From Access Control Policies to an Aspect-based Infrastructure: A Metamodel-based Approach*

Christiano Braga**

Universidad Complutense de Madrid

Abstract. Security is among the most successful applications of aspect-oriented concepts. In particular, in role-based access control, aspects capture access conditions in a quite modular way. The question we address in this paper is *how can aspects be generated from access control policies under a validated process?*
We present a metamodel-based transformation from SecureUML, a role-based access control language, to an abstract aspect language. Within this model-driven engineering context, a security policy is represented as an instance of SecureUML's metamodel and the generated aspect is represented as an instance of the abstract aspect language metamodel. Invariants specified on the *merged* metamodel of SecureUML and the abstract aspect language are checked to validate the generated aspect with respect to the given security policy.
We have prototyped our approach as a Java application on top of IT-P/OCL, a rewriting-based OCL evaluator. It outputs validated AspectJ code from a SecureUML policy.

1 Introduction

The use of aspects [16] in security is among the most successful uses of aspect-oriented concepts, both at the specification and coding levels (e.g. [21, 6, 13, 11, 20, 15]).

In particular, aspects capture, in a modular way, the control conditions of role-based access control (RBAC) [12] policies. An RBAC policy describes the constraints that a given user, within a certain *role*, must fulfill in order to perform an action, that is, access, a controlled system. Essentially, access control constraints can be understood as *preconditions* to calls for controlled resources.

Preconditions can be directly represented as the so called *before advices* in aspects. Before advices are program statements that are executed before an (user-defined) identifiable execution point, or join point in aspect-oriented terminology, is reached. An example of such a join point is a call to a particular

* Research sponsored by Ramón y Cajal program (MICINN), project DESAFIOS (TIN2006-15660-C02-01, MICINN) and project PROMESAS (S-0505/TIC/0407, CAM).

** On leave from Universidade Federal Fluminense, Brasil.

method. A join point can be intercepted using pointcut declaration in an aspect-based language, which essentially defines a pattern that matches whenever the desired join point is executed, such as a method call.

The benefits of generating aspect-code from RBAC policies is twofold: (i) we *modularly* represent access control constraints as pre-conditions captured as before advices in an aspect and (ii) the generated code is *automatically* called from the client code (whose access is being controlled) by the so called *weaving* process. The first benefit is actually shared with other approaches such as object-oriented design by contract. However, weaving is only supported by aspect-based languages. (See Section 2 for an example.)

The literature in the connection of aspects and RBAC is rich. In [21] the authors use aspect-oriented modeling to specify access control concerns. In [6] the focus is on web applications. The authors propose an aspect-oriented approach to declarative access control for web applications. In [13] they use aspects to implement the RBAC reference model [22]. In [11] the authors present quite clearly how aspects can be used to implement access control policies but identify deployment problems. They foresee that automating the generation process from higher-level descriptions is a must. The proposals in [20, 15] research in this direction. In [20] the authors present how role-slicing models are translated into aspect code. The proposal in [15] uses aspect-oriented modeling to represent security concerns and generates aspect code.

The question we address in this paper is *how can one automatically generate aspects for access control policies under a rigorous development method, that is, within a validated generation process?*

To answer this question we propose a model driven architecture approach (MDA) [18]. Moreover, we follow the model-driven security (MDS) ideas [1]. In MDS, we quote, "designers specify system models along with their security requirements and use tools to automatically generate system architectures from the models, including complete, configured access control infrastructures." It is argued that this approach "bridges the gap between security analysis and the integration of access control mechanism into end systems. Moreover, it integrates security models with system design models and thus yields a new kind of model, *security design models*."

We have defined a transformation from role-based access control policies, modeled in the RBAC-based language SecureUML, defined in [1], to aspects, modeled in a simple abstract aspect-oriented language, a subset of constructs commonly found in aspect-oriented languages, which essentially defines pointcuts and before advices. The aspect concepts we believe are the necessary ones to represent access control. Our approach is a metamodel-based one, that is, we consider the metamodels of the languages involved in the transformation. Therefore, a security policy is understood as an object model of the SecureUML metamodel and the generated aspect is an object model of the aspects metamodel.

The validation of the transformation process smothly integrates with the validation of SecureUML policies proposed in [7]. There, SecureUML policies are

validated by evaluating the OCL *invariants* of the SecureUML metamodel over policies captured as object models. We extend their approach, in the context of MDA, by proposing that model transformations from SecureUML policies to code could be validated on the *merged* metamodel of SecureUML and target language. Therefore, we consider the merged metamodel of SecureUML and aspects to guarantee conformance of a generated aspect with respect to the given security policy. We name *transformation invariants* the invariants associated with the merged metamodel.

Our MDA approach thus applies the metamodel transformation and model merging approaches from [18]. Moreover, it neither requires a new specification language, such as QVT [19], for its specification, nor commits to a particular implementation. A fact that we believe to be positive in the sense of using the same languages from the modeling phase (UML and OCL) and not imposing a particular implementation framework. A point of view that we believe is shared with [2]. At the implementation level, any programming language can be used as long as the implementation preserves the transformation invariants.

We have prototyped our approach on top of the rewriting-based OCL evaluator ITP/OCL [10, 4]. The prototype is implemented in Java and essentially loads ITP/OCL with all the above mentioned metamodels and invariants. When given an object model of a security policy as input the transformer generates an object model of an aspect. The invariants of the merged metamodel are then evaluated on the union of the object models of security policy and aspect, following the Design by Contract [17] idea of run-time monitoring of assertions. If they hold, the transformer outputs an aspect using AspectJ as concrete syntax.

To summarize our contribution in one paragraph: we propose an *automatic* and *validated* and code generation process from role-based access control policies into aspect code. Each policy gives rise to an aspect. The validation of the aspect generation is model-based. The generated aspect is validated by evaluating the OCL invariants, defined for the model of SecureUML and aspects, on model instances that represent the security policy and aspect. We have prototyped our approach on top of an OCL evaluator.

This paper is organized as follows. In Section 2 we illustrate how a SecureUML policy may be represented by an aspect. Section 3 presents our metamodel-based transformation approach to aspect generation from SecureUML access control policies. Section 4 discusses a prototype implementation to our approach. Section 5 algebraically formalizes the notions of merged metamodel, transformation invariants and validation of the transformation invariants algebraically. Finally, Section 6 concludes this paper with final remarks.

2 From SecureUML Policies to Aspect Code

An access control policy in SecureUML specifies which (user) *roles* are given *permissions* to perform *actions* over *resources* under certain *authorization constraints*. SecureUML has a loose semantics, in an algebraic sense [14], with respect to the resources which access may be controlled. Our resource language of

choice, called ComponentUML, defines resources to be entities which may have attributes, methods, and association ends. Attributes, methods, and association ends are also resources which access may be controlled.

As an example, consider the SecureUML policy specified by the model in Figure 1. The model specifies the access control over the entity *TRC* (for test report configuration). Users under three different roles may access a *TRC*: *Test_Operator*, *Test_Supervisor* or *Test_Administrator*. These roles are related under role inheritance. The most "powerful" role is a *Test_Administrator*, who inherits the access controls permissions from the *Test_Supervisor* role. The latter inherits from the *Test_Operator* role. Therefore, the *Test_Operator* role is the less powerful one.

For a user to execute the (atomic) *create* action over *TRC*, under a *Test_Operator* role, then the authorization constraint attached to the permission *NewPrivate* must hold. This permission means that the parameter *p_owner* of the *create* action must be equal to the name of the user (denoted by SecureUML's special variable *caller*) and also that the parameter *p_scope* of the *create* action must be equal to the constant *Private*. To perform the same action under the role of *Test_Supervisor* the permissions *NewGlobal* or *NewPrivate* (since *Test_Supervisor* inherits permissions from *Test_Operator*) must hold.

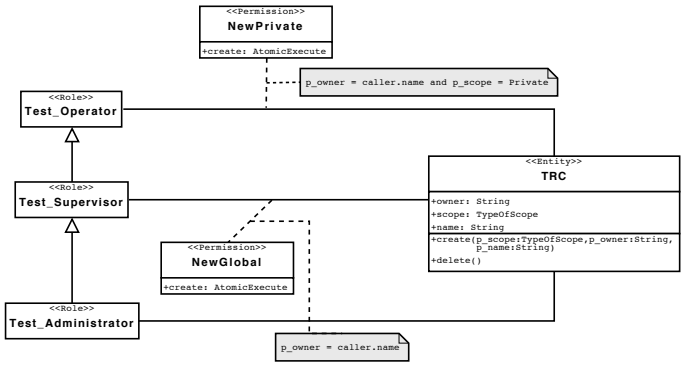

Fig. 1. An Example Access Control Policy

The model in Figure 1 is a quite simplified version of the models produced in an industrial project with a major Spanish technology company, which results are reported in [8].

From this model, essentially, we produce an abstract class, that represents an interface to a concrete implementation of the *TRC* entity, and an aspect, that implements the access control to *TRC*'s methods. For each *TRC* method, only *create* in this example, we declare, in the *TRCAccessControl* aspect, a pointcut, identified by *createPC*, to capture a call to *create*, and a before advice. The advice is triggered whenever *create* is called and checks if the authorization

constraints for *NewGlobal* or *NewPrivate* hold. It raises an exception otherwise. The abstract class and aspect for the *TRC* access control policy in Figure 1 is presented below.

```
abstract class TRC {
   int scope ;    String owner ;    String name ;
   abstract TRC create( int p_scope, String p_owner, String p_name )
                                                          throws Exception ;
}

aspect TRCAccessControl {
   static Role caller ;
   TRCAccessControl() { caller  = Env.getUserRole() ; }
   pointcut createPC(int s, String o, String n) :
     call(TRC TRC.create(int, String, String)) && args(s,o,n) ;
   before(int s, String o, String n) throws Exception : createPC(s,o,n)
   {
       if (caller  instanceof TestSupervisor)
          if (o == caller.name) return ;
          else throw new
              Exception("Current user may not create a TRC.") ;
       if (caller  instanceof TestOperator)
          if ((s == TypeOfScope.PRIVATE) && (o == caller.name)) return ;
          else throw new
              Exception("Current user may not create a TRC.") ;
       throw new Exception("Current user has role "+caller+
                                        " and may not create a TRC.") ;
   }
}
```

In our example, whenever the method `create`, from the class *TRC*, is called (represented by pointcut `createPC`), the associated before advice will be executed before the body of the method `create`. Therefore, the aspect modularly captures the access control requirement and the aspect weaving process transparently integrates it with the client code, which, in our example, calls method `create` from class `TRC`.

We have implemented an automatic and validated process for generating aspect code from SecureUML policies. Our translation follows the MDS ideas mentioned in the introductory section. The transformation process is the subject of Section 3.

3 Transforming SecureUML Policies into Aspects

Our approach consists on a validated and automatic transformation from SecureUML policies to aspect code. The target language of our transformation is an abstract aspect-oriented language, that we call Aspects for Access Control (AAC) which has abstract classes, protected attributes, methods, aspects, pointcuts and before advices as language constructs; the elements that appear to be necessary from the aspect-oriented paradigm to code access control.

The proposed transformation is thus a transformation between domain-specific languages: the source captures access control policies and the target an aspect language for access control. The transformation function essentially relates each component that requires access control security (called an entity in SecureUML terminology) into an abstract class and an aspect. The abstract class represents an interface that a concrete implementation component of the controlled component must implement. The aspect implements the access control constraints that must hold when a component's method (overloaded from the associated abstract class method) is called.

We use metamodels to specify and validate the transformation process, following the MDA approach and aiming at a smooth integration of the transformation process into the modeling phases. The syntax of each language (i.e. SecureUML and AAC) is specified as a metamodel in UML, together with its OCL invariants that capture the structural constraints of the given metamodel. We extend the model-based validation process for SecureUML proposed in [7] by defining the metamodel of SecureUML merged with AAC that disjointedly unites the two languages, adds new relationships among their classes and new invariants over the merged language. It specifies when an AAC model is a properly generated one from a given security policy.

In our proposal, validation means to check the invariants for a given access control policy, for the generated abstract class and aspect and all of them together, that is, for the generated abstract class and aspect with respect to the given access control policy. The validation process should occur in two different moments in time:

1. Before the transformation is applied: the invariants of the SecureUML metamodel are applied to the given access control policy. This step guarantees that the given access control policy is well-formed.
2. After the transformation is applied:
   (a) The invariants of the AAC metamodel are applied to the generated abstract class and aspect. This step guarantees that they form a valid AAC model *per se*.
   (b) The invariants of the *merged* metamodel of SecureUML and AAC are applied to both the security policy and the generated abstract class and aspect.

The validation process is automatic and is implemented on top of ITP/OCL tool, an OCL evaluator. This is the subject of Section 4.

This section continues as follows: in Section 3.1 we discuss the metamodels for each language and in Section 3.2 we outline the transformation function from SecureUML to AAC.

3.1 A Metamodel-based Approach

A metamodel defines the elements of a language, relationships between elements of a language and assertions that constraint the relationships between the elements of a language.

We chose UML's class diagrams to specify a language. Therefore, a metamodel essentially consists of classes, attributes, methods, associations between classes (with roles and multiplicities) and generalizations between classes. OCL is our specification language of choice to specify invariants on metamodels.

SecureUML Metamodel SecureUML[1] provides a language for modeling *Roles*, *Permissions*, *Actions*, *Resources*, and *Authorization Constraints*, along with their *Assignments*, i.e., which permissions are assigned to which roles, which actions are assigned to which permissions, which resources are assigned to which actions, and which constraints are assigned to which permissions. In addition, actions can be either *Atomic* or *Composite*. The atomic actions are intended to map directly onto actual operations of the modeled system. The composite actions are used to hierarchically group more lower-level ones and are used to specify permissions for sets of actions.

SecureUML leaves open what the protected resources are and which actions they offer to clients. These are specified in a so-called *dialect* and depend on the primitives for constructing models in the system design modeling language of choice. ComponentUML is our dialect of choice. It is a simple language for modeling component-based systems. Essentially, it provides a subset of UML class models: *Entities* can be related by *Associations* and may have *Attributes* and *Methods*. Therefore, by using SecureUML+ComponentUML, it is possible to model the permissions that an user playing a given role has over an entity, an attribute, a method or an associations, i.e., the actions such an user can execute while trying to access the resource. The metamodel of SecureUML+ComponentUML is given in Figure 2. Note that the security policy drawn in Figure 1 shows an instance of SecureUML+ComponentUML metamodel.

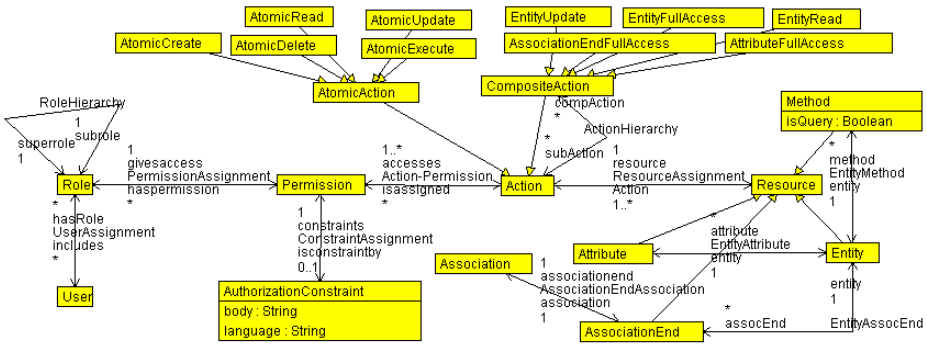

Fig. 2. The SecureUML+ComponentUML metamodel

[1] The material in this subsection is adapted from [8].

AAC Metamodel The elements of the AAC metamodel are: *ResClass* (where the prefix *Res* stands for resource), *ResAttribute*, *ResMethod*, *ResGetMethod*, *ResSetMethod*, *Aspect*, *Pointcut*, *BeforeAdvice*, *RoleClass* and *Env*. The metaclasses *ResClass*, *ResAttribute* and *ResMethod* represent elements of the generated abstract class after the application of the transformation function to a SecureUML policy. A *ResMethod* has two subclasses: *ResGetMethod* and *ResSetMethod*. The metaclasses *Aspect*, *Pointcut* and *BeforeAdvice* represent elements of the generated aspect. The metaclass *RoleClass* represents *Roles* as classes. The metaclass *Env* represents the environment that must provide user run-time information such as the user's current role and the user's name.

An instance of *ResClass* may have many *ResAttributes* and *ResMethods*. An instance of an *Aspect* may have many *Pointcuts* and each *Pointcut* has one and only one *BeforeAdvice*. Moreover, an instance of an *Aspect* must be related with one and only one *ResClass* and each *Pointcut* must be associated with a single *ResMethod*. The metaclass *RoleClass* is related to itself.

An example of OCL invariant for this metamodel is that each *ResMethod* in a *ResClass* must have one *Pointcut* in the *Aspect* associated with the *ResClass*. This invariant could be called consistency between *ResMethod* and *Pointcut*. The OCL invariant below declares that the navigation from a *ResMethod* through its link to its *Pointcut* and then its *Aspect* should point to the same object as navigating through the *ResMethod*'s *ResClass* and then its *Aspect*.

---

context ResMethod **inv**:
self.allInstances()−>forall(rm |
rm.Pointcut-ResMethod.Aspect-Pointcut = rm.Class-ClassMethod.Aspect-Class)

The Merged Metamodel The classes in the merged metamodel of SecureUML and AAC are given by the disjoint union of the classes in the metamodel of SecureUML and the classes in the metamodel of AAC. The relations in the merged metamodel of SecureUML and AAC are given by the disjoint union of the relations on each metamodel including new relations that associate the classes on each metamodel. Moreover, it specifies which properties must hold so that an instance of the merged metamodel of SecureUML and AAC is well-formed. That is, if an AAC model is a valid abstract class and aspect for the given SecureUML policy, with respect to the invariants defined in the merged metamodel.

In the merged metamodel, an *Aspect* is associated with one and only one *Entity* and such metaclass is related to a single *ResClass* to represent the class of an entity. A *ResMethod* is associated with one and only one *Method*, and a *ResAttribute* is associated with a single *Attribute*. The subclasses of *ResMethod*, *ResGetMethod* and *ResSetMethod*, are related to *Attribute* in order to indicate the getters and setters of the attributes, if any. In addition, an *AuthorizationConstraint* is associated with one and only one *BeforeAdvice* that will implement the constraint but such an advice may include the implementation of several constraints.

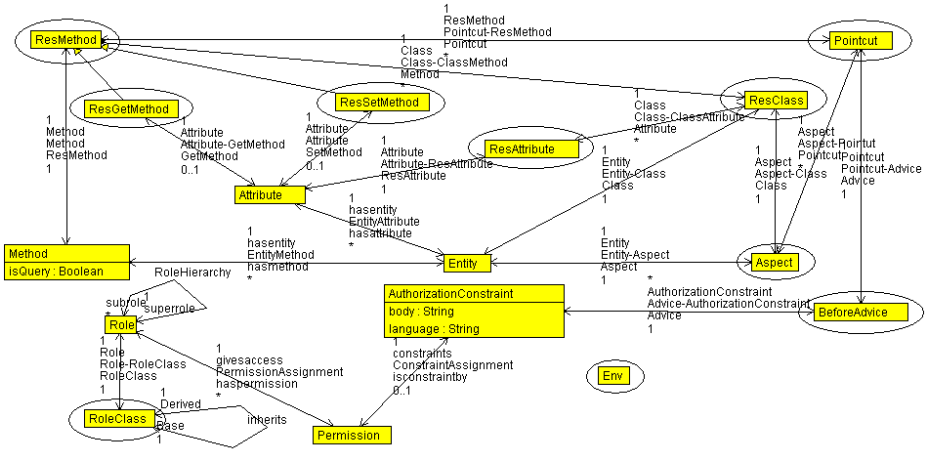

Fig. 3. A subset of the merged metamodel

An example of an OCL invariant over the merged metamodel is shown below.[2] The invariant specifies that for each *Method*, there exists a *ResMethod* in the *ResClass* associated with the *Method*'s *Entity*, such that, for each *ResMethod*, the *BeforeAdvice* in the *ResMethod*'s *Pointcut* is associated with the *ResMethod*'s *Authorization Constraints*.

The invariant uses two auxiliary operations:
(i) *ResMethod::validateResMethodConstraints(Method m):Boolean* and
(ii) *Method::getMethodResMethods:Set(ResMethod)*. The second one, which declaration is not shown, returns the *ResMethods* associated with a given *Method*. The first operation, shown below, checks if the given *Method* is associated with the current *ResMethod* (denoted by the special OCL variable *self*) and if the *AuthorizationConstraints* in the *BeforeAdvice* associated with the *Method*'s *Pointcut* are the same as the given *Method*'s *AuthorizationConstraints*, returned by the function *Method::allMethodConstraints()*.

---

context Method **inv**:
self.allInstances()−>forAll(m | (m.getMethodResMethods())−>
exists(rm | rm.validateResMethodConstraints(m)))

context ResMethod::validateResMethodConstraints(m:Method):Boolean **body**:
(self.Method = m) and
(self.Pointcut-ResMethod.Pointcut-Advice.Advice-AuthorizationConstraints =
m.allMethodConstraints())

---

[2] The complete set of invariants for the merged metamodel can be found in `http://maude.sip.ucm.es/~cbraga/transformationInvariants.pdf`.

3.2 The Transformation Function

For a given SecureUML policy the transformation function produces for each *Entity* an abstract class and an aspect. The abstract class (*ResClass*) represents the interface that has to be fulfilled by a concrete implementation for the given *Entity*. This abstract class is comprised by attributes (*ResAttribute*) and methods (*ResMethod*) that represent their *Entity*'s counterparts in the SecureUML policy and moreover: (i) with all attributes declared with protected visibility (that is, only directly accessible by instances of the given class or its heirs) and (ii) with the so called "getters" and "setters" methods for each attribute, that is, methods to, respectively, read and update the state of each attribute.

The *Aspect* generated by the transformation function controls the calls to the methods of the generated *ResClass*. For each *ResMethod* there exists a *Pointcut* and a *BeforeAdvice*. The *Pointcut* is declared as a call to the given *ResMethod*. The *BeforeAdvice* implements the permissions of the resource associated with the *ResMethod*, as follows:

- If the given *ResMethod* is a "getter" method to a *ResAttribute* then the body of the *BeforeAdvice* implements the *AuthorizationConstraints* of the read permissions of the attribute associated with the *ResAttribute* guarded by the given *ResMethod*. (Read permissions are those related with *Atomic Read*, *Attribute Full Access*, *Entity Read* and *Entity Full Access* actions in a SecureUML policy.)
- If the given *ResMethod* is a "setter" method to a *ResAttribute* then the body of the *BeforeAdvice* implements the *AuthorizationConstraints* of the write permissions of the *Attribute* associated with the *ResAttribute* controlled by the given *ResMethod*. (Write permissions are those related with *AtomicUpdate*, *AtomicDelete*, *Attribute Full Access*, *Entity Update* and *Entity Full Access* actions in a SecureUML policy.)
- If the given *ResMethod* is associated with a *Method* then the body of *BeforeAdvice* implements the *AuthorizationConstraints* of the permissions of the *Method* associated with the given *ResMethod*.

The *AuthorizationConstraints* are essentially predicates over the state of their associated *Entity*. We assume, for the sake of simplicity of this explanation, that each of them is already coded in the concrete syntax of our target language. The implementation of an *AuthorizationConstraint* is a condition which first tests for the user's *Role*, with respect to the *AuthorizationConstraint*'s *Role*, and then checks for the *AuthorizationConstraint*'s predicate.

The body of a *BeforeAdvice* is essentially a sequence of conditions. It may return successfully (then allowing a *ResMethod* to be called) if the user has an appropriate *Role* and fulfills the *AuthorizationConstraints* of at least one of the *Permissions* associated with the given *ResMethod*. Otherwise it returns an error (for instance, by raising an exception) if no *Permission* is fulfilled or if the user does not have an appropriate *Role* that copes with any of the *Permissions*. Therefore, the conditions in the body of a *BeforeAdvice* are *ordered* by

the *Role* associated with the *AuthorizationConstraint* that the condition implements, starting from the most powerful one (e.g. an administrator) to the least powerful one (e.g. an operator).

Each *Role* is transformed into a *RoleClass*. The *Role* hierarchy relationship is captured as a *RoleClass* inheritance relationship.

4 Monitoring Transformation Invariants

We have implemented the transformation function described in Section 3.2 as a prototype Java application on top of the OCL evaluator ITP/OCL. Our implementation is a three-tiered application. The higher layer implements the transformation function, the middle layer is a Secure UML policy manager and the bottom layer is an OCL evaluator.

The OCL evaluator is a "wrapper" Java class that provides access to ITP/OCL. Essentially, it defines methods for:

- Creation of class and instance diagrams.
- Creation and deletion of classes, relationships between classes, objects and links between objects.
- Evaluation of OCL queries on instance diagrams.
- Evaluation of OCL invariants on instance diagrams.

The SecureUML policy manager is implemented as a Java class that instantiates the OCL evaluator with the SecureUML metamodel as class diagram. The SecureUML policy manager defines an API to create SecureUML policies and operations to query a SecureUML policy. A SecureUML policy is represented internally as an instance diagram of the SecureUML metamodel. Operations on a SecureUML policy are translated into OCL expressions and executed as queries in the OCL evaluator instance held by the SecureUML policy manager. Moreover, it implements all the invariants and operations over the SecureUML metamodel defined in [7].

The SecureUML to aspect transformer is implemented as a Java class that extends the SecureUML policy manager. Given a SecureUML policy, the transformer instantiates a set of Java classes that faithfully represent the AAC metamodel described in Section 3.1. The instantiation process follows the transformation defined in Section 3.2. The set of Java objects produced in memory by the transformer are then traversed in order to generate an object model of AAC in the underlying instance of the OCL evaluator. Finally, all the invariants are checked and the abstract class and aspect are written into the output using AspectJ syntax together with Java classes for each role with the appropriate inheritance relationship.

The metaclass *Env* is translated to a Java class named *Env* with a single method with signature `Role getUsrRole()`. This method returns the role of the current user. The prototype generates a simple implementation for `getUserRole` just for the purpose of our experiment. This implementation is shared with the application that the generated aspect is connected with. Of course, a more robust

component could have been targeted that takes advantage of user information from the underlying operational system, for example.

Note that the function that produces the concrete syntax in AspectJ could be *overloaded* to produce different concrete syntax for Design by Contract languages such as Eiffel, JML or Spec#. For these languages the generated abstract class would be annotated with preconditions representing the access control assertions. Of course, in this case, the weaving process is not automatic and explicit calls to the methods implementing the pre-conditions would have to be written.

5 An Algebraic Interpretation of the Merged Metamodel and Transformation Invariants

Our approach is a particular instantiation of a general process for OCL evaluation over object diagrams which is formalized algebraically and implemented using rewriting techniques [9]. In this section we first outline the process and its algebraic specification [14] and then relate the concepts of merged metamodel and transformation invariants with elements in this general process.

The OCL evaluation process comprises, essentially, the generation of equational specifications from class diagrams, object diagrams, and OCL expressions and then evaluation of the OCL expressions by equational simplification using standard rewriting techniques.

A class diagram is captured as an order-sorted signature in an equational theory that includes other equational theories for the UML predefined types, declares sorts for each class in the class diagram, declares sorts for the collections of each class in the class diagram, declares operators for each class attribute, and operators for the roles on each relation in the class diagram.

An object diagram, which is also represented as an equational theory, first includes the equational theory that represents its associated class diagram, in extending mode, that is, new sort elements may be introduced but without identifying previously defined ones. Second, it defines constant operations to represent each object in the object diagram, of the appropriate sort, that is, the sort that represents the object's class. Third, it defines equations relating the operators that capture the relations' roles with the constant operators that represent the objects, or collections of them.

An OCL expression is also formalized as an equational theory. Recall that an OCL expression is evaluated in the context of an object diagram. Therefore, the equational theory for an OCL expression first extends the equation theory for the object diagram that represents the context of evaluation of the OCL expression. Second, a number of OCL operations are overloaded for the sorts that represent the classes in the class diagram associated with the given object diagram. Third, a number of equations are defined to implement the necessary OCL primitive functions (such as *allInstances* or *includes*) used in the given OCL expression, for the given object diagram, together with equations that represent the actual OCL expression.

The evaluation process for OCL expressions may be summarized with the following formula:

$$QueryTheory(\mathcal{OD}, e) \vdash QueryTerm(e) = AsTerm(v)$$

where $QueryTheory(\mathcal{OD}, e)$ is the equational theory resulting from the application of the mapping $QueryTheory$ to the object diagram $\mathcal{OD}$ and the OCL expression e, that is, the process described in the previous paragraphs; $QueryTerm(e)$ is a term in the theory $QueryTheory(\mathcal{OD}, e)$ representing an OCL expression and $AsTerm(v)$ is also a term in the theory $QueryTheory(\mathcal{OD}, e)$ that represents the value to be produced by the evaluation of the OCL expression e.

In our experiment, the merged metamodel is the equational theory resulting from the inclusion in *protected* mode of the equational theory that represents the class diagram for the metamodel of SecureUML with the equational theory that represents the class diagram of the metamodel of AAC. Inclusion in protected mode is more restrictive than in extending mode. It requires that no new sort elements are added and that no old elements are identified with new ones. Our prototype does not check for inclusion in protected mode as it would require theorem proving [3]. A solution could be to apply renaming to the model elements to guarantee uniqueness. This is not currently implemented. Note, however, that this would be the case for the general merging of any given two models. We manually guarantee it while merging the metamodels for SecureUML and AAC.

The transformation invariant is the equational theory resulting from the transformation of the associated OCL expressions. The validation of the transformation invariant is given by the equational simplification of the *QueryTerm* that represents the transformation invariant in the context of the equational theory given by the extension of the equational theory of the merged metamodel with the equational theory that represents the object diagram containing the given security policy and the generated abstract class and aspect.

6 Final Remarks

The use of aspects in security is among the most successful uses of the AOP paradigm. In particular, we refer to [20, 15, 21] as representatives of the use of AOP in access control.

In [20] the authors formalise *role-slices* to specify access control policies and a compilation process from access control policies into aspect code. The compilation process is described as a functional program. The process appears to be quite precise however there is no indication of a correctness proof or any validation of the proposed process. In [21] the authors propose the use of aspects at the modeling level (similarly to [15]) but analysis is left to future work.

The approach followed in [15], we quote, "translates security aspects specified as UMLsec stereotypes as concrete security mechanisms on the modelling level". They analyze, with a theorem prover, first-order logic (FOL) formulae generated out of control flow graphs obtained from the produced source code and associated security requirements. Their approach appears to be similar to [1, 5] where

the target of the transformation process is a FOL theory that can be reasoned about. Our approach is part of the so called *lightweight* formal methods. We aim at *validation* instead of the *verification* approach in [15, 1, 5]. Our approach allows the validation of each and every instance of the application of the transformation function over an access control policy and generated (abstract class and) aspect. We do not aim at allowing for the proof of general (inductive) properties that would require theorem proving but rather an approach following the pragmatic ideas of Design by Contract (DbC). In [1] the authors propose two transformations from SecureUML policies targeting two different object-oriented frameworks. We complement that effort with another transformation to aspects that further extends the code generation ideas in [7, 8]. Moreover, we extend [7] by applying validation to transformation invariants.

What appears to be novel in our MDS approach from RBAC policies to code is to generate *validated* AspectJ code. The use of aspects *modularly* code permission's constraints. We validate our transformation using a *merged* metamodel of the abstract syntax of the languages involved, focusing on the transformation invariants that specify structural constraints that the implementation of the transformation has to cope with. We do not commit ourselves to any particular transformation specification language or implementation language. The specification of the transformation occurs under the same model-based approach used during design. The transformation invariants are then checked by a tool during runtime, following DbC's run-time assertion monitoring idea. Also, the generation of different concrete syntax, besides AspectJ, can be targeted such as a DbC-based language.

We believe that our approach thus contributes to the efforts both of code generation for model driven security [1] and perhaps to model driven architecture itself, in the context of [2]. We foresee the continuation of this work with more experimentation, in particular on the exploitation of the notion of transformation contracts over merged metamodels, and by enhancing our tool support for OCL evaluation and SecureUML policy manager, both in terms of efficiency.

Acknowledgements I would like to thank Viviane Silva and Marina Egea for their comments on a draft of this paper. Jorge Ogalla Ramírez and Pedro Díaz Yeregui work on the implementation on the prototype is also acknowledged.

References

1. D. A. Basin, J. Doser, and T. Lodderstedt. Model driven security: From UML models to access control infrastructures. *ACM Transactions on Software Engineering and Methodology*, 15(1):39–91, 2006.
2. J. Bézivin, F. Butner, M. Gogolla, F. Jouault, I. Kurtev, and A. Lindow. Model transformations? Transformation models! In O. Nierstrasz, editor, *Proceedings of 9th International Conference on Model Driven Engineering Languages and System, MoDELS 2006, Genova, Italy, October 1-6*, volume 4199 of *Lecture Notes in Computer Science*, pages 440–453. Springer-Verlag, 1006.

3. A. Bouhoula, J.-P. Jouannaud, and J. Meseguer. Specification and proof in membership equational logic. *Theoretical Computer Science*, 236(1-2):35–132, 2000.
4. C. Braga, M. Clavel, F. Durán, S. Eker, A. Farzan, J. Hendrix, P. Lincoln, N. Martí-Oliet, J. Meseguer, , P. Olveczky, M. Palomino, R. Sasse, M.-O. Stehr, C. Talcott, and A. Verdejo. *All About Maude: A High-Performance Logical Framework*, volume 4350 of *Lecture Notes in Computer Science*, chapter 21, pages 667–693. Springer, 2007. `http://dx.doi.org/10.1007/978-3-540-71999-1_21`.
5. A. D. Brucker, J. Doser, and B. Wolff. A model transformation semantics and analysis methodology for Secure UML. In O. Nierstrasz, J. Whittle, D. Harel, and G. Reggio, editors, *Proceedings of the 9th International Conference on Model Driven Engineering Languages and Systems, Genova, Italy, October 1 - 6*, volume 4199 of *Lecture Notes in Computer Science*, pages 306–320. Springer-Verlag, 2006.
6. K. Chen and C.-W. Lin. An aspect-oriented approach to declarative access control for web applications. In X. Zhou, J. Li, H. T. Shen, M. Kitsuregawa, and Y. Zhang, editors, *Proceedings of the 8th Asia-Pacific Web Conference, Harbin, China, January 16-18*, volume 3841 of *Lecture Notes in Computer Science*, pages 176–188. Springer-Verlag, 2006.
7. M. Clavel, D. Basin, J. Doser, and M. Egea. Automated analysis of security-design models. *To appear in Information and Software Technology*, 2008. `http://maude.sip.ucm.es/~clavel/pubs/BCDE07-journal.pdf`.
8. M. Clavel, V. da Silva, C. Braga, and M. Egea. Model-driven security in practice: an industrial experience. In *Proceedings of ECMDA 2008, Fourth European Conference on Model Driven Architecture, Foundations and Applications, Berlin, Germany, June 09-12*, Lecture Notes in Computer Science. To appear, Springer, 2008. `http://maude.sip.ucm.es/~clavel/pubs/CSBE08.pdf`.
9. M. Clavel and M. Egea. Equational specification of UML+OCL static class diagrams. `http://maude.sip.ucm.es/~clavel/pubs/clavel-egea06a.pdf`, 2006.
10. M. Clavel and M. Egea. ITP/OCL: A rewriting-based validation tool for uml+ocl static class diagrams. In *Proceedings of 11th International Conference on Algebraic Methodology and Software Technology, AMAST 2006, Kuressaare, Estonia, July 5-8*, number 4019 in Lecture Notes in Computer Science, pages 368–373. Springer, 2006.
11. B. de Win, B. Vanhaute, and B. D. Decker. Security through aspect-oriented programming. In *Proceedings of the IFIP TC11 WG 11.4 First Annual Conference on Netwrok Security: Advances in Network and Distributed Systems Security*, volume 206, pages 125–138, 2001.
12. D. F. Ferraiolo, D. R. Kuhn, and R. Chandramouli. *Role-Based Access Control*. Artech House Publishers, 2nd edition, 2007.
13. S. Gao, Y. Deng, H. Yu, X. He, K. Beznosov, and K. Cooper. Applying aspect-orientation in designing security systems: A case study. In *Proceedings of 16th International Conference on Software Engineering and Knowledge Engineering, Banff, Alberta, Canada, june 20-24*, pages 360–365, 2004.
14. J. A. Goguen and J. Meseguer. Order-sorted algebra I: equational deduction for multiple inheritance, overloading, exceptions and partial operations. *Theoretical Computer Science*, 105(2):217–273, 1992.
15. J. Jurjens and S. H. Houmb. Dynamic secure aspect modeling with UML: From models to code. In L. Briand and C. Williams, editors, *Proceedings of 8th International Conference on Model Driven Engineering Languages and System, MoDELS 2005, Montego Bay, Jamaica, October 2-7*, volume 3713 of *Lecture Notes in Computer Science*, pages 142–155. Springer-Verlag, 2005.

16. G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. An overview of AspectJ. *Lecture Notes in Computer Science*, 2072:327–355, 2001.
17. B. Meyer. *Object-Oriented software construction*. Prentice Hall, 2nd. edition, 1997.
18. J. Miller and J. Mukerji, editors. *MDA Guide (Version 1.0.1)*. Number omg/2003-06-01. OMG, 2006.
19. Object Management Group. *MOF QVT Final Adopted Specification, OMG Adopted Specification ptc/05-11-01*, 2005.
20. J. Pavlich-Mariscal, L. Michel, and S. Demurjian. A formal enforcement framework for role-based access control using aspect-oriented programming. In L. Briand and C. Williams, editors, *Proceedings of 8th International Conference on Model Driven Engineering Languages and System, MoDELS 2005, Montego Bay, Jamaica, October 2-7*, volume 3713 of *Lecture Notes in Computer Science*, pages 537–552. Springer-Verlag, 2005.
21. I. Ray, R. France, N. Li, and G. Georg. An aspect-based approach to modeling access control concerns. *Information and Software Technology*, 46(9):575–587, 2004.
22. R. S. Sandhu, E. J. Coyne, H. L. Feinstein, and C. E. Youman. Role-based access control models. *IEEE Computer*, 29(2):38–47, February 1996.