Runtime support for rule-based access-control evaluation through model-transformation

Salvador Martínez, Jokin Garcia, Jordi Cabot

▶ To cite this version:

Salvador Martínez, Jokin Garcia, Jordi Cabot. Runtime support for rule-based access-control evaluation through model-transformation. the 2016 ACM SIGPLAN International Conference, Oct 2016, Amsterdam, Netherlands. pp.57-69, 10.1145/2997364.2997375 . hal-02864664

HAL Id: hal-02864664
https://hal.science/hal-02864664v1

Submitted on 11 Jun 2020

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Runtime Support for Rule-Based Access-Control Evaluation through Model-Transformation

Salvador Martínez
Open University of Catalonia &
Inria, Mines Nantes, LINA
Nantes, France
salvador.martinez_perez@inria.fr

Jokin García
IK4-IKERLAN Research Center
Arrasate, Spain
jgarcia@ikerlan.es

Jordi Cabot
ICREA
Open University of Catalonia
Barcelona, Spain
jordi.cabot@icrea.cat

Abstract

Access-control policies, often the mechanism of choice to implement the security requirements of confidentiality and integrity, can be found in a wide range of application scenarios. Although there are standard languages for access-control and a plethora of works devoted to assure the well-formedness of access-control policies, little attention has been paid to the problem of providing robust and adaptable runtime evaluation engines for the integration of access-control in new DSL's and platforms. Indeed, the integration of access-control requires the development of critical infrastructure facilities around it, so that the policies can be: 1) analyzed and validated and 2) efficiently evaluated against run-time access requests.

In order to solve this problem, this paper explores the use of the already mature model transformation frameworks as modern, application-independent infrastructures for access-control languages i.e., following the Policy Enforcement Point(PEP)-Policy Decision Point(PDP) architecture. More specifically, we show how model-driven engineering and the ATL model-transformation framework can be used to lift the infrastructure development burden from developers by providing a robust, flexible and re-usable runtime evaluation engine for rule-based access-control policies.

Categories and Subject Descriptors D.2.2. [*Software Engineering*]: Design Tools and Techniques—Computer-aided Software Engineering; K.6.m. [*Management of Computing and Information Systems*]: Miscellaneous —Security

Keywords Access-control, Model-driven Engineering, Model Transformations, DSLs

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from Permissions@acm.org.

Copyright is held by the owner/author(s). Publication rights licensed to ACM.

SLE'16, October 31 – November 1, 2016, Amsterdam, Netherlands
ACM. 978-1-4503-4447-0/16/10...$15.00
http://dx.doi.org/10.1145/2997364.2997375

1. Introduction

Access-control policies, used as a means of securing applications w.r.t. the confidentiality and integrity properties, are a pervasive mechanism in current information systems, with many different models and languages used for its specification. However, while there exist standard access-control languages following different well-defined paradigms (e.g., Role-based access-control [27], Mandatory access-control [1]) and a plethora of academic works focused in assuring the well-formedness and correctness of policies, little attention has been paid to the runtime evaluation of such languages. While enterprise applications and frameworks with a large base of users usually implement and provide their own access-control facilities (e.g., the Java EE access-control, well-know Content-management Systems, etc.), i.e., language and runtime infrastructures, as a core component, this is not the case for smaller frameworks and new DSLs. Moreover, frameworks providing ready-to-use generic access-control with runtime evaluation engines are scarce, being notable exceptions the XACML[21] and EPAL[3] framework that are, however, model-specific and still not widely adopted.

The integration of an access-control mechanism in a given platform or technology requires however a critical runtime infrastructure around it. Notably, facilities to allow 1) the analysis and validation of policies, so that errors and anomalies can be detected without impacting the real, protected resources, 2) run-time evaluation of access-requests to protected resources as, effectively, access-requests need to be evaluated against the corresponding authorization policies before issuing access decisions.

Developing such an infrastructure is a costly, time consuming and error prone task requiring important testing and validation efforts as effectively, any error or performance issues in these infrastructure facilities may impact both the correctness of the authorization decision and the usability of the application. This situation is worsened when the access-control infrastructure is tangled with the application as it makes it very difficult to perform validation and testing

tasks. For this reason, one recommended approach [33][21] for the implementation of access-control frameworks is to separate the infrastructure logic from the application logic by using a reference monitor [33] architecture. It consists of two basic components: a Policy Enforcement Point (PEP) and a Policy Decision Point (PDP). Then, every request made by a subject is intercepted by the PEP and then forwarded to PDP for an access decision evaluation.

In this paper, we propose the use of model transformation frameworks as ready-to-use infrastructures able to lift the development burden from security designers and developers. Concretely, we show how the ATL model transformation language and framework[15] can be used as a core component to create application-independent infrastructure for access-control languages that: 1) follow the Policy Enforcement Point(PEP)-Policy Decision Point (PDP) architecture and 2) are able to adapt to multiple access-control languages and paradigms. Effectively, we claim that the problem of the evaluation and validation of access-control policies can be translated into a model transformation problem where the model transformation execution is used to derive an authorization decision. We believe that a decade of research and development in the field of model transformations makes current model transformation frameworks mature infrastructures able to take over the task. We demonstrate the feasibility of our approach by developing a prototype implementation for an attribute-based policy language.

Our approach first translates a given access-control policy to a model transformation by means of a Higher-Order Transformation (HOT) that is automatically derived from the policy metamodel and that includes the semantics of the access-control language. Then, it uses the transformation execution framework to launch the generated transformation on concrete access-requests to yield access-control decisions. Finally, a refining transformation implementing common rule combination algorithms is used in order to solve possible rule conflicts. Note that using our approach the development of a PDP is greatly reduced as all the pattern-matching, rule scheduling and conflict resolution is delegated to the model transformation framework (the latter as an extra refining transformation) reducing testing and validation efforts. Moreover, using model transformation frameworks as infrastructure presents several advantages: 1) Flexibility: policy, request and evaluation metamodels can be easily modified for adding special features like traceability or support to obligations and recommendations 2) Efficiency: advanced execution modes like incremental propagation and lazy and parallel evaluation are readily available and can be used transparently without modifying the transformation specification 3) Verification and Validation: results on model transformation correctness could be reused to verify that the policy translation and execution are correct.

The rest of the paper is organized as follows. Preliminary concepts are introduced in Section 2 whereas motivation is provided in Section 3. Section 4 describes our generic approach followed by a demonstration of its application to concrete technologies in Section 5. In Section 6 an evaluation of our prototype is provided. Related work is discussed in Section7. Finally, we conclude the paper in Section 8 by summarizing our contributions and discussing future work.

2. Preliminary Concepts

In order to ease the discussion, in the following we will introduce in detail the concepts of access-control and model transformation.

2.1 Access-control

Access-control [4, 28], often simply called *Authorization* or *Secrecy*, is a mechanism aimed at assuring that the information within a given Software System is available only to authorized parties. Therefore, access-control is used in order to assure two system properties: Confidentiality and Integrity, by controlling that only trusted entities modify or write the data.

Figure 1 shows the core concepts of access-control, namely Object (or Resource), Subject, Action, and Permission (or Privilege). They are described as follows:

Objects are normally passive entities within systems. They represent pieces of information such as files in operating system or tables in relational databases. Thus, objects in the context of access-control are any resource that can be accessed within a system. **Subjects** are the active entities in a system (Subjects can be users of more sophisticated entities like groups or roles). They represent the actors to which the access to *Objects* is controlled. **Actions** are any kind of access to the *Objects* that may be performed by the *Subjects* in a given system. From the classical C.R.U.D. (Create Read, Update, Delete) operations in database systems to sending a HTTP package in a network. **Permissions** relate *Actions* with *Objects*. A permission is thus the right to perform a given *Action* (or set of actions) on a given *Object* (or set of objects). These permission are, in turn, granted to *Subjects*. Summarizing, access-control is about granting or denying to *Subjects* in a system the *Permissions* to perform *Actions* on *Objects*.

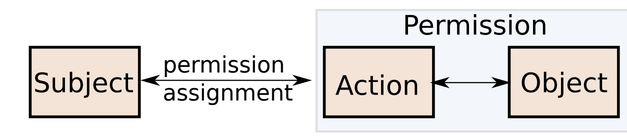

Figure 1. Access-control Core Concepts

However, directly assigning permissions to Subjects becomes unpractical when the user-base of the applications is large (as in the case of modern web applications) and thus, in real applications, the definition of the permissions and its assignation is often performed by using two concepts:

- **Rule:** A rule is the assignation (or denial) of a permission to a given subject. Generally, access control rules have the following form:

$$R_i : \{conditions\} \rightarrow \{decision\},$$

  where the subindex *i* specifies the ordering of the rule, *decision* can be accept or deny and *conditions* is a set of rule matching attributes like the source and destination addresses, holding roles, but also environment conditions like time, etc (note that there exist alternative, less-general models where attribute-based rules are used to the assignment of roles instead of permissions [2]).
- **Policy:** An access-control security policy is the set of permission assignations within a given information system, which is composed of a set of Rules. This policy constitutes a mere definition of the security requirements for the system, while the process of implementing the mechanisms to make the system follow the rules it defines is called enforcement.

Rule-based access-control simplifies the management of security policies. However, this comes at the price of introducing anomalies. In more detail, rule-based access-control policies may contain inconsistent rules, i.e., rules that for the same request yield different evaluation result. This problem is generally solved by using rule combination algorithms telling, out of a set of conflicting rules, which one to choose.

Finally, the core concepts of access-control described above can be organized in different ways and hold different meanings. Concretely, access-control policies are defined conforming to access-control models (or paradigms) that define the necessary elements to construct their contained rules along with their semantics. Due to the efforts that both the research and industrial communities have committed to the subject, diverse models have been proposed in the last decades where the most popular are Mandatory Access-Control (MAC) [1], Discretionary Access-Control (DAC) [1], Role-based Access-Control (RBAC) [27] and Attribute-Based Access-Control [34]. Notice that although we will focus here on attributed and rule based access control, the approach we propose can be applied to policies following any of the aforementioned paradigms.

2.2 Model Transformations

Model transformations are at the core of model-driven engineering by providing the means for automating the manipulation of models. A model to model transformation (M2M) transforms a model M_a conforming to a metamodel MM_a into a model M_b conforming to metamodel MM_b where MM_a and MM_b can be the same or different metamodels. While this kind of transformation can be implemented by using general purpose languages, model-transformation languages and frameworks exist in order to ease its specification by providing facilities to efficiently query and manipulate models.

In some model transformation languages, for example the QVT-based languages [26], a model transformation is itself a model, that is, it conforms to a metamodel which is part of the model transformation language's definition. This facilitates the definition of Higher Order Transformations (HOTs i.e. transformations which have other transformations as input and/or output).

There exist many different model transformation languages and paradigms with different features. Here we will focus on rule-based, declarative and non recursive model transformation languages such as QVT, ATL or ETL[18].

3. Runtime Evaluation Support: Requirement of an Efficient and Flexible PDP for DSLs

Let us consider the integration of access-control into a given application scenario. While enterprise applications and frameworks with a large base of users usually implement and provide their own access-control facilities, i.e., language and runtime infrastructures, as a core component, this is not the case for smaller applications and DSLs. In the same sense, there are no widely adopted access-control runtime infrastructures ready to be re-used. As a consequence, in order to integrate access-control, several different tasks need to be performed by the security designer or developer. Considering a PEP-PDP architecture and workflow as the one depicted in Figure 2, the tasks to be performed are as follows:

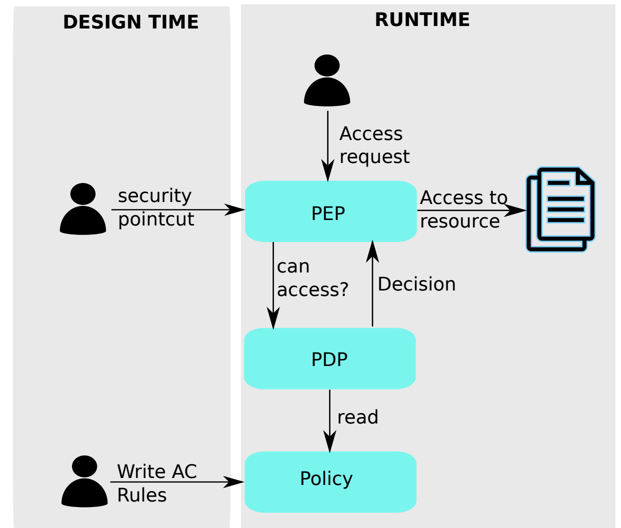

Figure 2. PEP-PDP Architecture and Workflow

1. Definition of the abstract and concrete syntax of the languages used to specify access-control policies and access-requests.

2. An infrastructure allowing the evaluation of access-control requests against access-control rules written in a given policy, i.e., a PDP. Developing and testing efficient pattern-matching, rule scheduling and conflict resolution facilities would be part of this task.
3. An interface mechanism between the host application and the access-control infrastructure (i.e., the PEP), so that access to protected resources is intercepted (i.e., access-control is enforced).

Since access-control languages are small, the first step can be considered a relatively simple task. Abstract syntax may be as simple as a database schema and concrete syntax may be just a web form. When not done manually, the third step usually deals with some sort of aspect weaving [10, 23] where certain operations are *marked* as secured. Therefore, the most complex and critical development effort lies in the second task. Building a PDP is a costly operation that can easily become prohibitive. Being a critical component by its very nature (unintended information disclosures may lead to important losses in terms of money and reputation), the development of such infrastructures would require an intense effort for assuring the current behaviour of its core components. Thus, in an ideal situation, it would be desirable to have re-usable and flexible access-control infrastructures ready to be used in new applications scenarios. Unfortunately, that is not the case. The aforementioned facilities of enterprise applications are rarely open nor re-usable, and moreover, they are specially tailored to specific domains, so that they are either not adaptable to a new environment, or as costly as implementing the infrastructure from scratch.

Furthermore, for a real seamless integration into small frameworks and DLSs a number of more specific requirements must be fulfilled:

- The access-control framework should provide ease-to-use concrete syntax to improve usability. Moreover, concrete syntaxes should be easily pluggable and adaptable without changing the rest of the components.
- New constraints, beyond those imposed by the access-control paradigm of choice should be easy to add. Moreover, validation and verification of the policy but also of its evaluation should be easy to perform by using readily available tools and techniques.
- The policy representation used by the evaluation engine must be explicit and flexible, so that extra features and adaptations to new requirements can be performed.
- It should be possible to adapt the evaluation execution. Examples of such adaptation includes: 1) the generation of customized traceability information for the decision process 2) the enhancement of the evaluation outcome, so that recommendations and obligations are supported. 3) the addition of new rule combination algorithms.
- The evaluation framework must be efficient and adaptable to complex scenarios without requiring changes in other components nor requiring the security designer or developer to deal with this extra complexity. Indeed, scenarios where a policy requires live evaluation are not rare (i.e., a scenario where the access-request and corresponding access-decision need to be constantly synchronized) and thus, must be transparently supported.

4. Approach

In order to solve the aforementioned problem, in this paper we explore the use of explicit models and model transformation frameworks as core components to provide generic, flexible and ready-to-use PDPs that 1) can be automatically derived from the policy model in the general case 2) are easily customizable for different scenarios.

Indeed, similarly to model access-control policies, a model transformation function $Mt : \{SourceModel\} \rightarrow \{TargetModel\}$ taking as an input a source model and producing a target model can be seen as composed by rules of the form: $Mt_{ri} : \{Match \times Conditions\} \rightarrow \{Output \times BindingValues\}$ where a *Match* is a source model element, *Conditions* are a set of guards that must hold for the rule to fire, *Output* is a target model element and *BindingValues* are initialization values.

When the element to match and the element to produce for a rule are always of the same type (in our case, and as discussed in the next section, of type *Request* and *Result* respectively), this function can be simplified to be $Mt_{ri} : \{Conditions\} \rightarrow \{BindingValues\}$. If we make *BindingValues* contain the values of an access decision (e.g., {accept, deny}), we have that the evaluation of access-control rules can be seen as a specific case of model transformation where input elements are always of the same type and the output model element contains a decision value.

As model transformation frameworks are mature tools with a large amount of research and development work committed to them (e.g., ATL or ETL [18]), they are ideally placed to take over the task of evaluating access-control policies as a special case of their natural range of use.

Following our approach the security developer obtains for his application a mature and efficient PDP infrastructure without the need of dealing with pattern-matching, rule scheduling and conflict resolution. Moreover, she may transparently benefit from a wide range of advanced features studied over transformation languages as incremental propagation [14], lazy [30] or parallel evaluations [31] or transformation correctness verification [7], [9], [8]. The use of model as an explicit representation of all involved artifacts also present advantages, as the possibility of adding domain-specific constraints to the policy [19], [12].

Note that we consider the problem of integrating PEPs into host platforms (and the translation of external events and calls to corresponding access-control requests) as already

solved in the literature and thus, we will not deal with it here and refer to the mentioned related work.

This section is devoted to describe our approach to obtain a PDP infrastructure based on model-driven engineering and the execution of model transformations. Our approach can be considered as having two different workflows: the basic workflow as seen by the developer that wants to integrate access-control into his platform without requiring a particular access-control model or concrete syntax, and the advanced workflow that is intended to provide a higher customization degree for all involved components.

4.1 Basic Workflow

The basic workflow, depicted in Figure 3, implies a transparent integration of access-control facilities. To take advantage of our framework, the platform or DSL developer only needs to 1) specify the security policy for their resources by using a given access-control concrete syntax. 2) Issue access-request, again using a given syntax, and finally 3) retrieve the access decision and act accordingly.

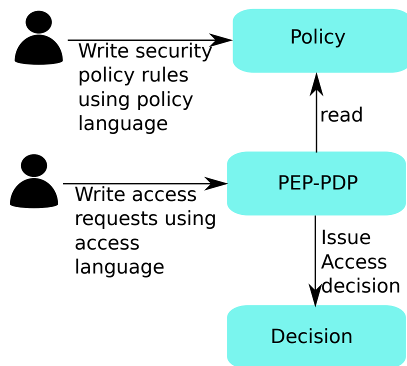

Figure 3. Basic Workflow

We will illustrate this integration with an example. Having two roles on the system (*Employee* and *Administrator*) and with the aim to protect a resource *O1* w.r.t. a *read* action, the security responsible will write the security policy rules shown in Listing 1 (note that Listing 1 and Listing 2 use a concrete textual syntax we provide for the metamodels we describe in Section 5). Then, this security policy will be automatically and transparently translated into a model transformation specification as the one showed in Listing 3 (the listing shows an ATL transformation. Further details about it will be given in the following sections).

Listing 1. Rule Example

```
Rule r1 (
    Subject S1 {attributes <'role' = 'Manager'>},
    Object O1,
    Action Read
) -> Accept

Rule r2 (
    Subject S2 {attributes <'role' = 'Employee'>},
    Object O1,
    Action Read
) -> Deny
```

Then, it will have to add interception of access to the protected object so that it can be passed to the PDP to obtain a decision. Listing 2 shows the syntax used to pass access-request to the PDP. Upon reception of this access-request, our PDP will execute the model transformation in Listing 3 and issue and access-decision. Note that in the presence of conflicts, the output of the *transformation specification* becomes input to rule conflict resolution algorithms (the choice of conflict resolution algorithm is the same used for the original access-control policy, and thus, can be established beforehand.) so that an unique decision is delivered.

Listing 2. Access Request Example

```
Access(
    Subject S1 {attributes <'role' = 'Manager'>},
    Object O1,
    Action Read
)
```

Finally, access decision would need to be managed by the developer accordingly to its needs.

4.2 Advanced Workflow

We have seen how our framework can be integrated in a given platform or DSL to obtain generic access-control support by translating access-control policies into model transformation specifications. In the following we will explain the internals of the basic workflow, i.e., how the artifacts it uses are obtained and thus, how they can be customized for specific requirements. The workflow, summarized in Figure 4, is composed of four steps.

The first two steps, namely, the injection of the original policy into a model and the generation of an equivalent model transformation specification are language-dependent. The third step, namely the refinement of the evaluation results is independent from the language and thus, can be reused for any access-control language.

4.2.1 Language-dependent Tasks

As mentioned above, the first two steps of our approach are language dependent and thus, they need to be performed for each new access-control language. Note however that, as access-control paradigms do not differ greatly, it is possible to reuse previously developed artifacts as generic components. We will show how that is achieved in section 5 by providing a generic access-control language and corresponding artifact generators.

1. As a first step, our approach requires to inject the original policy specification into a model. For this purpose, a *metamodel of the policy specification language* and a way to populate instances from policy samples is required. In many cases the metamodel may be readily available and the injection step could be simplified by the use of generative language frameworks, like XText [5], or reduced to perform database queries. We will also need to

Listing 3. Generated Transformation Policy

```
Module SecurityPolicy
create OUT: Evaluation from IN: Request;

nodefault rule rule1 {
  from
    s:Req!Requests (
        s.filter(Sequence{Tuple{id='S1', attributes = Sequence{Tuple{name = 'role', value = 'administrator'}}},
          Tuple{id='O1', attributes = Sequence{}},
          Tuple{id='read', attributes = Sequence{}}}))
  to
    t : Evaluation!Evaluation (
      effect <- 'Permit',
      ruleId <- 'Rule1',
      ruleOrder <- 1
    )
}
nodefault rule rule2 {
  from
    s:Req!Requests (
        s.filter(Sequence{Tuple{id='S2', attributes = Sequence{Tuple{name = 'role', value = 'employee'}}},
          Tuple{id='O1', attributes = Sequence{}},
          Tuple{id='read', attributes = Sequence{}}}))
  to
    t : Evaluation!Evaluation (
      effect <- 'Deny',
      ruleId <- 'Rule2',
      ruleOrder <- 2
    )
}
```

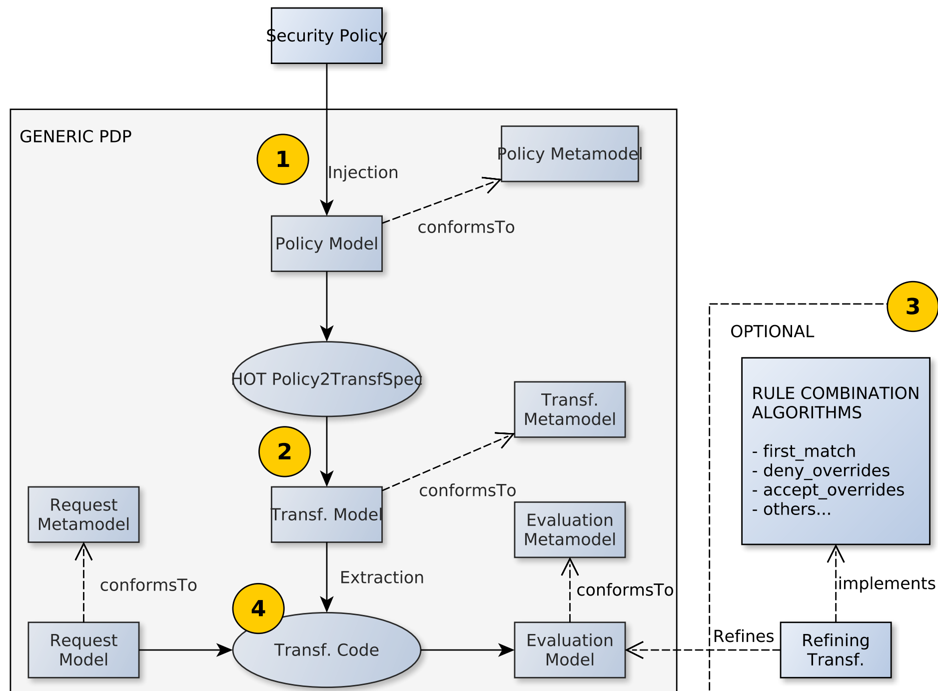

Figure 4. Approach

perform the injection step for the access-control request if they are not directly specified as a model. However, attribute-based access request are very generic and thus, the requests metamodel we describe in Section 5 could serve a large variety of access-control languages so that we can consider it language independent.

2. Once the original policy specification is represented as a model, we can proceed to the second step, namely, the generation of the corresponding model transformation

specification. This transformation specification is automatically generated by a HOT (see Policy2TransfSpec in Figure 4) that is, in turn, automatically derived from the metamodel specification of the access-control language at hand by the means of a Model-to-text transformation. It is thus adaptable to metamodel variability. Moreover, while the generated HOT may require some manual adaptation for some very domain specific access-control languages and models, most of it will remain re-usable.

Once the HOT is available and together with the previous injection step, policies written in the original access-control language will be transparently translated to transformation specifications for any application.

4.2.2 Language-independent Task

While the previously described steps need to be performed for each new access-control language, the third step use generic models and thus, 1) it is language-independent and 2) only need to be implemented once. Then, it is reused for any access-control language. Concretely, when the injection and HOT generation steps are ready, the refinement of the evaluation result to eliminate possible conflicts is fully reusable.

As introduced in Section 2, access-control policies may contain inconsistent rules, i.e., rules that for the same request yield different evaluation result. This problem is generally solved by using rule combination algorithms telling, out of a set of conflicting rules, which one to choose. Our approach takes this situation into account and includes an extra refining step as an in-place model transformation. Dashed line box of Figure 4 summarizes this step.

Finally, the elements created in the aforementioned steps are then combined in order to provide support for the basic workflow explained in 4.1.

5. Building a PDP Infrastructure with ATL

In the following we detail our generic approach for the specific case of implementing a PDP for a rule and attribute based access-control language by using the ATL transformation language. The choice of ATL as a concrete technology for our prototype derives from its popularity among the scientific and industrial communities (ATL being the de-facto standard and arguably one of the most used model transformations languages). We choose a generic rule and attribute based access-control language because of its flexibility (attributed access-control have been demonstrated to be able to emulate other access-control models like MAC or DAC [13]) and applicability to real world applications where rule-based access control with mixed positive and negative logic is often required. Notice that although we demonstrate our approach over ATL and a generic rule and attribute based access-control language, our approach is generic and both ATL and the access-control language could be substituted for other languages of similar features without requiring modifications to the general process.

5.1 Language-dependent Tasks (Steps 1 and 2)

In the following, we describe the two language-dependent steps that need to be performed when a new access-control language is integrated in an application by using our approach. Note that when an existing language infrastructure as the one we describe here (metamodel plus parsers) is readily available to the software developers and designers, these two steps are not required.

5.1.1 Step 1. Policy Metamodel and Injection:

The first step of our approach supposes a mere translation between technical spaces. We intend to provide the means to automatically pass from the technical space of the concrete policy representation (in our case a textual syntax as showed in Listing 1 and as it is common in real applications) to modelware, so that we can use it as an input to the transformation specification step. As mentioned above, we will work here with a rule and attribute based access-control language that follows the ABAC paradigm.

In Figure 5 we show the simplified conceptual model of our policy language (note that for simplicity, we choose to provide our own policy language, however, other generic rule-based policy languages as the one described in [23] could be used in its place). In this language a *Policy* contains a number of access-control *Rules*. These rules are composed of a left-hand side *LHS* and a right-hand side *RHS*. The left-hand side is meant to be used to express a number of conditions for a given access-control rule to be fired. This conditions are represented as *ConditionFields*. Each *ConditionField* may have *Attributes* that will further describe it. *Attributes* have a type and a value. We provide our language with three specific *ConditionField* elements, namely *Subject*, *Object*, and *Action* but other more-specific *ConditionField* elements can be easily added to the language if needed. The right-hand side is used to express the effect the application a *Rule* has by means of a number of *DecisionField* elements. We provide our language with three kinds of *DecisionField* elements: *AccessDecision*, *Obligation* and *Recommendation*. We focus here in *AccessDecisions* that represents upon an access request, an answer with value equal to permit, deny or not applicable. As in the case of *ConditionField* elements, the language can be easily extended to add other more-specific *DecisionFields*.

We have used EMF[29] for the creation of our policy metamodel. Then, we have created the grammar listed in Listing 4 to provide the language with the concrete textual syntax showed in listing 1 (other concrete syntax's may be provided). We have fed this grammar to the Xtext framework together with the metamodel to automatically obtain a language parser, an injector and the corresponding editors. From this point, users can write access-control policies using the concrete textual syntax while getting them transparently

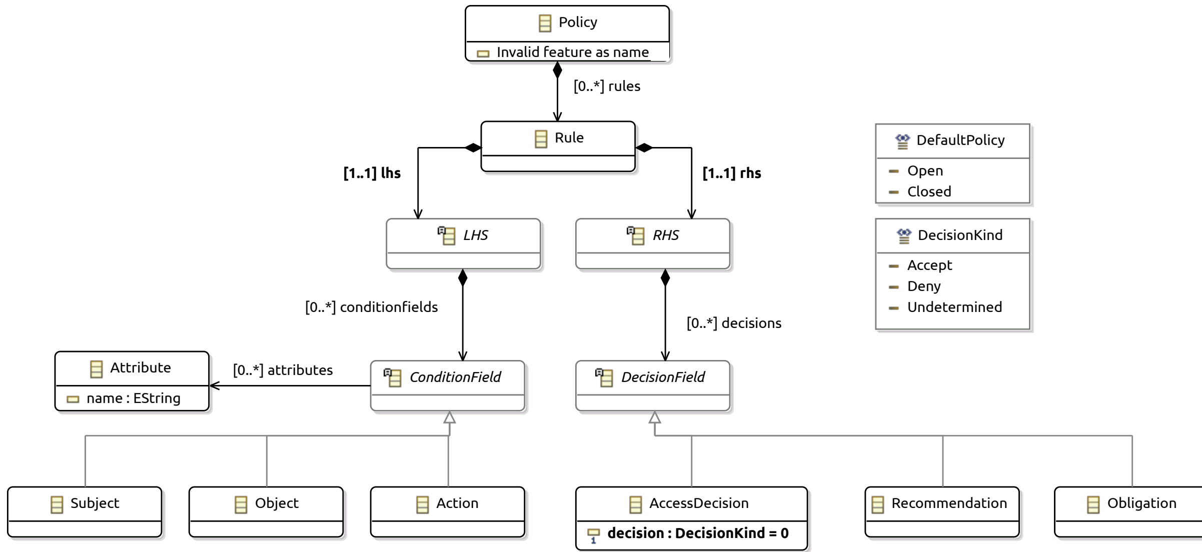

Figure 5. Rule and Attribute Access-control Policy Metamodel

Listing 4. Policy Language Grammar

```
Policy:
   rules+=Rule*;

Rule returns Rule:
   ’Rule’ id=ID ’(’ lhs=LHS ’)’ ’->’ rhs=RHS;

LHS returns LHS:
   conditionfields += ConditionField (","
        conditionfields += ConditionField)*;

RHS returns RHS:
   decisions+=AccessDecision;

ConditionField:
   (Subject | Object | Action)
   (
   ’{’(’attributes’ ’<’ attributes+=Attribute ( ","
        attributes+=Attribute)* ’>’ )?
   ’}’)? ;

Attribute returns Attribute:
   {Attribute} name=EString ’=’ value=EString;

Subject returns Subject:
   {Subject} ’Subject’ id=ID;

Object returns Object:
   {Object} ’Object’ id=ID;

Action returns Action:
   {Action} ’Action’ id=ID;

AccessDecision returns AccessDecision:
   decision=DecisionKind;

enum DecisionKind returns DecisionKind:
   Accept = ’Accept’ | Deny = ’Deny’ | Undetermined =
        ’Undetermined’;
```

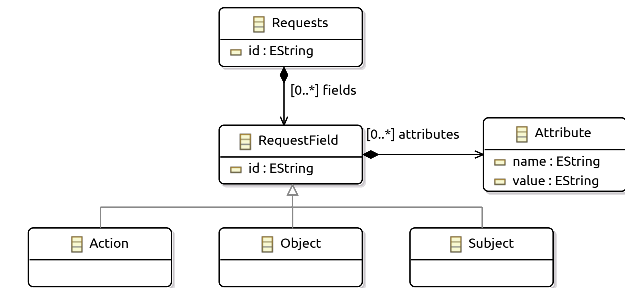

Figure 6. Request Metamodel

translated to models so that they can be used in the following steps.

We need to perform the same steps to provide support to the specification of access-control requests. We provided a metamodel for access-requests requests (see Figure 6) that is inspired on the one provided in the XACML specification. It consists basically in a *Request* element containing the elements identifying the access request, namely, *Subject*, *Resource* and *Action*. These elements can hold *Attribute* elements, that represent additional characteristics (e.g., the role of the subject). We would like to remark that this metamodel is generic and can be reused for other access-control languages.

For brevity we will not describe the grammar we provide for this metamodel but it would allow us to write (and get directly parsed as a model) requests as the one showed in Listing 2.

5.1.2 Step 2. HOT

In our approach, access requests will be evaluated by executing an ATL model transformation representing the origi-

nal access-control policy. In order to do so, we need first to obtain such a model transformation.

Let us consider a policy represented in Listing 1. From that representation, we want to obtain an ATL transformation module showed in Listing 3. The input element of this rule is an access-request element (the *requestFilter* call in the guard references a Helper used to encapsulate OCL expression and make the rule more readable) and the output is an Evaluation model element.

The access-request input model element conforms to the **Request metamodel** excerpt in Figure 6. The Evaluation model element corresponds to our **Evaluation metamodel** that basically stores a decision for each fired rule and traceability information (notably, rule ID and order) that will be used later for the refining step.

As model transformations are also models and thus, can be input and/or output of another model transformation, we can automatically generate the model transformation by using a HOT. This HOT takes a policy conforming to our Policy metamodel as input and produces the desired ATL model transformation specification as output.

Listing 5. Model-to-text HOT Generation

```
1
2 [module generate('http://www.eclipse.org/emf/2002/
      Ecore')]
3 [template public generateElement(policy : EClass) ?
4 (name = 'Policy')]
5 [comment @main /]
6     [file ('hot.atl', false, 'UTF-8')]
7     module Policy2ATL;
8     create OUT: ATL from IN: Policy;
9
10    rule Module {
11    ...
12    ...
13
14    [for (r : EStructuralFeature | policy.
          eAllContents(EStructuralFeature))]
15            [if (r.eType.name = 'Rule')]
16                [policy.generateMatchedRule(r.eType
                      .oclAsType(EClass))/]
17            [/if]
18    [/for]
19
20    [policy.generateFilterHelper(self)/]
21    [/file]
22 [/template]
23
24 [template public generateMatchedRule(rule : EClass)
      ]
25    rule Rule2MatchedRule {
26    from
27        s : Policy!Rule
28    to
29        mr : ATL!MatchedRule (
30            name <- s.ruleId
31        ...
32        ...
33 [/template]
34
35 [template public generateFilterHelper(Policy :
      EClass)]
36 ...
37 [/template]
```

However, the metamodel of ATL is big and thus, writing a HOT transformation is a tedious and relatively complex step (see Listing 6) where most of the effort will be devoted to creating the right structure for the future model-transformation and not with access-control issues. For that reason, in our approach the HOT transformation is automatically derived from the Policy metamodel by the means of a model-to-text transformation.

This way we have a double generative process. First, a model-to-text transformation automatically generates a HOT from the Policy Metamodel, and then, the HOT automatically generates transformation specifications out of concrete access-control policies.

Listing 5 presents an excerpt of the Acceleo[24] template used to generate the HOT. As can be seen, it uses the metaclasses of the Policy Metamodel to generate ATL code (see lines 3, 24, 35 where templates are defined on EClass and lines 16 and 20 where templates are called on concrete elements). Note that this *reflexive* way of working adds flexibility to our approach. As an example, the concept of role that until now has been considered just as another attribute to the condition field Subject can be consolidated as a separated condition. Thus, it would appear explicitly in the list of conditions of the rule and hold attributes itself, what would allow to express role hierarchies, etc. This can be achieved transparently by only adding a metaclass Role to our metamodel.

Finally, Listing 6 shows an excerpt of the generated HOT in charge of producing an ATL policy transformation specification (due to space limitations we only show the rule in charge of producing ATL matched rules from Policy rules). Basically, the HOT works as follows:

1. It matches the policy root element of the source access-control language and produces a new transformation specification that takes an access-control request as input and produces an evaluation model as target. In our use case, it matches the *Policy* metaclass and produces an ATL transformation module.
2. A *requestFilter* helper is created in order to factorize the encoding of access-request's conditions (Subject, Resource and Action) into OCL predicates (we choose to use a helper instead of directly adding the OCL in the ATL rule guards for the sake of readability). This helper will be then called from the rule guards. For simplicity, the current version of our prototype deals only with *Attributes* of type *String* as so does our Request Filter that basically performs string comparison on attribute values. It can be easily extended to support 1) other types, 2) the use of a Policy Information Point (PIP) to retrieve attributes not specified in the Request. We left those extensions as a future work.
3. Each access-control rule (*Rule* metaclass) is matched in order to produce transformation rules as the ones showed in Listing 3. The HOT rule *Rule2MatchedRule* matches the *Rule* metaclass and produces an ATL rule with all its required elements: input Pattern, guard, output pat-

tern and initialization bindings. Notice that, by default, an ATL transformation module cannot contain two rules matching the same elements what clearly differs from the natural behavior in access-control languages where having rules matching the same set of request is common (and where conflicts are solved afterwards by running rule combination algorithms). To solve this issue, we have implemented our HOT to produce *nodefault* ATL rules. This is a special, not very well-know, feature of ATL allowing for the creation of rules that may match input elements matched by other rules. While this will deactivate the standard tracing and resolution algorithms of ATL it will not impact the usage of our transformation in any way as the first can be easily simulated by creating explicit traces and the second is not needed as our output model does not have a containment hierarchy. General properties of termination and confluence are not impacted either (in the absence of recursive helpers, termination is guaranteed as input models remain read-only. Confluence is assured as generated ATL access-control rules are not inter-dependent).

Listing 6. HOT Generating Rules of Listing 3

```
1  rule Rule2MatchedRule {
2  from
3      s : Policy!Rule
4    to
5      mr : ATL!MatchedRule (
6        name <- s.id,
7        isNoDefault <- true,
8        isAbstract <- false,
9        isRefining <- false,
10       inPattern <- ip,
11       outPattern <- op
12     ),
13     -- start from part
14     ip : ATL!InPattern (
15       elements <- Sequence{ipe},
16       filter <- filter
17     ),
18     ipe : ATL!SimpleInPatternElement (
19       varName <- 's',
20       type <- ipet
21     ),
22     -- start filter
23     filter: ATL!OperationCallExp (
24       operationName <- 'filter'
25     ),
26     fvar: ATL!VariableExp (
27       referredVariable <- ipe,
28       appliedProperty <- filter
29     ),
30     fseq: ATL!SequenceExp(
31       parentOperation <- filter
32     ),
33     fsub: ATL!TupleExp (
34       collection <- fseq
35       --retrieve subject attributes
36     ),
37     fobj: ATL!TupleExp (
38       collection <- fseq
39       --retrieve objetc attributes
40     ),
41     fact: ATL!TupleExp (
42       collection <- fseq
43       --retrieve action attributes
44     ),
45     --end filter
46     ipet : ATL!OclModelElement (
47       name <- 'Request',
48       model <- om
49     ),
50     om : ATL!OclModel (
51       name <- 'Request'
52     ),
53     --end from part
54     --begin to part
55     op : ATL!OutPattern (
56       elements <- Sequence{ope}
57     ),
58     ope : ATL!SimpleOutPatternElement(
59       varName <- 't',
60       type <- opet,
61       bindings <- Sequence{b1, b2}
62     ),
63     opet : ATL!OclModelElement (
64       name <- 'Evaluation',
65       model <- om2
66     ),
67     om2 : ATL!OclModel (
68       name <- 'Evaluation'
69     ),
70     --begin bindings
71     b1 : ATL!Binding (
72       propertyName <- 'effect',
73       value <- se1
74     ),
75     se1 : ATL!StringExp(
76       stringSymbol <- s.rhs.decisions.first().decision
           ↪.toString()
77     ),
78     b2 : ATL!Binding (
79       propertyName <- 'RuleId',
80       value <- se2
81     ),
82     se2 : ATL!StringExp(
83       stringSymbol <- s.id
84     )
85     --end bindings
86     --end to part
87   }
```

5.2 Language-independent Task (Step 3)

As introduced in Section 4, access-control policies may contain contradictions which are typically resolved by the means of rule combination algorithms. Examples of such algorithms are: *first-match*, where the first triggered rule is selected, *permit-overrides* where a rule granting access have higher precedence that one precluding it, *deny-overrides* with the opposite behaviour, or others meant to provide default behaviour as *deny unless permit*.

As our evaluation metamodel contains tracing information such as rule identification and rule-ordering, the aforementioned algorithms can be easily implemented as a refinement transformation over an evaluation model that filters out the evaluation elements for which the algorithm condition does not hold. Moreover, as the algorithms will be defined over our generic evaluation model, they can be used, once implemented, for solving conflicts in any access-control language. As an example, we have implemented the first-match algorithm as a refining ATL transformation (see Listing 7). It defines the first match condition as a helper and then uses it in the rule application guard so than only the right evaluation is kept.

Listing 7. First-match Algorithm

```
1  create OUT: Evaluation refining IN: Evaluation;
2
3  helper context Evaluation!Evaluation
4  def:isFirstMatch():Boolean =
5    let allEvaluations :
6      Sequence(Evaluation!Evaluation) =
7      Evaluation!Evaluation.allInstances()
8      ->asSequence() in
9    allEvaluations->iterate(p; y : Boolean = true |
10     if p.request.toInteger() <
11     self.request.toInteger() then
12       false
13     else
14       if y = true then
15         true
16       else
17         false
18       endif
19     endif);
20
21 rule Evaluation {
22   from
23     s : Evaluation!Evaluation(s.isFirstMatch())
24   to
25     t : Evaluation!Evaluation ()}
```

5.3 Runtime Evaluation (Step 4)

The previous steps had made available: 1) the means to inject access-control policies (and requests) conforming to our rule and attribute based language into models; 2) a (generated) model transformation specification semantically equivalent to the original access-control policy; and 3) a refining transformation for the resolution of conflicts. We are ready to start evaluating access-control request.

The Workflow that needs to be performed to achieve this goal can be seen in Figure 7. It consists in 5 steps: 1) create an access-request (by using the concrete syntax available to the user); 2) inject the access-request into a model conforming to our Request metamodel; 3) Launch the ATL engine with the policy transformation specification and the Request model as parameter. At this point, we obtain an Evaluation model; 4) as mentioned in Section 4, in the presence of conflicts (more than one decision present in the Evaluation model) an extra step is need. It consists in launching the ATL engine with a refining transformation implementing one conflict resolution algorithm with the Evaluation model as input. This process yields an unique access decision; 5) finally, an access decision is returned to the application.

Note however that most of the steps correspond to the internal workflow and that the use and chain of MDE technologies is hidden to the developer (e.g., there exists ant tasks helping to launch and chain transformations). Therefore the task of evaluation access-control request is simplified and reduced to only issue the access-request and wait for the decision.

6. Evaluation

In order to demonstrate the feasibility of our approach, a prototype PDP for a rule and attribute based language have been developed. In this section, we evaluate its performance.

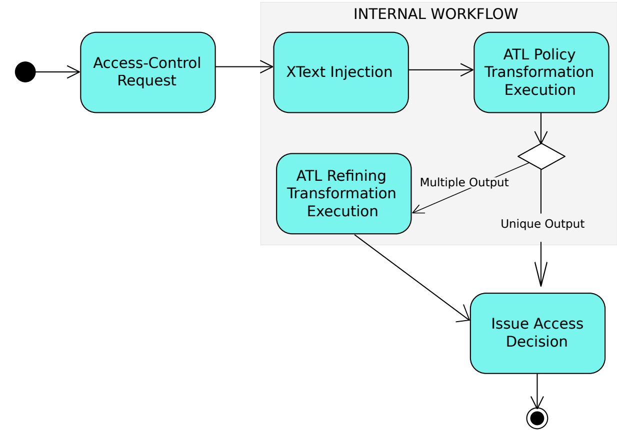

Figure 7. Access Request Evaluation Workflow

We have conducted an experiment consisting in the evaluation of an access-control request against access-control policies (in our case, the transformation specification) containing an increasing number of rules. Concretely, we have launched our transformation in order to obtain an evaluation for a given request against policies containing 10, 50, 100, 250, 750 and 1000 rules[1]. We consider 1000 rules as a realistic case for the applications our approach is meant to be applied. It is also consistent with the size used in other PDP evaluation works where one and four thousand respectively where the limits tested (and that in an enterprise environment[32] [20]. The rules used for our evaluation have been uniformly generated so that the complexity of each rule set is similar and thus, no distortion is added to the experimentation results.

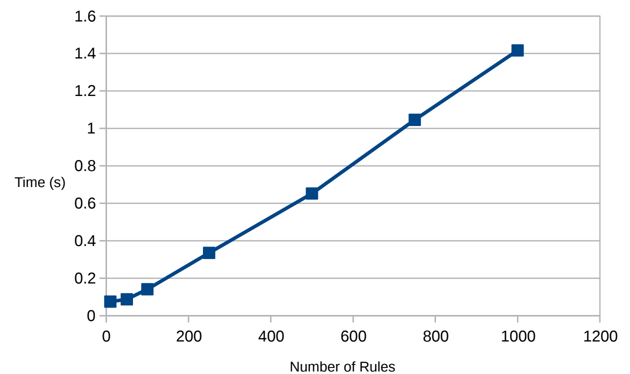

Figure 8. Performance Evaluation Results

The results of this experiments can be seen in Table 6 and Figure 8. We can conclude that the PDP performs reasonably well as the obtained results are on par with those obtained for the XACML PDP implementations evaluated in [32]. Concretely, the obtained execution times are good with values below one second for quite complex policies with up to 700

[1] experimentation done in a Ubuntu 14.04 64 bits environment with the following hardware configuration: Intel(R) Core(TM) i7-4600U CPU @ 2.10GHz, 16 GB RAM DDR3, SSD.

Rules	10	50	100	250	500	750	1000
Time(s)	0.075251	0.08754	0.141308	0.335488	0.652573	1.045864	1.416372

Table 1. Times for the Evaluation of an Access-request Against Policies with Increasing Number of Rules

rules and only slightly over one second for a policy containing up to 1000 rules.

7. Related Work

Providing efficient PDPs as separated modules for the evaluation of access-control policies is an open challenge in the security and software engineering research communities.

XACML[21] and EPAL[3] are access-control and privacy frameworks providing a generic access-control language and propose a PEP-PDP based architecture. However their focus is in the policy language and not in the evaluation engine, that mostly acts as a black-box (extensions and adaptations can only be achieved through the policy language with the corresponding limitations).

Approaches like UMLSec [16] and SecureUML[22] offer the possibility of modeling (role-based) access-control policies but they do not directly support their execution. Close to our work by using MDE techniques for providing PDPs, in [25] the authors provide a PDP/PEP implementation for RBAC based in the PCIM standard whereas in [11] the authors use Security@runtime to implement a PDP (the PDP itself being implemented in Prolog) that supports runtime changes in the access-control policy. Milhau et al. combine secureUML and B-method to obtain verifiable access-control PDP. Contrary to us, those works are tight to specific access-control languages and technologies and their contributed PDPs not specially conceived to be generic and adaptable. In [23] the authors present a generic rule-based access-control metamodel and a framework to 'obtain' domain specific PDPs. However, their approach translates the security policies to the representations used by existing PDPs (e.g., XACML evaluation engines) instead of providing their own execution facilities.

In [12] and [19] authors use OCL to formalize different properties of access-control paradigms such as static and dynamic Separation of Duty (SoD). Although some access-control rules may be expressed in the form of properties and thus evaluated as correctness verification their focus is not in the evaluation of access-requests.

Graph transformations are used to specify and reason about policies in [17]. Contrary to us, they do not hide the complexities of working with transformation to the final developer and propose the direct use of the formalism for the specification of policies. Their focus is also more in the theoretical capabilities of the formalisms than in providing PDPs.

More similar to our approach, in [10], authors propose the implementation of PEP-PDP by using aspect-oriented programming and Drools, a business rule management system for the representation of the policy and PDP reasoning. Our approach is similar to theirs but we provide a generic approach that 1) does not impose any concrete access-control language nor paradigm but that works on any existing one. 2) allows for the uniform (e.g., by using the same set of tool and techniques) manipulation on both the models (policy) and the execution specification itself (i.e., the transformation) to add advanced capabilities like traceability.

We took inspiration from [6], where the authors use ATL for checking OCL constraints in models. We propose a similar approach specially tailored to the case of access-control policy evaluation. Concretely, we add a refining step for the elimination of rule conflicts and the use of HOTs for the generation of the transformation performing the access-request evaluation.

Finally, evaluation of existing PDPs for the XACML language is provided in [32] while in [20] a more specific and optimized PDP is provided by transforming policies to a mathematical representation.

8. Conclusions and Future Work

In this paper we have explored the use of a model transformation framework as an efficient PDP module for access-control languages. We have shown how the ATL language can be used as a generic solution for both the generation of a corresponding transformation specification policy from an existing policy and the evaluation of the policy rules against access-control request. We have also shown how the same transformation framework can be used to implement combination algorithms to deal with rule conflicts.

As a future work, we intend to extend our approach by exploring: 1) the use of modeling and transformation frameworks to manage systems with multiple, possibly-related, access-control policies. We consider that a method for deciding the right transformation to be used for a given access-request in a multi-policy environment will be a natural extension of our approach; 2) the possible benefits that experimental transformation execution modes like lazy evaluation and incremental propagation may provide to the problem of evaluating access-control policies as model transformation executions; 3) the application of model transformation validation and verification techniques to our HOT and generated ATL transformations. We believe that this V&V step may help to discover not only problems with the transformations, but also problems in the original access-control policies.

References

[1] D. 5200.28-STD. *Trusted Computer System Evaluation Criteria*. Dod Computer Security Center, December 1985.

[2] M. A. Al-Kahtani and R. Sandhu. A model for attribute-based user-role assignment. In *Computer Security Applications Conference, 2002.*, pages 353–362. IEEE, 2002.

[3] P. Ashley, S. Hada, G. Karjoth, C. Powers, and M. Schunter. Enterprise privacy authorization language (epal 1.2). *Submission to W3C*, 156, 2003.

[4] M. Benantar. *Access Control Systems: Security, Identity Management and Trust Models*. Springer, 2006.

[5] L. Bettini. *Implementing Domain-Specific Languages with Xtext and Xtend*. Packt Publishing Ltd, 2013.

[6] J. Bézivin and F. Jouault. Using ATL for Checking Models. *Electronic Notes in Theoretical Computer Science*, 152:69–81, 2006.

[7] J. Bézivin, F. Büttner, M. Gogolla, F. Jouault, I. Kurtev, and A. Lindow. Model transformations? transformation models! In *Models'06*, pages 440–453. Springer, 2006.

[8] L. Burgueno, J. Troya, M. Wimmer, and A. Vallecillo. Static fault localization in model transformations. *IEEE Transactions on Software Engineering*, 41(5):490–506, 2015.

[9] F. Büttner, M. Egea, and J. Cabot. On verifying atl transformations using off-the-shelfsmt solvers. In *Models'12*, pages 432–448. Springer, 2012.

[10] Y. Elrakaiby and Y. Le Traon. A PEP-PDP Architecture to Monitor and Enforce Security Policies in Java Applications. In *ARES*, pages 367–374. IEEE, 2013.

[11] Y. Elrakaiby, M. Amrani, and Y. Le Traon. Security@runtime: A flexible MDE approach to enforce fine-grained security policies. In *ESSoS*, pages 19–34. Springer, 2014.

[12] A. B. Fadhel, D. Bianculli, and L. Briand. A comprehensive modeling framework for role-based access control policies. *Journal of Systems and Software*, 107:110–126, 2015.

[13] X. Jin, R. Krishnan, and R. S. Sandhu. A unified attribute-based access control model covering dac, mac and rbac. *DB-Sec*, 12:41–55, 2012.

[14] F. Jouault and M. Tisi. Towards incremental execution of atl transformations. In *ICMT'10*, pages 123–137. Springer, 2010.

[15] F. Jouault, F. Allilaire, J. Bézivin, and I. Kurtev. ATL: a Model Transformation Tool. *Science of Computer Programming*, 72: 31–39, 2008.

[16] J. Jürjens. UMLsec: Extending UML for secure systems development. In *"UML'02"*, pages 412–425. Springer, 2002.

[17] M. Koch and F. Parisi-Presicce. Describing policies with graph constraints and rules. In *ICGT'02*, pages 223–238. Springer, 2002.

[18] D. S. Kolovos, R. F. Paige, and F. A. Polack. The Epsilon Transformation Language. In *ICMT*, pages 46–60. Springer, 2008.

[19] M. Kuhlmann, K. Sohr, and M. Gogolla. Employing UML and OCL for designing and analysing role-based access control. *Mathematical Structures in Computer Science*, 23(04):796–833, 2013.

[20] A. X. Liu, F. Chen, J. Hwang, and T. Xie. Xengine: a Fast and Scalable XACML Policy Evaluation Engine. In *ACM SIGMETRICS Performance Evaluation Review*, volume 36, pages 265–276. ACM, 2008.

[21] H. Lockhart, B. Parducci, and A. Anderson. OASIS XACML TC, 2013.

[22] T. Lodderstedt, D. Basin, and J. Doser. SecureUML: A UML-based modeling language for model-driven security. In *"UML'02"*, pages 426–441. Springer, 2002.

[23] T. Mouelhi, F. Fleurey, B. Baudry, and Y. Le Traon. A model-based framework for security policy specification, deployment and testing. In *MODELS'08*, Toulouse, France, Oct. 2008.

[24] J. Musset, É. Juliot, S. Lacrampe, W. Piers, C. Brun, L. Goubet, Y. Lussaud, and F. Allilaire. Acceleo user guide. *See also http://acceleo. org/doc/obeo/en/acceleo-2.6-user-guide. pdf*, 2, 2006.

[25] R. Nabhen, E. Jamhour, and C. Maziero. A policy-based framework for RBAC. In *Self-Managing Distributed Systems*, pages 181–193. Springer, 2003.

[26] Q. Omg. Meta object facility (mof) 2.0 query/view/transformation specification. *Final Adopted Specification (November 2005)*, 2008.

[27] R. Sandhu, D. Ferraiolo, and R. Kuhn. The NIST Model for Role-Based Access Control: Towards a Unified Standard. In *RBAC'00*, pages 47–63. ACM, 2000. ISBN 1-58113-259-X.

[28] R. S. Sandhu and P. Samarati. Access Control: Principle and Practice. *Communications Magazine, IEEE*, 32(9):40–48, 1994.

[29] D. Steinberg, F. Budinsky, M. Paternostro, and E. Merks. *EMF: Eclipse Modeling Framework (2nd Edition) (The Eclipse Series)*. Addison-Wesley Professional, 2008. ISBN 0321331885.

[30] M. Tisi, S. Martínez, F. Jouault, and J. Cabot. Lazy execution of model-to-model transformations. In *Models'11*, pages 32–46. Springer, 2011.

[31] M. Tisi, S. Martinez, and H. Choura. Parallel execution of atl transformation rules. In *Models'13*, pages 656–672. Springer, 2013.

[32] F. Turkmen and B. Crispo. Performance Evaluation of XACML PDP Implementations. In *ACM workshop on Secure web services*, pages 37–44. ACM, 2008.

[33] X.812. Information technology – open systems interconnection — security frameworks for open systems: Access control framework (ITU-T Rec X.812 / ISO/IEC-10181-3:1996). International Standard ISO-10181-3/X.812, 1996.

[34] E. Yuan and J. Tong. Attributed Based Access Control (ABAC) for Web Services. ICWS '05, pages 561–569, Washington, DC, USA, 2005. IEEE.