It implies that at least k distinct values need to be visible to the user, but if fewer than that number are readable, there is no self-evident deterministic way for selecting a subset of hidden values for obfuscation. Therefore our approach is only applicable if no such feature values are hidden from the user when the host object itself is visible.

5.2 Formal treatment of conflicts

5.2.1 Ordering and connections

The set Π of priority classes is totally ordered using $>_\Pi$, i.e. from any two classes, one of them has higher priority. Each rule is assigned an (external) priority class, but multiple rules can be in the same class; in fact, we expect to commonly find policies where all query-based rules share the same priority class.

The permission levels for an operation are totally ordered according to permissiveness, i.e. $\text{Levels}_W = \{deny_W <_W allow_W\}$ and $\text{Levels}_R = \{deny_R <_R obfuscate_R <_R allow_R\}$. In case further refinements to the permission model is necessary, it would be fairly easy to extend our approach so that the ordered set is replaced with a finite lattice.

Detecting conflicts relies on the notion of *Galois connections.* For partially ordered sets A and B, a (monotone) Galois connection is an ordered pair $\langle f, g \rangle$ of monotone (order-preserving) functions, the lower adjoint $f : A \mapsto B$ and the upper adjoint $g : B \mapsto A$, with the property that $f(a) > b \Leftrightarrow g(b) < a$ (or equivalently, $f(a) \leq b \Leftrightarrow g(b) \geq a$). Intuitively, $f(a)$ puts an upper bound on B-values that are related to a in a certain way, while $g(b)$ is a lower bound for A-values that have this same connection with b.

A key observation is that the compatibility matrix (see Table 2) between the read and write permissions of a single asset encodes such a Galois connection; this is a consequence of the fact that an incompatible pair of read and write permission levels can always be made compatible either by choosing a more permissive read or a more restrictive write level. In other words, for each read permission level $p_R \in \text{Levels}_R$, there is a corresponding write permission level $\lceil p_R \rceil$ such that p_R is compatible with write permission level p_W if and only if $p_W \leq_W \lceil p_R \rceil$. Dually, for each $p_W \in \text{Levels}_W$, there is a $\lfloor p_W \rfloor$ such that p_W is compatible with a read permission level p_R iff $\lfloor p_W \rfloor \leq_R p_R$. Concretely, $\lceil deny_R \rceil = deny_W$, $\lceil allow_R \rceil = allow_W$, $\lfloor deny_W \rfloor = deny_R$, and $\lfloor allow_W \rfloor = allow_R$ or $obfuscate_R$ (the adjoint of obfuscation is left open by the Table). This will be useful for resolving type II conflicts.

5.2.2 Judgments

In the context of determining permissions for a given user, we treat lower bounds (e.g. "this object must be visible, at least in an obfuscated form") and upper bounds (e.g. "this object must not be completely readable, it may be obfuscated at best") on a permission level as separate judgements.

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, a pair of upper and lower bound judgments are added; the asset is given by the pattern match, and the operation, priority class and permission level are determined by the rule header. While not discussed in this paper, it is possible to have rules that impose e.g. an upper bound only.

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 (due to sanity or dependencies) between the judgments. Sec. 5.2.3 will discuss how conflicts are formalized.

5.2.3 Conflict detection

Conflict detection is the task of determining which judgments contradict which ones. For upper bound judgment $j_>$ and lower bound judgment $j_<$, we denote their conflict by $j_> \not\sqsupseteq j_<$. The symmetric relation $j \not\sim j' := j \not\sqsupseteq j' \vee j' \not\sqsupseteq j$ denotes a conflict where either of the two judgments can be the upper bound. Compatibility (the lack of conflict in either direction) is denoted by the symmetric relation $j \sim j'$.

Conflicts are detected in the following cases (excerpt), always between an upper bound and a lower bound judgment:

Type I. $\langle a, o, p, >, \pi \rangle \not\sqsupseteq \langle a, o, p', <, \pi' \rangle$ iff $p < p'$; i.e. occurs if there are contradictory judgments for the same asset and operation.

Type II. $\langle a, R, p_R, >, \pi_R \rangle \not\sqsupseteq \langle a, W, p'_W, <, \pi'_W \rangle$ iff $p_R <_R \lfloor p'_W \rfloor$ (equivalently $\lceil p_R \rceil <_W p'_W$ by the identity introduced above); i.e. occurs if the read and write judgments for the same asset are incompatible.

Type III. $\langle a, R, p_R, >, \pi_R \rangle \not\sqsupseteq \langle a', R, p'_R, <, \pi'_R \rangle$ for $a \neq a'$ iff one of the following cases is true:
(A) $a = obj(x, \tau) \wedge a' = ref(x, \tau', y) \wedge (p_R = deny_R \wedge p'_R \neq deny_R)$; i.e. a reference link asset is visible, while its source endpoint isn't.
(B) $a = ref(y, \tau, x) \wedge a' = obj(x, \tau') \wedge (p_R = deny_R \wedge p'_R \neq deny_R)$, where τ is a containment edge type; i.e. an object asset is visible, while the containment link that holds it isn't.
(C) etc. (See for Sec. 5.1 informal description of all cases.)

Note that the permission-related conditions in cases III/A and III/B can be rephrased as $(p_R <_R L^{<}_{(a,o),(a',o')}(p'_R))$, where the function $L^{<}_{(a,o),(a',o')}(.)$ is defined as

$$L^{<}_{(a,o),(a',o')}(p'_R) = \begin{cases} deny_R, & \text{if } p'_R = deny_R \\ obfuscate_R, & \text{otherwise} \end{cases}$$

The equivalence stands as there is no read permission level more restrictive than $deny_R$, only the lower branch of the function definition will lead to actual conflicts. We can also reason about the dependent asset based on its dependency asset; the statement is further equivalent to $(p'_R >_R L^{>}_{(a',o'),(a,o)}(p_R))$, where

$$L^{>}_{(a',o'),(a,o)}(p_R) = \begin{cases} deny_R, & \text{if } p_R = deny_R \\ allow_R, & \text{otherwise} \end{cases}$$

Note that the two newly introduced functions are distinguished by having different ordering directions in their superscripts; and that the subscript identifies first the asset-operation pair whose permission is restricted in that direction, and second the asset-operation pair whose permission level determines the threshold.

In this case, functions $L^{<}_{(a,o),(a',o')}(.)$ and $L^{>}_{(a',o'),(a,o)}(.)$ play a similar role than $\lfloor . \rfloor$ respectively $\lceil . \rceil$ do in type II conflicts. Thus for each dependency between asset-operation pairs, we have such a pair of functions that form a Galois connection.

For uniformity in case of type II conflicts, let $\lfloor p' \rfloor = L^{<}_{(a,o),(a,o')}(p')$ and $\lceil p \rceil = L^{>}_{(a,o'),(a,o)}(p)$. Likewise, introducing $L^{<}_{(a,o),(a,o)}(p) = L^{>}_{(a,o),(a,o)}(p) = p$ for type I conflicts, we can treat all three kinds of conflicts uniformly, using the mathematical foundation of Galois connections. Finally, for the cases where two assets have no dependency, $L^{<}_{(a,o),(a',o')}(.)$ and $L^{>}_{(a',o'),(a,o)}(.)$ are defined to take the most respectively least restrictive value, so that they will not indicate conflict for any permission levels.

Using this notation, the following three statements are equivalent:

- $\langle a, o, p, >, \pi \rangle \not\sqsupseteq \langle a', o', p', <, \pi' \rangle$
- $p < L^{<}_{(a,o),(a',o')}(p')$
- $p' > L^{>}_{(a',o'),(a,o)}(p)$

Essentially, conflict occurs if and only if the upper bound judgment is too restrictive to be compatible with the lower bound one, or equivalently, the lower bound is too permissive to be compatible with the upper bound.

5.2.4 Propagation of consequences

As a judgment imposes compatibility constraints on its dependencies, its consequences can be propagated and directly represented as additional judgments on foreign asset/operation pairs. Using the *propagated consequence* judgments, all conflicts can be transformed into type I conflicts.

For judgment $j = \langle a, o, p, \psi, \pi \rangle$ of bound $\psi \in \Psi = \{>, <\}$, with $\langle a', o' \rangle$ as a dependency, the propagated strong consequence judgment is defined as $j_{\langle a',o' \rangle} := \langle a', o', l, \psi, \pi \rangle$ where $l = L^{\psi}_{(a',o'),(a,o)}(p)$. By the equivalent forms of conflict stated above, it follows that for any $j' = \langle a', o', p', \psi', \pi' \rangle$, $j \not\sim j'$ iff $j_{\langle a',o' \rangle} \not\sim j'$, where the latter is a type I conflict, expressed on the same asset and operation.

Extending the above notion, is also possible to propagate *weak consequences* that encapsulate a default effect of a given judgement, not a necessary condition. For example, all contained elements of a visible object should also be made visible by default - unless another rule denies the read permission. A weak consequence does not inherit the priority of the original judgement, but is assigned a lower priority instead, and may be overridden by more dominant judgements without conflicting with the original judgement. It can be formally captured as $j^*_{\langle a',o' \rangle} := \langle a', o', l^*, \psi, \pi^* \rangle$ with $l^* = W^{\psi}_{(a',o'),(a,o)}(p)$ (where $W^{\psi}_{(a',o'),(a,o)}$ is not necessarily a Galois connection). We propose 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.

5.3 Domination of Rules

Computing the effective permission requires us to resolve the conflicts and provide a consistent secure view of the model. If there are conflicts, it is inevitable that some rules will not fully apply to all assets identified by their query, but rather will be overruled by other, more dominant rules on some assets. There are two ways to compare rules in order to determine which one will dominate, as explained below.

External Priority Rules may be decorated with external priority information, where the policy engineer directly expresses which rule shall dominate over which one.

Internal Priority In case of two rules belong to the same priority class, we can still compare them based on the permission level they apply. Between the permission levels, we defined a precedence order: *deny* < *obfuscate* < *allow*. Based on this precedence, either the most restrictive (min) or the most permissive (max) rule can be selected as dominant, as per preferences of the policy engineer.

Example. As we introduced in Sec. 3.3 a restrictive conflict resolution is used in the running example. In the conflict of Fig. 3, the *deny* permission level is more restrictive than *allow*, thus signal `s4` remains hidden from the secure view.

5.4 Formal treatment of conflict resolution

5.4.1 Resolution step

Conflicting rules may dominate each other either by external priority, or, if they share a priority class, by internal priority. Thus within each priority class for rules, either the more permissive or the more restrictive rule is considered dominant, by discretion of the policy engineer. For each priority class $\pi \in \Pi$, one of the bounds $\psi_\pi \in \Psi = \{>, <\}$

is given to indicate dominance; e.g. if ψ_π is $>$, restrictive judgments will dominate over permissive ones in the same equivalence class.

Thus using ψ_π and the Galois connections discussed in Sec. 5.2.1, we can now introduce the following *domination relation* between two conflicting judgment: if $j \not\sim j'$ for $j = \langle a, o, p, \psi, \pi \rangle$ and $j' = \langle a', o', p', \psi', \pi' \rangle$,

$$j \succ j' := (\pi >_P \pi') \vee (\pi = \pi' \wedge \psi = \psi_\pi)$$

Observations: (a) the domination relation introduces a partial ordering among judgments, (b) two conflicting judgment are related by this domination relation one way or the other, i.e. one of them is always dominant over the other.

A *resolution step* takes two judgments that are in a Type I conflict, compares them using the dominance relation, and modifies the dominated judgment to make it compatible with the dominant judgment. For $j = \langle a, o, p, \psi, \pi \rangle$ and $j' = \langle a, o, p', \psi', \pi' \rangle$ with $j \not\sim j' \wedge j \succ j'$ 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:

- j'' relaxes j', i.e. upper bounds are raised when replaced, while lower bounds are lowered.
- $j'' \sim j$ is guaranteed by the construction.
- The resolution step upholds the completeness of the permission set.

5.4.2 Resolution process

The *resolution process* (see Alg. 1) iterates over judgments in the order of domination, to propagate their consequences (also the weak consequences, unless contradicted by a previously processed judgement) and resolve any type I conflicts they are in.

Algorithm 1 The resolution process in pseudocode

```
  ▷ The policy is assumed as an implicit global parameter
function GetEffectivePermissions(model, user)
    permissionSet ← getInitialPermissions(model, user)
    processed ← ∅
    loop
        if permissionSet ⊆ processed then
            return permissionSet
        end if
        j ← chooseDominant(permissionSet \ processed)
        processed ← processed ∪ {j}
        for all dependencies ⟨a′, o′⟩ of j do        ▷ propagate
            conseq ← {j_⟨a′,o′⟩}
            if j′ ∈ processed : j*_⟨a′,o′⟩ ∼ j′ then        ▷ weak
                conseq ← conseq ∪ {j*_⟨a′,o′⟩}
            end if
            permissionSet ← permissionSet ∪ conseq
        end for
        conflicts ← typeIConflictsOf(j, permissionSet)
        for all j′ ∈ conflicts do               ▷ resolve locally
            j″ ← ResolutionStep(j, j′)          ▷ j dominates
            permissionSet ← permissionSet ∪ {j′} \ {j″}
        end for
    end loop
end function
```

We have the following success criteria against the resolution process, derived from the goals stated in Sec. 1.2:

Termination the process must eventually halt.

Correctness the process must yield a resolved permission set upon termination.

Confluence in order to work predictably and deterministically, the effective permissions must be completely determined by the user in question, the model and of course the policy.

The presented algorithm is terminating, as (for given assets) there is a finite space of possible judgments, each of which is processed at most once.

Once a judgment j has been processed, it will not be removed from the permission set anymore, as all judgments that could potentially dominate it have already been processed, and neither propagation nor resolution would create such dominating judgments anymore. It will not enter into new conflicts as the dominating party either, since all of its consequences have been propagated, so any judgment that would propagate a strong consequence that would be dominated by it, would never be processed, but would have previously been removed in a resolution step. Note that weak consequences are not propagated if they would create a conflict with an already processed judgement.

When a judgment is processed, its local (type I) conflicts are resolved; as it cannot enter new conflicts, there will be no more type I conflicts when the process terminates. This also means that there are no more conflicts of any type, since all strong consequences have been propagated. All the while the completeness of the permission set is maintained, thus the end result is a resolved permission set, and the process is correct.

Note that when processing a judgment, the associated propagations and resolutions yield deterministic results. Thus confluence requires that the end result is the same regardless of the order that judgments are selected for processing in `chooseDominant(.)`. The process has free choice only between judgments that do not dominate each other; in this case they are of the same priority class and bound, so the end result will contain all of their transitive consequences (with no domination between them), and conflicts will be confluently resolved to be compatible with the conjunction of all these judgments, independently of the order of steps.

5.5 Elaboration on the Running Example

The effective permissions calculation on the running example works as follows.

Initialization Before all, the initial permission set is determined. The default judgements are assigned to the model assets, which deny read and write operations for each element, as described in Sec. 5.4.2. Then each query-based security rule in the policy is evaluated on the model, and judgments to override the default values are created for the assets that are identified by query results.

- `permitControl` allows read and write operations for obj(ctrl3, Control).
- `viewSignal` allows read for obj(s3, Signal), obj(s4, ConfidentialSignal), obj(s5, Signal) and obj(s6, ConfidentialSignal).

- `editSignal` allows write operation for obj(s3, Signal), obj(s4, ConfidentialSignal).
- `viewConsume` allows read for ref(ctrl1, consumes, s3), ref(c1, consumes, s3) and ref(c1, consumes, s4).
- `denyConfSignal` denies read and write for obj(s4, ConfidentialSignal) and obj(s6, ConfidentialSignal).

We place all the above query-based rules in a single priority class π_1 (superior to both default permissions and weak consequences), and select the *restrictive* conflict resolution type specified in Sec. 3.3, i.e. $\psi_{\pi_1} =>$ so that upper bound judgements dominate.

As a result, judgements based on `denyConfSignal` take precedence above all other rules, and are picked first for being processed. The `viewSignal` and `editSignal` judgements on s4 and s6 are in type I conflict with these more dominant denials, and are thus overridden - the confidential signals remain hidden.

The denial judgements are propagated to dependencies: since s4 and s6 are hidden, any `provides` or `consumes` references pointing to them must also remain hidden. These propagated strong consequences have the same priority class and are also upper bounds on permissions, so they dominate over e.g. `viewConsume`. Thus these conflicts will be resolved next by relaxing the existing lower bounds, thereby hiding the references.

Next, judgements from the other five rules (except where overridden by the denial) are selected as dominant ones, and they propagate strong consequences. For instance, the visibility of ref(ctrl1, consumes, s3) has not been overridden by the denial of confidential signals, so its starting point ctrl4 must be present at least in an obfuscated form, thus the propagated strong consequence is a lower bound judgement of the same priority class. Similarly, s5 remained visible, so its containing object ctrl4 must be at least obfuscated, which in turn will propagate as additional lower bounds on the visibility of c2, c1 and finally the root. These propagated strong consequences will override the default judgement of hiding these objects. And as a consequence of having these objects visible but obfuscated, the identifier attributes (such as the `id` of c2) will also be visible with obfuscation; non-identifier attributes such as `vendor` remain hidden.

At the same time, as read access was granted to objects `ctrl3`, s4 and s5 (and not overridden by `denyConfSignal`), they propagate their weak consequence of making all their attribute values, contained objects and outgoing references visible, unless precluded by any rule (e.g. in case of s4). As `ctrl3` and s3 are also writable, their attributes are made writable by default as well.

Finally, the default judgements, where left intact, are processed. They neither override any judgements nor propagate meaningful consequences. For example, s4 kept the default permissions $deny_R$ and $deny_W$.

Discussions.

We think that the properties of the resulted secure model is worth to discuss. In the example, the heater control engineer can see only control unit `ctrl3`, signals s3 and s5. The rest of the objects are obfuscated or hidden, thus no confidential information is presented to the user. Furthermore, control unit `ctrl3` and its signal s3 are editable and removable. However, the user cannot delete these object in the current state of the model, because the *consumes* references pointing to s3 are only readable. Removing the control unit or the signal would also imply that the *consumes* references be removed from the model, but these operations are denied. Thus the specialist needs to collaborate with the other engineers and cannot delete on their own those elements from the model on which other users depend. On the other hand, our specialist can create a new heater control unit under composite c1, and the signal can be moved under this new unit (without deleting incoming references); this enables the specialist to delete the previous unit.

In addition, control unit `ctrl3` does not consume any signals in the model. Because of the fact that the control unit and all of its features are editable, the heater control engineer has the permission to create new *consumes* references in the model. Currently, the signal s5 is visible in the model, which means that the specialist can create a ref(ctrl3,consumes,s5) reference.

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 (judgements) 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 judgements, 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 [12] (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 [10], 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.

Context-aware Access Control RDF Stores. Models can be persisted into triples to store them in triple or quad stores (Neo4EMF[3], EMF Triple). Graph-based access control is a popular strategy for many RDF stores (4store [11], 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 [17], but resolution of contradicting rules are not discussed. Amit

Jain et. al. [15] 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+[20], VirtualEMF[6], WebGME[16]. The collaborative hardware design platform VehicleFORGE stores their model in graph-based databases and has an access control scheme TrustForge [7] that uses an implementation of KeyNote [5] 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.

7. CONCLUSION AND FUTURE WORK

In this paper, we have proposed formal foundations for describing the interpretation of rule-based access control policies in MDE. We have identified the necessary criteria (deterministic, conflict-free resolution) for successfully interpreting such policies. We have proposed a framework of conflict resolution processes that is guaranteed to meet these criteria, and yet be flexible enough so that it can adapt to the preferences of the the policy engineer. We have demonstrated the problems and an example solution using a case study inspired by a real-world industrial domain.

As future work, we (i) plan to define a textual syntax for the proposed security policy language that includes the defaults, the query-based rules and also the conflict resolution options, and investigate (ii) preselected sets of policy options (such as the resolution strategies of XACML) and accompanying "design patterns" on how policies should be constructed. Efficient incremental computation of the effective permission set also remains a challenge for the future.

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] Alessandra Bagnato, Etienne Brosse, Andrey Sadovykh, Pedro Maló, Salvador Trujillo, Xabier Mendialdua, and Xabier De Carlos. Flexible and scalable modelling in the mondo project: Industrial case studies. In *XM@ MoDELS*, pages 42–51, 2014.

[3] 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.

[4] Gábor Bergmann, Csaba Debreceni, István Ráth, and Dániel Varró. Query-based Access Control for Secure Collaborative Modeling using Bidirectional Transformations. In *ACM/IEEE 19th Int. Conf. on MODELS*, 2016.

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

[6] 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.

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

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

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

[10] 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.

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

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

[13] Dennis Hofheinz, John Malone-Lee, and Martijn Stam. Obfuscation for cryptographic purposes. In *Theory of Cryptography Conference*, pages 214–232. Springer, 2007.

[14] Rex Jaeschke. Encrypting c source for distribution. *Journal of C Language Translation*, 2(1):71–80, 1990.

[15] 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.

[16] 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.

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

[18] Sebastian Schrittwieser and Stefan Katzenbeisser. Code obfuscation against static and dynamic reverse engineering. In *Int. Workshop on Information Hiding*, pages 270–284. Springer, 2011.

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

[20] 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.

[21] Zoltán Ujhelyi, Gábor Bergmann, Ábel Hegedüs, Ákos Horváth, Benedek Izsó, István Ráth, Zoltán Szatmári, and Dániel Varró. EMF-IncQuery: An integrated development environment for live model queries. *Sci. Comput. Program.*, 98:80–99, 2015.

[22] VIATRA Project. VIATRA Model Obfuscator. https://wiki.eclipse.org/VIATRA/ModelObfuscator.

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