PolicyDSL: Towards Generic Access Control Management Based on a Policy Metamodel

Branislav Trninić*, Goran Sladićć, Gordana Milosavljević*, Branko Milosavljević*, Zora Konjović*
* University of Novi Sad/Faculty of Technical Sciences, Novi Sad, Serbia
tminic@gmail.com, sladieq@uns.ac.rs, grsit@uns.ac.rs, mbranko@uns.ac.rs, ftn_zora@uns.ac.rs

Abstract—The paper presents a generic access control management infrastructure suitable for a broad set of systems. The generic infrastructure is based on our policy metamodel (level M2), which is used for the specification of the needed policy model (level M1) such as RBAC, GTRBAC, etc. Having a defined policy model, the abstract and concrete syntaxes of PolicyDSL, our textual DSL for expressing access control policies, are dynamically generated. A security expert is then able to express the actual access control policies (level M0) for the given access control model using the generated DSL. The presented solution can be applied, with no changes, to a number of systems that are based on different access control models or their variants.

I. INTRODUCTION

RBAC (Role Based Access Control) systems are widely used in both academia and the industry. Besides the basic RBAC model, a number of extended RBAC models have been developed that are specialized for a certain domain or purpose (for example, COBAC for context-sensitive control in workflow systems [1, 2]). Although there are successful approaches for formal specification and analysis of access control models, there is a lack of software implementations that facilitate the modeling of concrete policies [3]. The administration of users, roles, permissions, and relations among them may become a huge task for systems with large numbers of entities. In such a large system, the current means for expressing access control rules are cumbersome and inefficient for administrators.

In order to make the task of administering access control systems easier, a number of domain-specific languages (DSLs) have been introduced. Generally, a DSL facilitates explicit representation of domain knowledge. An important goal of a DSL is to keep the language as simple and understandable as possible, while preserving the expressiveness for the given domain [4].

However, the papers that deal with the use of DSLs for modeling access control are mostly focused either on particular access control models, such as SecureUML [9], or on modeling constraints in access control, such as RSL.99 [10] and RCL2000 [11] for separation of duties (SoD). These languages do not support different access control models, or model families (such as RBAC with its variants), or support only one of the aspects of the access control model (such as SoD). Lately, requirements for access control systems are growing more complex and render traditional mechanisms, based on a single access control model, inefficient and impractical. Hence, these requirements have to be met by combinations and specializations of standard models – e.g. hybrid models [12].

Another problem is that most currently known DSLs for modeling access control (AC) are based on the UML language. The paper [5] identifies the most common ways of using UML for specifying security. The UML 2.0 standard contains a large set of modeling concepts that are related in complex ways. The profile mechanism does not provide a means for precisely defining semantics associated with extensions [6, 7]. It is almost impossible to use only a part of UML without including the rest of the whole standard. Although conceived as a standard for general modeling of various application domains, UML has failed to provide expected results in the field of model-driven development [8].

In order to overcome these difficulties, this paper presents a novel solution – a three-level access control management infrastructure suitable for a broad set of systems. The infrastructure is based on our policy metamodel (level M2), which is used for the specification of the needed policy model (level M1) such as RBAC, GTRBAC, etc. Having a defined policy model, the abstract and concrete syntaxes of PolicyDSL, our textual DSL for expressing access control policies, are dynamically generated. A security expert is then able to express the actual access control policies (level M0) for the given access control model using the generated DSL. The syntax and semantics of the PolicyDSL presented in this paper facilitates the authoring of access control rules by security experts.

The goal of the approach presented here is to provide a universal way of specifying access control rules for families of different access control models, while keeping the used DSL from degenerating into a general-purpose modeling language. This DSL should directly support the concepts of the given access control model with its syntax in order to minimize the cognitive distance [13].

We have decided to use a DSL with a textual concrete syntax that is not based on UML. The process of modeling does not require the knowledge of UML syntax and semantics. The use of textual concrete syntax facilitates development automation and integration with other development tools. While the use of graphical concrete syntax is good for visualizing the big picture, initial system administration, and understandable to non-experts, it soon becomes cluttered with a large number of graphical elements and has proven to be insufficient [14].

The rest of the paper is structured as follows. Section 2 reviews the related work. Section 3 introduces our three-level policy model architecture. Section 4 presents the access control policy metamodel at level M2. Section 5
presents an example of an access control model at level M1. Section 6 describes PolicyDSL, a language for expressing access control policies at level M0. Section 7 concludes the paper.

II. RELATED WORK

Currently published papers focus mainly on the use of the standard Model-Driven Architecture (MDA) methodology supported by UML-based languages for modeling the access control infrastructure, or the creation of a particular DSL that is aimed to support a particular access control model.

The paper [15] suggests the use of an AC metamodel for generating test cases. AC policies are defined according to the presented metamodel, and then transformed to XACML. The integration of the application and the XACML platform is carried out using AOP (aspect-oriented programming). The metamodel is used for verification of the defined policies and generating fault-injection test cases. Contrary to our approach, the metamodel supports the concepts of rule-based AC systems and lacks direct support for the RBAC family of AC models and DSLs.

The paper [9] introduces SecureUML, a language for modeling computer security systems. SecureUML defines a vocabulary for adding access control information to UML-based models. The added information is combined with the UML model of business processes and the infrastructure for AC enforcement is generated. MOF is used for specifying a new UML-based language for specifying RBAC permissions and constraints. SecureUML does not include the resources being protected by the AC system, so the same authors propose ComponentUML [16], a language for modeling component-based systems. ComponentUML’s entities are specializations of Action and Resource entities from the SecureUML metamodel. Application resources being protected by the AC system are modeled by their instantiation.

The paper [17] is among the first to deal with modeling security aspects of business systems. It presents UMLSec, an extension of UML that facilitates embedding the AC-related information in diagrams used for system specification. UMLSec is implemented as a UML profile using standard UML extension mechanisms. UMLSec does not directly support RBAC, DAC, or MAC entities, but only general security aspects.

Dae-Kyo et al. in [18] propose the use of RBAC UML templates. RBAC templates are extended as needed with application-dependent concepts and are integrated with the main model of the business application. The methodology includes specifying AC permissions in the form of OCL expressions. This facilitates the identification of static AC permission conflicts.

Sanchez et al. in [19] present ModelSec, an approach that defines a generative architecture based on chained transformations of security models given in different abstraction levels. That paper also includes SecML, a DSL for specifying security requirements. SecML is not based on UML, but on the authors’ security requirements metamodel, and includes a graphical concrete syntax. ModelSec represents a general MDS approach and is not focused on a specific field (such as access control). In contrast to our approach, this method is tightly coupled with the application development process, does not support full SoD and changing AC policies at runtime.

In [20], Ray et al. use parameterized UML diagrams for modeling RBAC and MAC frameworks. These frameworks are then combined to create a hybrid AC model. Although this model is more general than MAC or RBAC, it is not extensible. An AC metamodel and a DSL for specifying AC policies (apart from UML) are not defined.

Slimani et al. [12] define an AC metamodel as a basis for the development of a UML-based modeling language. Their metamodel comprises four classes: Subject, Category, Resource, and Action. In order to model MAC or RBAC, the metamodel needs to be specialized by introducing new classes that inherit the initial four, and thus specializing the semantics. This approach, among all reviewed, is the most similar to ours. However, their metamodel is more of a hybrid AC model rather than a metamodel. It is not a metamodel in the sense that it is used to model AC models, but is a generic template that can yield RBAC, MAC, or some other model through specialization.

The papers [10, 11] present formal languages for specifying SoD constraints in RBAC systems, RSL99 and RCL2000. The languages are formulated as predicate logic expressions. The paper [21] uses OCL as a basis for defining SoD and other types of constraints. These languages cannot be used to model all aspects of access control. Contrary to those approaches, our PolicyDSL models entity instances, attribute values, and relations. In our case, the constraints are given implicitly (for example, by roles being elements of an SoD set) or through attribute values that can be textual representations of logical expressions, given in OCL.

III. POLICY ARCHITECTURE OVERVIEW

Our approach is based on models at 3 different abstraction levels defined by MOF classification, as presented in Figure 1. Level M2 features an AC policy metamodel. Level M1 features an AC policy model (for example, RBAC) that conforms to the metamodel at Level M2. Classes such as Role, User, and Permission are AC entities. Classes such as RoleHierarchy and UserRole are AC relations. Level M0 features instances that conform to the policy model at Level M1. For specifying instances we use our PolicyDSL with a textual concrete syntax. The syntax includes concepts defined by AC model being used, with as little syntactic noise [22] as possible. This DSL is used to define the following:

- entity instances (for example, roles Clerk and Manager)
- relations among entity instances (for example, Manager inherits Clerk)
- attribute values of entity instances
- attribute values of relations among entity instances

Figure 2 presents the prototype implementation of the architecture based on the previously described 3-level hierarchy. The development process starts at Level M1 with the definition of a policy model (that conforms to the policy metamodel at Level M2). The policy model is usually defined once for the particular system, and is further used to configure the syntax of policy DSL and for generating the access control interfaces for the target implementation platform. A PolicyDSL workbench is a
development environment that uses the policy model to configure the abstract DSL syntax by parameterizing language constructs. Parameterized constructs support creation of instances of entities and relations defined at level M1. The environment also facilitates development by syntax coloring and context-dependent hinting. For example, a list of available roles can be displayed while entering a relation that involves roles.

The program interfaces being generated from the policy model are implemented using the target platform. Modeling AC policies in a real-world system is a continuous process. Should a policy model change, the program interfaces for AC enforcement need to be re-generated, possibly re-implemented and re-compiled for the target platform. If the change in the policy model only introduces new AC entities and relations, old DSL scripts that contain actual policies are still executable. In case that some AC entities and relations are removed from the policy model, old DSL scripts need to be adjusted to the concepts of the new model. For example, if SoD is removed, all declarations of SoD relations among user roles need to be removed from the scripts.

User authentication is delegated to a separate authentication service and is considered to be outside of the scope of this AC infrastructure that deals only with the process of authorization.

I. POLICY METAMODEL

Our access control policy metamodel is presented in Figure 3. AC Concept is the parent metaclass for AC Entity and AC Relation metaclasses.

AC Entity metaclass instances are, for example, Role, User, Permission – entities of the particular access control model. An entity can be instantiated in a DSL script or externally (for example, from the list of registered users). The attribute fromDSL indicates whether the particular class instances are defined in a DSL or externally.

AC Relation metaclass instances represent different types of relations that are present in the particular access control system, such as:

- the belonging of a user to a role
- role inheritance
- static SoD
- dynamic SoD
- sets of conflicting roles
- sets of conflicting users

The association between the type of relation and the entity is represented by the Relation Member metaclass. An instance of the Relation Member describes an association of a class representing a relation with a class representing an entity being a part of the relation. For example, the relation “role inheritance” includes two Role entities, so it is represented with a single instance of Relation Member with the attribute Cardinality = 2. The relation “user-role assignment” has associated two Relation Member instances with the attribute Cardinality = 1. From a DSL syntax standpoint, if the policy model specifies a relation “SSoD” (static SoD), with the cardinality 2 x Role, the resulting DSL syntax will contain a construct that begins with an SSoD token and then expects two arguments of type Role.

Attribute metaclass instances represent an attribute of an entity or a relation being modeled. Particular attribute values are assigned by the security expert. These attribute values can be used during authorization process. For example, attributes can hold context conditions in context-sensitive AC models such as COBAC.
II. POLICY MODEL

For the given access control policy metamodel PolicyMM, and the given access control policy model PolicyModel, the following predicates are defined:

- **InstanceOf**((x, y)) – denotes that x ∈ PolicyModel is an instance of the metaclass y ∈ PolicyMM.
- **References**((x, y)) – denotes that instance x ∈ PolicyModel references another instance y ∈ PolicyModel. This predicate is commutative.
- **Property**((x, y)) = z – attribute y of instance x ∈ PolicyModel has the value z.

An instance of a metaclass from PolicyMM is a class in the PolicyModel. For example, an instance of AC Entity metaclass is the class Role (as used in RBAC). A model class instance is an object from the real system: an instance of Role is Manager that represents a user role in a particular application.

Figure 4 presents a class diagram of the standard RBAC model with role hierarchy and SoD extensions as Level M1 model within our architecture. Separation of Duty contains the attribute Dynamic which, with the value true, denotes that separation of duties is dynamic rather than static.

The inheritance relation is modeled as an ordered pair (a, b) a,b ∈ R, where R is a set of roles. The SoD relation is modeled as an ordered n-tuple (a, b, c, ..., n) a,b,c,...,n ∈ R, i.e., in a single relation instance there are two or more mutually conflicting roles.

The presented RBAC model and its connection to the metamodel PolicyMM can be defined with the following formal expressions:

```java
// Classes in RBAC PolicyModel are instances of PolicyMM
// metaclasses
InstanceOf(User, AcEntity)
InstanceOf(Role, AcEntity)
InstanceOf(Permission, AcEntity)
InstanceOf(UserName, Attribute)
InstanceOf(RoleName, Attribute)
InstanceOf(PermissionName, Attribute)
InstanceOf(DynamicSoD, Attribute)
// Relations of associating a class and its attributes
References(User, UserName)
References(Role, RoleName)
References(Permission, PermissionName)
// Attribute values in RBAC PolicyModel
Property(UserName, Name) = "Name"
Property(UserName, Type) = "String"
Property(RoleName, Name) = "Name"
```

The predicate Property(User, FromDSL) = false in the previous listing states that the set of instances is not defined in a DSL script, but is loaded from an external source (for example, a user database).

Figure 5 presents a graphical representation of previously defined formal expressions.

The policy model is used to generate interfaces for the target platform that will enable the integration of DSL workbench and the AC infrastructure. For the previous RBAC example, generated Java interfaces are as follows:

```java
public interface IAcUser extends IAcEntity {
    public String getName();
    public static List<IAcUser> list();
}

public interface IAcRole extends IAcEntity {
    public String getName();
    public void setName(String name);
    public static IAcRole create(String name);
    public void delete();
}

public interface IAcPermission extends IAcEntity {
    public String getName();
    public static List<IAcPermission> list();
}
```
public interface IAcRoleUser extends IAcRelType{
    public IAcRole getRole();
    public IAcUser getUser();
    public static IAcRoleUser create(IAcRole role, IAcUser user);
    public void delete();
}

public interface IAcRolePerm extends IAcRelType{
    public IAcRole getRole();
    public IAcPermission getPermission();
    public static IAcRolePerm create(IAcRole role, IAcPermission perm);
    public void delete();
}

public interface IAcRoleHierarchy extends IAcRelType {
    public IAcRole getRoleE1();
    public IAcRole getRoleE2();
    public static IAcRoleHierarchy create(IAcRole role, IAcRole role);
    public void delete();
}

public interface IAcSeparationOfDuty extends IAcRelType {
    public boolean getDynamic();
    public void setDynamic(Boolean dynamic);
    public List<IAcRole> getRoleMembers();
    public static IAcSeparationOfDuty create(List<IAcRole> roles);
    public void delete();
}

Interface IAcUser contains only the getName method, since User is denoted as external, so all its attributes are read-only. Obtaining the list of all users is performed by invoking the list static method.

Interface IAcRole contains getter/setter methods for the attribute name and methods for creation and removal of Role instances.

Interface IAcPermission, similarly to IAcUser, provides methods for reading the permission name and retrieving currently defined permissions.

III. POLICY DSL

PolicyDSL is a DSL with the syntax that is dynamically adapted to the features of the system being modeled. PolicyDSL’s abstract syntax is parameterized by the data from the policy model. This way we have achieved dynamic specialization of the syntax for the particular AC model. Hence, it is not necessary to design another DSL and its environment for every variation of the AC model. One flexible DSL can handle a family of models (for example, based on RBAC).

A DSL, by definition, should be as specific as possible and directly support concepts from the modeling domain with its language constructs. From the software engineering standpoint, it is desirable to achieve the reuse of existing software artifacts. Our approach enables the reuse of DSL syntax, since it automates the process of its adjustment, while keeping the language domain-specific.

The grammar of PolicyDSL in PEG notation is given in the following listing.

```
Policy ← (AcEntity / AcRelation)*
AcEntity ← EntType EntName Attribs? '{' RelDef*'}'
EntName ← [A-Za-z] ([A-Za-z0-9]+)*
EntType ← [A-Za-z] ([A-Za-z0-9]+)*
RelDef ← RelType Attribs? '{' MulMember* '}'
RelType ← [A-Za-z] ([A-Za-z0-9]+)*
Attribs ← '{' Value (',' Value)* '}'
Value ← IntValue / StringValue / BoolValue
MulMember ← EntName (',' EntName)*
IntValue ← [0-9]+ StringValue ← '"' (!"" AnyChar)* '"'
BoolValue ← false / true
EntName ← '*' EntName* BoolValue ← '*' BoolValue

AcRelation ← RelType Attribs? '{' MulMember* '}'
```

EntType and RelType symbols are taken from the policy model, as well as cardinality and type of attributes associated to an entity or relation, and cardinality and type of entities in a relation. Consider the following PolicyDSL expression:

```
Figure 5. RBAC as an instance of the policy metamodel
```
SeparationOfDuty(true) {
  Manager*CEO*Clerk, Accountant*Clerk
}

This expression defines an SoD relation. There are two instances of an SoD relation, i.e., two sets of conflicting roles. The first set contains three roles, while the second contains two. The value true is assigned to the Dynamic attribute of the SoD relation, implying that these relations represent dynamic SoD. Next, let’s consider the following expression:

RoleHierarchy {
  Manager*Accountant, Manager*Chairman, Manager*RemoteUser
}

This expression defines three inheritance relations in which role Manager inherits the other three roles. An inheritance expression can be placed inside a definition. In this case, the first role in a relation is omitted, as in the following example:

Role: Manager {
  RoleHierarchy {Accountant, Chairman, and RemoteUser}
  RolePermission {P1, P2, P3}
};

This expression defines an AC entity of type Role named Manager. Manager is involved in two types of relations. The first is role hierarchy, where Manager is inheriting from Accountant, Chairman, and RemoteUser. The second is the assignment of three permissions P1, P2, P3 to Manager.

IV. CONCLUSION

This paper has presented a novel way of modeling and metamodelling access control policies. This process spans three meta-levels. At level M2, we have defined a policy metamodel. Using this metamodel, different policy models can be defined at level M1 (such as RBAC, GTRBAC, etc.). The definition of the policy model is used to parameterize the syntax of PolicyDSL used at Level M0 for specifying actual access control policies in a particular system.

This approach has yielded several important features.

- PolicyDSL is exclusively focused on expressing AC policies, with as little syntactic noise as possible. This can aid security experts to author access control rules more efficiently.
- PolicyDSL syntax is dynamically specialized for the given AC model at Level M1. Hence, it is not necessary to design another DSL for every variation of the AC model. One flexible DSL can handle a family of models.
- Facilitated reuse of the same DSL development tools for families of different AC models.
- Integration of DSL tools with the actual AC system and external sources is facilitated by generated programmatic interfaces for the target implementation platform.

The proposed solution does not thoroughly consider dynamic constraints, like context-sensitive constraints, when defining and enforcing access control. A further direction in development of our policy model includes the support for those constraints. This will provide support for defining and enforcing context-sensitive access control.

Our experience in using the prototype implementation showed that some segments (like role hierarchy) would be efficiently defined by using graphical DSL, rather than textual ones. Therefore research on a graphical DSL is in progress.

REFERENCES


