Formal Reasoning about Fine-Grained Access Control Policies

Miguel A. García de Dios **Carolina Dania** **Manuel Clavel**

IMDEA Software Institute, Spain
{miguelangel.garcia,carolina.dania,manuel.clavel}@imdea.org

Abstract

Nowadays, most of the main database management systems offer, in one way or another, the possibility of protecting data using fine-grained access control (FGAC) policies, i.e., policies that depend on dynamic properties of the system state. Reasoning about FGAC policies typically amounts to answering questions about whether a security-related property holds in a (possibly infinite) set of system states. To carry out this reasoning, we propose a novel, tool-supported methodology, which consists in transforming the aforementioned questions about FGAC policies into satisfiability problems in first-order logic. In addition, we report on our experience using the Z3 Satisfiability Modulo Theory (SMT) solver for automatically checking the satisfiability of the first-order formulas generated by the tool implementing our methodology, called SecProver, for a non-trivial set of examples.

1 Introduction

In Role-Based Access Control (RBAC) (Ferraiolo et al. 2001), permissions specify which roles are allowed to perform given operations. These roles typically represent job functions within an organization. Users are granted permissions by being assigned to the appropriate roles based on their competencies and responsibilities in the organization. RBAC allows one to organize the roles in a hierarchy where roles can inherit permissions along the hierarchy. In this way, the security policy can be described in terms of the hierarchical structure of an organization. However, it is not possible in RBAC to specify *fine-grained access control* (FGAC) policies, i.e., policies that depend on dynamic properties of the system state, for example, to allow an operation only during weekdays.

SecureUML (Basin et al. 2006) is a modeling language for formalizing FGAC policies. It extends RBAC with *authorization constraints*, which allow one to specify constraints for granting permissions. Authorization constraints are formalized in SecureUML using the Object Constraint Language (OCL) (Object Management Group 2014). Using SecureUML, one can then model access control decisions that depend on two kinds of information:

1. *static information*, namely the assignments of users and permissions to roles, and the role hierarchy, and
2. *dynamic information*, namely the satisfaction of authorization constraints in the given system state.

SecureUML is currently supported by ActionGUI (Basin et al. 2014, ActionGUI 2012), a model-driven methodology for developing secure data-management applications. In ActionGUI, system developers proceed by modeling three different views of the desired application: its data model, security model, and GUI model. These models formalize respectively the application's data domain, authorization policy, and its graphical interface together with the application's behavior. Afterwards a model-transformation function lifts the policy specified by the security model to the GUI model. Finally, a code generator generates a multi-tier application, along with all support for fine-grained access control, from the security-aware GUI model.

In this paper we propose a novel methodology for carrying out formal reasoning about FGAC policies specified using SecureUML. Reasoning about FGAC policies typically amounts to answering questions about whether a security-related property holds in a (possibly infinite) set of states. The key component of our methodology is a mapping from OCL to first-order logic (Clavel et al. 2009, Dania & Clavel 2013), thanks to which we are able to transform the aforementioned questions about FGAC policies into satisfiability problems in first-order logic. Finally, to validate our methodology, we have implemented a tool, called SecProver (SecProver 2014), and we have applied the Z3 SMT solver (de Moura & Bjørner 2008) for automatically checking the satisfiability of the first-order formulas generated by SecProver, for a non-trivial set of security-related questions about SecureUML models.

Organization. In Section 2 we provide background information about SecureUML, and we also discuss its semantics and compare its expressiveness with that of other languages currently supported by commercial database management systems. In Section 3 we summarize the key principles underlying our mapping from OCL to first-order logic. Then, in Section 4, we explain how, using the aforementioned mapping, interesting questions about SecureUML models can be translated into satisfiability problems in first-order

This work is partially supported by the Spanish Ministry of Economy and Competitiveness Project "StrongSoft" (TIN2012-39391-C04-01 and TIN2012-39391-C04-04).

logic, and, in Section 5, we report on our experience using the Z3 SMT solver for automatically checking the satisfiability of the formulas generated by our methodology, for a non-trivial set of examples. We conclude the paper with sections on related work and future work.

2 Modeling Fine-Grained Access Control Policies

SecureUML (Basin et al. 2006) is a modeling language for specifying fine-grained access control policies (FGAC) for actions on protected resources. Using SecureUML one can model *roles* (with their hierarchies), *permissions*, *actions*, *resources*, and constraints on the permissions, which are called *authorization constraints*. SecureUML is, however, *generic* in that it leaves open the nature of the protected resources, i.e., whether these resources are data, business objects, processes, controller states, etc. Basin et al. (2006) initially combined SecureUML with a design modeling language based on class diagrams, called ComponentUML, and with a language based on state diagrams, called ControllerUML. More recently, Basin et al. (2014) have combined SecureUML with a language for modeling graphical user interfaces for data-centric applications, called ActionGUI. In this work, we will use the aforementioned combination of SecureUML with ComponentUML, called SecureUML+ComponentUML.

Next, we will explain, and illustrate with examples, the main concepts used when modeling with SecureUML+ComponentUML: namely, resources and actions; invariants; authorization constraints; and permissions. Also, we will briefly compare SecureUML+ComponentUML with other languages supported by commercial data management systems for specifying FGAC policies

2.1 Resources and Actions

ComponentUML provides a subset of UML class models where entities (classes) can be related by associations and may have attributes. In SecureUML+ComponentUML, the protected resources are the ComponentUML entities, along with their attributes and association-ends (but not the associations as such), and the actions that they offer to the actors are those shown in the following table:

Resource	Actions
Entity	create, delete
Attribute	read, update
Association-end	read, create, delete

Example 1 In Figure 1 we show a ComponentUML model, named `EmplBasic.dtm`. This model specifies that every employee may have a name, a surname, and a salary; that every employee may have a supervisor and may in turn supervise other employees; and that every employee may take one of two roles: `Worker` or `Supervisor`. In the terminology of ComponentUML, `Employee` is an *entity*; `name`, `surname`, `salary`, and `role` are *attributes*; `supervisedBy` and `supervises` are *association-ends*; and `Role` is an *enumerated class*. Notice that the association-end `supervises` has multiplicity `0..*`, meaning that an employee may supervise zero or more employees, while the association-end `supervisedBy` has multiplicity `0..1` meaning that an employee may have at most one supervisor.

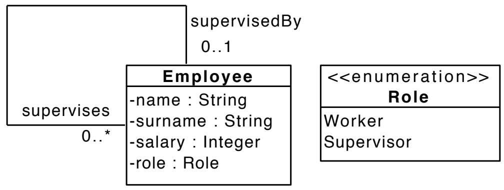
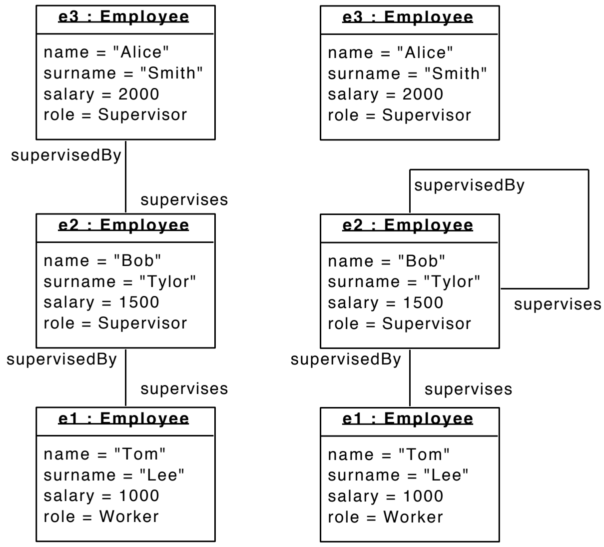


Figure 1: `EmplBasic.dtm`: a ComponentUML model for employees' information.



(a) Instance 1 (b) Instance 2

Figure 2: Two instances of `EmplBasic.dtm`

As expected, the *instances* of ComponentUML models are, basically, UML object models where objects can be related by links and can have values for their attributes.

Example 2 In Figure 2 we show two different instances of `EmplBasic.dtm`. In Instance 2a there are three employees, e_1, e_2 and e_3, and e_1 is supervised by e_2, e_2 is supervised by e_3, and e_3 has no supervisor at all. Moreover, e_1 has role `Worker` and both e_2 and e_3 have role `Supervisor`. Instance 2b has also three employees, e_1, e_2 and e_3, but this time e_1 is supervised by e_2, e_2 is supervised by itself, and e_3 has no supervisor at all. As before, e_1 has role `Worker` and both e_2 and e_3 have role `Supervisor`.

2.2 Invariants

ComponentUML models can be further refined by adding to them *invariants*, i.e., expressions specifying properties that every *valid* instance of the model must satisfy. Invariants are formalized in ComponentUML using the Object Constraint Language (OCL) (Object Management Group 2014).

OCL is a strongly typed, declarative language: expressions either have a primitive type (integer, real, string, or boolean), an entity type, a tuple type, or a collection type (set, bag, or collection). It provides standard operators on collections, such as $\rightarrow$`isEmpty`, $\rightarrow$`includes`, and $\rightarrow$`excluding`, as well as operators to iterate over collections, such as $\rightarrow$`forAll`, $\rightarrow$`exists`, and $\rightarrow$`select`. It also provides standard operators on primitive data and tuples, and a dot-operator to access the values of the objects' attributes and association-ends. Moreover, it includes two constants, `null` and `invalid`, to represent *undefinedness*. Intuitively, `null` represents unknown or undefined values, whereas `invalid` represents error and exceptions. To check if a value is `null` or `invalid`, it provides, respectively, the boolean operators `oclIsUndefined()` and `oclIsInvalid()`.

Example 3 We can refine the model `EmplBasic.dtm` (Figure 1) by adding invariants to this model. In particular, consider the following constraints:

1. There is exactly one employee who has no supervisor.
2. Nobody is its own supervisor.
3. An employee has role `Supervisor` if and only if it has at least one supervisee.
4. Every employee has one role.

These constraints can be formalized in OCL as follows:

```
(1) Employee.allInstances()→one(e|
    e.supervisedBy.oclIsUndefined())
(2) Employee.allInstances()→forAll(e|
    not(e.supervisedBy = e))
(3) Employee.allInstances()→forAll(e|
    (e.role = Supervisor implies
        e.supervises->notEmpty())
    and (e.supervises->notEmpty()
        implies e.role = Supervisor))
(4) Employee.allInstances()→forAll(e|
    not(e.role.oclIsUndefined())
```

In what follows, we will refer to the constraint (1) as `oneBoss`, (2) as `noSelfSuper`, (3) as `roleSuper`, and (4) as `allRole`. Also, we will denote by `Empl1.dtm` the refined version of `EmplBasic.dtm` that includes as invariants the constraints `oneBoss`, `noSelfSuper`, `roleSuper`, and `allRole`. Notice that these four constraints evaluate to `true` in Instance 2a of `EmplBasic.dtm` (Figure 2), and therefore we say that this instance is a *valid* instance of `Empl1.dtm`. On the other hand, since `noSelfSuper` and `roleSuper` evaluate to `false` in Instance 2b of `EmplBasic.dtm` (Figure 2), we say that this other instance is a not a valid instance of `Empl1.dtm`.

2.3 Authorization Constraints

In SecureUML+ComponentUML, authorization constraints specify the conditions that need to be satisfied for a permission being granted to an actor (user) who requests it to perform an action. They are formalized using OCL, but they can also contain the following keywords:

- `self`: it refers to the root resource upon which the action will be performed, if the permission is granted. The root resource of an attribute or an association-end is the entity to which it belongs.
- `caller`: it refers to the actor that will perform the action, if the permission is granted.
- `value`: it refers to the value that will be used to update an attribute, if the permission is granted.
- `target`: it refers to the object that will be linked at the end of an association, if the permission is granted.

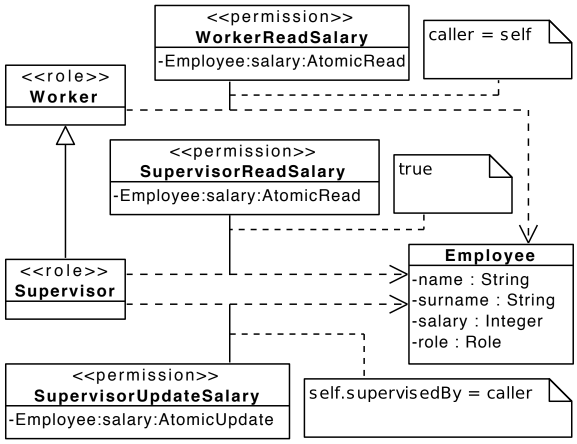


Figure 3: `Empl.stm`: a SecureUML+ComponentUML model for accessing employees' information.

Example 4 In Figure 3 we show a SecureUML+-ComponentUML model, named `Empl.stm`. This model specifies a basic FGAC policy for accessing the employees' information modeled in `Empl1.dtm`. Permissions are assigned to users depending on their *roles*, which can be `Worker` or `Supervisor`. Also, users with role `Supervisor` *inherit* all the permissions granted to users with role `Worker`, since `Supervisor` is a *subrole* of `Worker`. Finally, permissions are constrained by *authorization constraints*: namely,

1. A worker is granted permission to read an employee's salary, provided that it is its own salary, as specified by the authorization constraint `caller = self`.
2. A supervisor is granted unrestricted permission to read an employee's salary, as specified by the authorization constraint `true`.
3. A supervisor is granted permission to update an employee's salary, provided that it supervises this employee, as specified by the authorization constraint `self.supervisedBy = caller`.

2.4 Permissions

SecureUML+ComponentUML provides various syntactic sugar constructs for expressing FGAC policies in a more compact way. Basically, in the 'sweeter' presentation of a model, some roles may not have *explicitly* assigned any permission for some actions, while the following always holds in the de-sugared presentation of the model: every role has assigned exactly one permission for every action, and this permission has assigned exactly one authorization constraint.

Next, we will explain, and illustrate with examples, the rules that we apply for de-sugaring a SecureUML+ComponentUML model $\mathcal{S}$:

- *Role hierarchies.* Let *act* be an action and let r and r' be two roles. Suppose that r is a subrole of r' in $\mathcal{S}$, and that there is a permission in $\mathcal{S}$ for r' to execute *act* under the constraint *auth*. Then, when de-sugaring $\mathcal{S}$, we add a new permission to $\mathcal{S}$ for the role r to execute *act* under the same constraint *auth*.
- *Delete actions.* Let *entity* be an entity. Let *act* be the action delete(*entity*). Suppose that there is a permission in $\mathcal{S}$ for a role r to execute *act* under the constraint *auth*. Then, when

de-sugaring $\mathcal{S}$, for every association-end *assoc* owned by *entity*, we add to $\mathcal{S}$ a new permission for r to execute delete(*assoc*) under the same constraint *auth*.

- *Opposite association-ends.* Let *assoc* and *assoc'* be two opposite association-ends. Let *act* be the action create(*assoc*). Suppose that there is a permission in $\mathcal{S}$ for a role r to execute *act* under the constraint *auth*. Then, when de-sugaring $\mathcal{S}$, we add to $\mathcal{S}$ a new permission for the role r to execute create(*assoc'*) under the constraint that results from replacing in *auth* the variable `self` by `target` and the variable `target` by `self`. De-sugaring is done similarly when *act* is the action delete(*assoc*).

- *Denying by default.* Let r be a role and let *act* be an action. Suppose that there is no permission in $\mathcal{S}$ for the role r to execute *act*. When de-sugaring $\mathcal{S}$, we add to $\mathcal{S}$ a new permission for the role r to execute *act* under the constraint `false`.

- *Disjunction of authorization constraints.* Let r be a role and let *act* be an action. Suppose that there are n permissions in $\mathcal{S}$ for the role r to execute *act*. When de-sugaring $\mathcal{S}$, we replace these n permissions by a new permission and assign to it the authorization constraint that results from disjoining together all the authorization constraints of the original n individual permissions.

In what follows, we will denote by $\text{Auth}(\mathcal{S}, r, act)$ the authorization constraint assigned, in the de-sugared presentation of the ComponentUML+SecureUML model $\mathcal{S}$, to the role r's permission for performing the action *act*.

Example 5 Consider the value of $\text{Auth}(\mathcal{S}, r, act)$ in the following cases:

Auth(`Empl.stm`, `Worker`, update(`salary`))= `false`,

by the rule "denying by default".

Auth(`Empl.stm`, `Supervisor`, update(`salary`))=
`self.supervisedBy = caller or false`,

by the combination of the rules "denying by default", "role hierarchies", and "disjunction of authorization constraints".

Auth(`Empl.stm`, `Worker`, read(`salary`))=
`caller = self`.

Auth(`Empl.stm`, `Supervisor`, read(`salary`))=
`caller = self or true`,

by the combination of the rules "denying by default", "role hierarchies", and "disjunction of authorization constraints".

2.5 Expressiveness

Traditionally, database management systems (DBMS) support 'all-or-nothing' access control with respect to the cells in the column of a table, i.e., a policy will either give or deny access to all the cells in the column. Nowadays, however, some of the main commercial DBMS also support fine-grained access control, which means that a policy can also give access only to a subset of the cells of the column. Next, we will provide some initial comparison between SecureUML+ComponentUML and the languages currently supported by Oracle Virtual Private Database (Huey 2014), IBM/DB2 (IBM 2013), Microsoft SQL Server (SQL 2012), and Teradata (Teradata 2014) for specifying FGAC policies.

Oracle supports FGAC in its Virtual Private Database (VPD) through the use of *security policy functions* (SPF), which are written in Oracle PL/SQL. The idea is that when a user executes a statement, the corresponding SPF is transformed into a `WHERE` clause and is added to the user's original statement. Clearly, authorization constraints play the same role as SPFs, and we conjecture, based on our experience mapping OCL into SQL (Egea et al. 2010), that any SPFs written in declarative SQL could be formalized as an authorization constraint written in OCL. However, since SPFs are written in PL/SQL, they would typically contain non-declarative code.

IBM/DB2 implements FGAC through the use of *row access control* and *column access control rules*. They specify, respectively, which rows and columns can be accessed and under which conditions. Again, authorization constraints play the same role as row and column access rules, and we also conjecture that any combination of row and column access control rules written in declarative SQL could be formalized as an authorization constraint written in OCL. On the other hand, and differently from SecureUML+ComponentUML, IBM/DB2 only supports column access control rules for `SELECT` statements, and, therefore, they can only be used, in general, to protect read-actions over attributes.

Finally, both Microsoft SQL Server and Teradata support FGAC policies through the use of *security labels*, which can be assigned to users and resources, and *constraints*. In SecureUML+ComponentUML, security labels can be represented as additional attributes of the entities representing users and resources, and constraints can then be formalized as OCL authorization constraints referring to the values of these additional attributes. On the other hand, security labels can only be assigned to entities, and therefore, they can not be used to protect read- or update-actions over attributes.

3 Mapping OCL to First-Order Logic

In previous work (Clavel et al. 2009, Dania & Clavel 2013) we proposed a mapping from OCL to first-order logic, which consists of two, inter-related components: (i) a map from ComponentUML models and boolean OCL expressions to first-order formulas, called $\text{ocl2fol}_{\text{def}}$; and (ii) a map from boolean OCL expressions to first-order formulas, called ocl2fol. The following remark formalizes the main property of our mapping from OCL to first-order logic.

Remark 1 Let $\mathcal{D}$ be a ComponentUML model, with invariants $expr_1, \ldots, expr_n$, and let *expr* be a boolean expression. Then, *expr* evaluates to `true` in every valid instance of $\mathcal{D}$ if and only if

$$\begin{aligned}&\text{ocl2fol}_{\text{def}}(\mathcal{D})\\ &\quad\cup \bigcup_{i=1}^{n} \text{ocl2fol}_{\text{def}}(expr_i) \cup \bigcup_{i=1}^{n} \{\text{ocl2fol}(expr_i)\}\\ &\quad\cup\, \text{ocl2fol}_{\text{def}}(expr) \cup \{\neg(\text{ocl2fol}(expr))\}\end{aligned}$$

is unsatisfiable.

Next, we will explain, and illustrate with examples, the main ideas behind the maps $\text{ocl2fol}_{\text{def}}$ and ocl2fol. We refer the interested reader to the original papers (Clavel et al. 2009, Dania & Clavel 2013) for a more formal presentation of these maps and of the subset of OCL that they currently support.

3.1 The map $\mathrm{ocl2fol}_{\mathrm{def}}$ (models)

In our mapping from OCL to first-order logic, we represent entities by predicates, attributes by functions, and association-ends, depending on their multiplicity, either by binary predicates or by functions. Also, we represent `null` and `invalid`, respectively, by the constants null and invalid, and we introduce two unary predicates IsNull and IsInvalid, to represent when an element is `null` or `invalid`.

Let $\mathcal{D}$ be a ComponentUML model. $\mathrm{ocl2fol}_{\mathrm{def}}(\mathcal{D})$ returns the axioms formalizing the properties of the predicates and functions that represent the entities, attributes and association-ends in $\mathcal{D}$, as well as the axioms formalizing the constants null and invalid, and the predicates IsNull and IsInvalid.

Example 6 Consider the ComponentUML model `EmplBasic.dtm` shown in Figure 1. Among others, $\mathrm{ocl2fol}_{\mathrm{def}}$(`EmplBasic.dtm`) returns the axiom

$$\forall(x)(\mathrm{Employee}(x) \Rightarrow \neg(\mathrm{isNull}(x) \vee \mathrm{isInvalid}(x))),$$

which formalizes that neither `null` nor `invalid` are objects of the entity `Employee`, as well as the axiom

$$\forall(x)\forall(y)(\mathrm{supervises}(y, x) \Rightarrow (\mathrm{supervisedBy}(x) = y)),$$

which formalizes the key property of `supervises` as the opposite association-end of `supervisedBy`.

The following remark formalizes the main property of the map $\mathrm{ocl2fol}_{\mathrm{def}}$.

Remark 2 Let $\mathcal{D}$ be a ComponentUML model. Then, there is a one-to-one correspondence between the instances of $\mathcal{D}$ and the first-order models that satisfy $\mathrm{ocl2fol}_{\mathrm{def}}(\mathcal{D})$.

3.2 The map ocl2fol

In our mapping from OCL to first-order logic, we represent boolean expressions as formulas.

Let *expr* be a boolean expression. ocl2fol(*expr*) is defined recursively over the structure *expr*, according to the following principles:

- Each subexpression C`.allInstances()` is represented by a predicate formula whose predicate is the one representing the entity C.
- Each boolean subexpression is represented by a formula which mirrors its logical structure.
- Each integer subexpression is represented by the corresponding functional expression.
- Each set subexpression is represented by a predicate formula whose predicate's definition is generated using the map $\mathrm{ocl2fol}_{\mathrm{def}}$ (see below).

Example 7 Consider the constraints `oneBoss` and `noSelfSuper` introduced in Example 3. ocl2fol(`oneBoss`) returns the formula

$$\begin{aligned}&\exists(e)(\mathrm{Employee}(e) \wedge \mathrm{isNull}(\mathrm{supervisedBy}(e))\\ &\quad \wedge \forall(e')(\mathrm{Employee}(e') \wedge \mathrm{isNull}(\mathrm{supervisedBy}(e'))\\ &\qquad \Rightarrow\ e' = e)),\end{aligned}$$

and ocl2fol(`noSelfSuper`) returns the formula

$$\forall(e)(\mathrm{Employee}(e) \Rightarrow \neg(\mathrm{supervisedBy}(e) = e)).$$

The following remark formalizes the main property of the map ocl2fol.

Remark 3 Let $\mathcal{D}$ be a ComponentUML model. Let *expr* be a boolean expression. Suppose that *expr* does not contain any subexpression of type collection. Then, there is a one-to-one correspondence between the instances of $\mathcal{D}$ in which the *expr* evaluates to `true` and the first-order models that satisfy $\mathrm{ocl2fol}_{\mathrm{def}}(\mathcal{D}) \cup \{\mathrm{ocl2fol}(\mathit{expr})\}$.

3.3 The map $\mathrm{ocl2fol}_{\mathrm{def}}$ (expressions)

Often, OCL expressions include subexpressions that will evaluate to collections: e.g., those which are built using →`select`, →`collect`, or →`excluding`. In our mapping from OCL to first-order logic, when these subexpressions are of type set, we represent them using new predicates whose definitions, which follow the principles underlying $\mathrm{ocl2fol}_{\mathrm{def}}$, are given by the map $\mathrm{ocl2fol}_{\mathrm{def}}$.

Example 8 Consider the expression

```
Employee.allInstances()→select(e|
    e.supervises→notEmpty()).
```

This expression, which we refer to as `colOfSuper`, will evaluate to a set containing only those employees whose list of supervisees is not empty. Then, $\mathrm{ocl2fol}_{\mathrm{def}}$(`colOfSuper`) returns the following axiom,

$$\begin{aligned}&\forall(x)(\mathrm{P\_colOfSuper}(x) \Leftrightarrow\\ &\qquad (\mathrm{Employee}(x) \wedge \exists(y)(\mathrm{supervises}(x, y)))),\end{aligned}$$

where the new predicate P_colOfSuper represents the set that will be returned when evaluating `colOfSuper`.

The following remark formalizes the main property of the map $\mathrm{ocl2fol}_{\mathrm{def}}$ over expressions of type set.

Remark 4 Let $\mathcal{D}$ be a ComponentUML model. Let *expr* be an expression of type set. Let P_*expr* be the predicate symbol generated by ocl2fol(*expr*). Then, there is a one-to-one correspondence between the instances of $\mathcal{D}$ and the first-order models that satisfy $\mathrm{ocl2fol}_{\mathrm{def}}(\mathcal{D}) \cup \mathrm{ocl2fol}_{\mathrm{def}}(\mathit{expr})$ for which the following holds: *expr* evaluates to $\{o_1, \ldots, o_n\}$ in $\mathcal{I}$ if and only if the element that corresponds to o_i belongs to the set that interprets P_*expr*, for $i = 1, \ldots, n$.

3.4 Reasoning about Data Models

Here we provide a simple example of the use of our mapping from OCL to first-order logic for reasoning about ComponentUML models.

In what follows, when a ComponentUML model $\mathcal{D}$ contains invariants $\mathit{expr}_1, \ldots, \mathit{expr}_n$, we will consider that $\mathrm{ocl2fol}_{\mathrm{def}}(\mathcal{D})$ includes also the formulas $\bigcup_{i=1}^{n} \mathrm{ocl2fol}_{\mathrm{def}}(\mathit{expr}_i)$.

Example 9 Consider the following question about the model `Empl1.dtm` in Example 3: *Is there a valid instance in which someone is supervised by one of its own supervisees?* Let us formalize the property that no employee is supervised by their own supervisees as follows:

```
Employee.allInstances()→forAll(e|
    e.supervises→excludes(e.supervisedBy)).
```

We will refer to this expression as `noMixSuper`. Then, according to Remark 1, the answer to our question is 'Yes' since

$$\text{ocl2fol}_{\text{def}}(\texttt{Empl1.dtm}) \cup \{\neg(\text{ocl2fol}(\texttt{noMixSuper}))\}.$$

is satisfiable. Indeed, consider, for example, an instance of `Empl1.dtm` with just four employees, e_1, e_2, e_3, and e_4, such that e_1 is linked through the association-end `supervisedBy` with e_4, and similarly e_3 with e_2, and e_2 with e_3. Suppose also that e_1 is of role `Worker`, and e_2, e_3, and e_4 are of role `Supervisor`. This instance is certainly a valid one, since all the invariants evaluate to `true`. However, the expression `noMixSuper` evaluates to `false` because e_2 is linked through `supervisedBy` with e_3, but at the same time e_2 is also linked through the association-end `supervises` with e_3 (since e_3 is linked through `supervisedBy` with e_2).

4 Reasoning about Fine-Grained Access Control Policies

As discussed by Basin et al. (2014), SecureUML+ComponentUML models have a rigorous semantics. In particular, let $\mathcal{S}$ be a SecureUML+ComponentUML model and let $\mathcal{I}$ be an instance of its underlying data model. Also, let u be a user, with role r, and let *act* be an action, with arguments *args*. Then, according to the semantics of SecureUML+ComponentUML, $\mathcal{S}$ authorizes u to execute *act* in $\mathcal{I}$ if and only if $[\text{Auth}(\mathcal{S}, r, act)]^{(u,args)}$ evaluates to `true` in $\mathcal{I}$, where $[\text{Auth}(\mathcal{S}, r, act)]^{(u,args)}$ is the expression that results from replacing in $\text{Auth}(\mathcal{S}, r, act)$ the keyword `caller` by u, and the keywords `self`, `value`, and `target` by the corresponding values in *args*.

In what follows, given a SecureUML+ComponentUML model $\mathcal{S}$, we use the term *scenario* to refer to any valid instance of $\mathcal{S}$'s underlying data model in which a user requests permission to execute an action. For the sake of simplicity, we will assume that neither the user requesting permission nor the resource upon which the action will be executed can be *undefined*.

Next, we will explain, and illustrate with examples, how one can use the mapping from OCL to first-order logic discussed in Section 3 to reason about SecureUML+ComponentUML models. Unless stated otherwise, all our examples refer to the SecureUML+ComponentUML model `Empl.stm` (Example 4). Recall that this model's underlying data model is the ComponentUML model `Empl1.dtm` (Example 3), which includes the invariants `oneBoss`, `noSelfSuper`, `roleSuper`, and `allRole`.

We organize our examples in blocks or categories. In the first block, we are interested in knowing if there is any scenario in which someone with role r will be allowed to execute an action *act*. Notice that, by Remark 1, the answer will be 'No' if and only if the following set of formulas is unsatisfiable:

$$\text{ocl2fol}_{\text{def}}(\mathcal{D}) \cup \{\exists(caller)\exists(self)\exists(target)\exists(value)$$
$$(\text{ocl2fol}(caller\texttt{.role = }r) \wedge \text{ocl2fol}(\text{Auth}(\mathcal{S}, r, act)))\}.$$

Example 10 Consider the following question: *Is there any scenario in which someone with role* `Worker` *is allowed to change the salary of someone else (including itself)?* Recall that

$$\text{Auth}(\texttt{Empl.stm}, \texttt{Worker}, \text{update}(\texttt{salary})) = \texttt{false}.$$

According to Remark 1, the answer to this question is 'No', since the following set of formulas is clearly unsatisfiable:

$$\text{ocl2fol}_{\text{def}}(\texttt{Empl1.dtm}) \cup \{\exists(caller)\exists(self)$$
$$(\text{ocl2fol}(caller\texttt{.role = Worker})$$
$$\wedge\, \text{ocl2fol}(\texttt{false}))\},$$

(Note that ocl2fol(`false`) returns $\perp$.) Indeed, there is no scenario in which the expression `false` can evaluate to `true`.

Example 11 Consider the following question: *Is there any scenario in which someone with role* `Supervisor` *is allowed to change the salary of someone else (including itself)?* Recall that

$$\text{Auth}(\texttt{Empl.stm}, \texttt{Supervisor}, \text{update}(\texttt{salary})) =$$
$$(\texttt{self.supervisedBy = caller or false}).$$

According to Remark 1, the answer to this question is 'Yes', since the following set of formulas is satisfiable:

$$\text{ocl2fol}_{\text{def}}(\texttt{Empl1.dtm}) \cup \{\exists(caller)\exists(self)$$
$$(\text{ocl2fol}(caller\texttt{.role = Supervisor})$$
$$\wedge\, \text{ocl2fol}(self\texttt{.supervisedBy = }caller\texttt{ or false}))\}.$$

(Note that ocl2fol($self$`.supervisedBy =` $caller$) returns $\text{supervisedBy}(self) = caller$). Consider, for example, a scenario with just two employees, e_1 and e_2, such that e_1 is linked with e_2 through the association-end `supervisedBy`. Suppose also that e_1 has role `Worker` and e_2 has role `Supervisor`. Clearly, for $caller = e_2$ and $self = e_1$, the expression $self$`.supervisedBy =` $caller$ evaluates to `true` in this scenario.

Example 12 Consider the following question: *Is there any scenario in which someone with role* `Supervisor` *is allowed to change its own salary?* Notice that in any scenario in which someone is requesting to change its own salary, the values of *self* (i.e., the employee whose salary is to be updated) and *caller* (i.e., the employee who is updating this salary) are the same. According to Remark 1, the answer to this question is 'No', since the following set of formulas is unsatisfiable:

$$\text{ocl2fol}_{\text{def}}(\texttt{Empl1.dtm}) \cup \{\exists(caller)\exists(self)$$
$$(\text{ocl2fol}(caller\texttt{.role = Supervisor})$$
$$\wedge\, \text{ocl2fol}(self\texttt{ = }caller\texttt{ and}$$
$$(self\texttt{.supervisedBy = }caller\texttt{ or false})))\}.$$

Indeed, notice that, in every valid scenario the invariant `noSelfSuper` evaluates to `true`, which implies that there are no values for *caller* and *self* such that the expressions $self$ `=` $caller$ and $self$`.supervisedBy =` $caller$ both evaluate to `true`.

Example 13 Consider the following question: *Is there any scenario in which someone with role* `Supervisor` *is allowed to change the salary of someone who has no supervisor at all?* Notice that in any scenario in which someone (*caller*) is requesting to change the salary of someone (*self*) who has no supervisor at all, the value of $self$`.supervisedBy` must be `null`. According to Remark 1, the answer to this question is 'No', since the following set of formulas is unsatisfiable:

$$\text{ocl2fol}_{\text{def}}(\texttt{Empl1.dtm}) \cup \{\exists(caller)\exists(self)$$
$$(\text{ocl2fol}(caller\texttt{.role = Supervisor})$$
$$\wedge\, \text{ocl2fol}(self\texttt{.supervisedBy = null and}$$
$$(self\texttt{.supervisedBy = }caller\texttt{ or false})))\}.$$

Indeed, notice that, by assumption, *caller* is always a defined object, i.e., it can not be `null`, and therefore, if the expression *self*.`supervisedBy = null` evaluates to `true`, then the expression *self*.`supervisedBy =` *caller* evaluates to `false`.

In our second block of examples, we are interested in knowing if there is any scenario in which someone with role r will not be allowed to execute an action *act*. Notice that, by Remark 1, the answer will be 'No' if and only if the following set of formulas is unsatisfiable:

$$\text{ocl2fol}_{\text{def}}(\mathcal{D}) \cup \{\exists(\mathit{caller})\exists(\mathit{self})\exists(\mathit{target})\exists(\mathit{value})$$
$$(\text{ocl2fol}(\mathit{caller}.\texttt{role = } r) \wedge \neg(\text{Auth}(\mathcal{S}, r, \mathit{act})))\}.$$

Example 14 Consider the following question: *Is there any scenario in which someone with role* `Supervisor` *is not allowed to change the salary of someone else (including itself)?* According to Remark 1, the answer to this question is 'Yes', since the following set of formulas is satisfiable:

$$\text{ocl2fol}_{\text{def}}(\texttt{Empl1.dtm}) \cup \{\exists(\mathit{caller})\exists(\mathit{self})$$
$$(\text{ocl2fol}(\mathit{caller}.\texttt{role = Supervisor}) \wedge$$
$$\neg(\text{ocl2fol}(\mathit{self}.\texttt{supervisedBy = } \mathit{caller} \texttt{ or false})))\}.$$

Consider, for example, a scenario with just three employees, e_1, e_2, and e_3 such that e_1 is linked with e_2 through the association-end `supervisedBy`, and similarly e_2 with e_3; but e_1 is not linked with e_3 through the association-end `supervisedBy`. Suppose that e_2 and e_3 have role `Supervisor` and e_1 has role `Worker`. Clearly, for $\mathit{caller} = e_3$ and $\mathit{self} = e_1$, the expression *self*.`supervisedBy =` *caller* evaluates to `false` in this scenario.

In our third block of examples, we are interested in knowing if there is any scenario in which nobody with role r will be allowed to execute an action *act*. Notice that, by Remark 1, the answer will be 'No' if and only if the following set of formulas is unsatisfiable:

$$\text{ocl2fol}_{\text{def}}(\mathcal{D}) \cup \{\exists(\mathit{self})\exists(\mathit{target})\exists(\mathit{value})\forall(\mathit{caller})$$
$$(\text{ocl2fol}(\mathit{caller}.\texttt{role = } r) \Rightarrow$$
$$\neg(\text{ocl2fol}(\text{Auth}(\mathcal{S}, r, \mathit{act}))))\}.$$

Example 15 Consider the following question: *Is there any scenario in which nobody with role* `Supervisor` *is allowed to change the salary of someone else (including itself)*? According to Remark 1, the answer to this question is 'Yes', since the following set of formulas, which we will refer to as Forms(Ex 15), is satisfiable:

$$\text{ocl2fol}_{\text{def}}(\texttt{Empl1.dtm}) \cup \{\exists(\mathit{self})\forall(\mathit{caller})$$
$$(\text{ocl2fol}(\mathit{caller}.\texttt{role = Supervisor}) \Rightarrow$$
$$\neg(\text{ocl2fol}(\mathit{self}.\texttt{supervisedBy = } \mathit{caller} \texttt{ or false})))\}.$$

Indeed, consider, for example, a scenario with just two employees, e_1 and e_2, such that e_1 is linked with e_2 through the association-end `supervisedBy`. Suppose that e_1 has role `Worker` and e_2 has role `Supervisor`. Clearly, for $\mathit{self} = e_2$, for every value for *caller*, the expression *self*.`supervisedBy` $=$ *caller* evaluates to `false`.

In our fourth block of examples, we are interested in knowing if, in every scenario, there is at least one object upon which nobody with role r will be allowed to execute an action *act*. Notice that, by Remark 1, the answer will be 'Yes' if and only if the following set of formulas is unsatisfiable:

$$\text{ocl2fol}_{\text{def}}(\mathcal{D}) \cup \{\forall(\mathit{self})\exists(\mathit{target})\exists(\mathit{value})\exists(\mathit{caller})$$
$$(\text{ocl2fol}(\mathit{caller}.\texttt{role = } r) \wedge \text{ocl2fol}(\text{Auth}(\mathcal{S}, r, \mathit{act})))\}.$$

Example 16 Consider the following question: *In every scenario, is there at least one employee whose salary can not be changed by anybody with role* `Supervisor`*?* According to Remark 1, the answer to this question is 'Yes', since the following set of formulas is unsatisfiable:

$$\text{ocl2fol}_{\text{def}}(\texttt{Empl1.dtm}) \cup \{\forall(\mathit{self})\exists(\mathit{caller})$$
$$(\text{ocl2fol}(\mathit{caller}.\texttt{role = Supervisor}) \wedge$$
$$\text{ocl2fol}(\mathit{self}.\texttt{supervisedBy = } \mathit{caller} \texttt{ or false}))\}.$$

Indeed, notice that in every valid scenario the invariant `oneBoss` evaluates to `true`, which means that there is one employee in the scenario who has no supervisor. In other words, for every valid scenario, we can find a value for *self* such that no value for *caller* can be found such that the expression *self*.`supervisedBy =` *caller* evaluates to `true`.

To end this section, we want to illustrate the importance of taking into account the invariants of the underlying data model when reasoning about FGAC policies. Let `Empl2.dtm` be the ComponentUML model that results from adding to the model `EmplBasic.dtm` (Example 1) the invariants `noSelfSuper`, `roleSuper`, `allRole`, plus the following invariant:

5. Everybody has one supervisor.

This invariant, which we will refer to as `allSuper`, can be formalized in OCL as follows:

```
Employee.allInstances()→forAll(e|
    not(e.supervisedBy.oclIsUndefined())).
```

Example 17 Consider the security model `Empl.stm` (Example 4), but this time with `Empl2.dtm` as its underlying data model. Consider again the question that we asked ourselves in Example 15: namely, *is there any scenario in which nobody with role* `Supervisor` *is allowed to change the salary of someone else (including itself)*? According to Remark 1, the answer to this question is different from Example 15, namely, 'No', since the set of formulas Forms(Ex 15) is now unsatisfiable. Indeed, notice that in every valid scenario the invariants `allSuper` and `roleSuper` both evaluate to `true`, which means that, for each value for *self*, we can find a value for *caller* such that the expressions *self*.`supervisedBy =` *caller* and *caller*.`role = Supervisor` both evaluate to `true`.

Finally, let `Empl3.dtm` be the ComponentUML model that results from removing from `Empl2.dtm`, the invariant `roleSuper`.

Example 18 Consider the security model `Empl.stm` (Example 4), but this time with `Empl3.dtm` as its underlying data model. Consider, once again, the question that we asked ourselves in Example 15: namely, *is there any scenario in which nobody with role* `Supervisor` *is allowed to change the salary of someone else (including itself)*? According to Remark 1, the answer to this question is now different from Example 17, namely, 'Yes', since the set of formulas Forms(Ex 15) is now satisfiable. Indeed, consider a scenario with three employees e_1, e_2, and e_3, such that e_1 is linked with e_2 through the association-end `supervisedBy`, and similarly e_2 with e_3 and e_3 with e_1. Suppose also that e_2 and e_3 have role `Supervisor`, but e_1 has role `Worker`. (Notice that, since `roleSuper` is not included in `Empl3.dtm`, nothing prevents e_1 from not having the role `Supervisor`, despite the fact that it is linked with e_3 through the association-end `supervises`.) Clearly, for *self* $= e_3$, for every *caller* of role `Supervisor`, namely, e_2 and e_3, the expression *self*.`supervisedBy = `*caller* evaluates to `false`.

5 Automatically Reasoning about Fine-Grained Access Control Policies

Satisfiability modulo theories (SMT) solvers are tools for automatically proving the satisfiability of first-order formulas in a number of logical theories and their combination (Barrett et al. 2009). Basically, SMT generalizes boolean satisfiability (SAT) by incorporating equality reasoning, arithmetic, fixed-size bit-vectors, arrays, quantifiers, and other useful first-order theories. Of course, when dealing with quantifiers, SMT solvers cannot be complete, and may return *unknown* after a while, meaning that they can neither prove the quantified formula to be unsatisfiable, nor can they find an interpretation that makes it satisfiable. In the past years, there has been a great deal of interest and research on the foundational and practical aspects of SMT. They have also become the backbone of numerous applications in automated verification, artificial intelligence, program synthesis, security, product configuration, and much more.

We briefly report here on our experience using the Z3 SMT solver (de Moura & Bjørner 2008) to automatically obtain the answers to the questions posed in the examples in Section 4. Table 1 below summarizes the results of our experiments. For each example, we show the time it takes Z3 to return an answer (in all cases, less than 1 second); the answer that it returns (in all cases, the expected one); and the first-order model that it generates when the answer is `sat`, i.e., when it finds that the input set of formulas is satisfiable. Each model represents a scenario (not necessarily the one discussed in Section 4 for the corresponding example), and here we simply indicate the number of employees that it contains, which employees are linked through the association-end `supervisedBy`, which employees have the role `Worker`, which employees have the role `Supervisor`, which employee is the one requesting permission to change the salary (*caller*), and which employee is the one whose salary will be changed (*self*) if permission is granted. We ran our experiments on a laptop computer, with a 2.66GHz Intel Core 2 Duo processor and 4GB 1067 MHz memory, using Z3 version 4.3.2 9d221c037a95-x64-osx-10.9.2. Finally, the input for Z3 has been generated using our tool SecProver (SecProver 2014). This tool takes the following parameters: a data model, a security model, a set (possibly empty) of invariants, an action, a role, a set (possibly empty) of additional constraints, and a *question type*.[1] SecProver automatically generates the set of first-order formulas whose satisfiability will determine, according to our methodology, the answer to the given question.

Ex.	Time	Ans.	Interpretation
10	0.078s	`unsat`	—

11	0.107s	`sat`	#employees = 3
			supervisedBy = $\{(e_3, e_2), (e_1, e_2)\}$
			Worker = $\{e_1, e_3\}$
			Supervisor = $\{e_2\}$
			$caller = e_2, self = e_1$

12	0.041s	`unsat`	—
13	0.042s	`unsat`	—

14	0.306s	`sat`	#employees = 6
			supervisedBy = $\{(e_1, e_2), (e_2, e_3), (e_4, e_2), (e_5, e_3), (e_6, e_3)\}$
			Worker = $\{e_1, e_4, e_5, e_6\}$
			Supervisor = $\{e_2, e_3\}$
			$caller = e_3, self = e_1$

15	0.078s	`sat`	#employees = 1
			supervisedBy = $\emptyset$
			Worker = $\{e_1\}$
			Supervisor = $\emptyset$
			$self = e_1$

16	0.485s	`unsat`	—
17	0.060s	`unsat`	—

18	0.506s	`sat`	#employees = 15
			supervisedBy = $\{(e_1, e_2), (e_2, e_4), (e_3, e_4), (e_4, e_6), (e_5, e_4)(e_6, e_{12}), (e_7, e_4), (e_8, e_{14}), (e_9, e_4), (e_{10}, e_4), (e_{11}, e_{15}), (e_{12}, e_{13}), (e_{13}, e_4), (e_{14}, e_4), (e_{15}, e_4)\}$
			Worker = all
			Supervisor = $\emptyset$
			$self = e_2$

Table 1: Automatic reasoning over the examples 10-18 introduced in Section 4.

6 Related Work

Many proposals exist for reasoning about RBAC policies, each one using a different logic or formalism, including the so-called "default" logic (Woo & Lam 1993), modal logic (Massacci 1997), higher-order logic (Appel & Felten 1999), C-Datalog (Bacon et al. 2002), first-order logic (Jajodia et al. 2001, Bertino et al. 2003), and description logic (Zhao et al. 2005). To the best of our knowledge none of these proposals has been properly extended to cope with FGAC policies. In recent years, however, there has been a growing interest in finding appropriate formalisms and frameworks for specifying and analysing FGAC policies. In a nutshell, our proposal differs from other approaches in that: (i) we use Se-

[1] Currently, only four question types are supported, which correspond to the four blocks of examples considered in Section 4, but other question types will be added soon. The reason for using question types is to make it easier for those users who may not be familiar with first-order logic to understand the precise meaning of their questions, as well as the responses eventually given by the SMT solver to these questions.

cureUML+ComponentUML (Basin et al. 2006) for modeling FGAC policies, and (ii) we use a mapping from OCL to first-order (Clavel et al. 2009, Dania & Clavel 2013) for reasoning about these policies. In our opinion, our approach has two main advantages: (i) the reasoning about FGAC policies can take into account the properties of the system states, since OCL is the language that we use both for specifying the invariants in the data model and the authorization constraints in the security model; and (ii) the reasoning about FGAC policies can be done automatically (although sometimes may fail to find a result), since the mapping that we use for translating OCL into first-order logic supports the effective application of SMT solvers over the generated formulas.

Halpern & Weissman (2008) have proposed an interesting framework for specifying and reasoning about FGAC policies, called Lithium. It is based on a decidable fragment of (multi-sorted) first-order logic. Differently from OCL, this logic does not consider undefined values, which, based on our experience, is something crucial when formalizing properties of the system states. Unfortunately, we are not aware of case studies that have been carried out using Lithium, and which we could use to compare it with our approach in terms of the expressiveness of the underlying formalisms and of the effectiveness of the associated reasoning tools.

Kuhlmann et al. (2011, 2013) propose a domain-specific language for specifying role-based policies which is based on UML and OCL. For the purpose of analyzing these policies, they propose to use SAT solvers, and, in particular the one implemented in Alloy (Jackson 2002). Differently from SMT solvers, Alloy requires the search space to be bounded, by explicitly indicating the number of objects in each entity, the number of links of each association and the possible values of each attribute. Also, integer expressions are not allowed, neither in the invariants nor in the policies under consideration. On the other hand, this approach enables one to include, within the policies, some time-constraints, which are not possible in our approach.

Finally, in the context of XACML (OASIS 2013), there exists a XACML profile for the specification of RBAC policies (OASIS 2010). However, no methods have been proposed for reasoning about policies written with this profile. Also, it is unclear whether this profile can be extended to cope with fine-grained access control policies. To address the first concern, Helil & Rahman (2010) propose an extension of the XACML profile for RBAC based on OWL. This approach supports the use of an OWL-DL reasoner for deciding about RBAC policies within XACML. More interestingly, Ramli et al. (2014) have recently proposed a new syntax and semantics for XACML, for the purpose of supporting formal reasoning about XACML policies. One of the challenges here is to formalize the different algorithms for enforcing policy rules which are available in XACML. Ramli et al. (2014) formalize the majority of these algorithms, and propose two new algorithms (one of which is very close to the semantics of SecureUML+ComponentUML.) Another challenge is to formalize the concepts of obligations and advices in XACML, but they are not covered by Ramli et al. (2014). Finally, with respect to methods for reasoning about XACML policies, Ramli et al. (2014) propose to explore the use of SMT solvers, but no experiments are reported yet.

7 Conclusions and Future Work

Model-driven engineering supports the development of complex software systems by generating software from models. Model-driven security (Basin et al. 2011) is a specialization of this paradigm, where system designs are modeled together with their security requirements and security infrastructures are directly generated from the models. Of course, the quality of the generated code depends on the quality of the source models. If the models do not properly specify the system's intended behavior, one should not expect the generated system to do so either. Experience shows that even when using powerful, high-level modeling languages, it is easy to make logical errors and omissions. It is critical not only that the modeling language has a well-defined semantics, but also that there is tool support for analyzing the modeled systems' properties.

In this paper we have presented a novel, tool-supported methodology for reasoning about fine-grained access control policies (FGAC). We have also briefly reported on our experience using the Z3 SMT solver (de Moura & Bjørner 2008) for automatically proving non-trivial properties about FGAC policies. Within our methodology, we use SecureUML (Basin et al. 2006) to specify FGAC policies. SecureUML is a modeling language that extends role-based access control (RBAC) (Ferraiolo et al. 2001) with authorization constraints, which are formalized using the Object Constraint Language (OCL) (Object Management Group 2014).

The key component of our methodology is a mapping from OCL to first-order logic (Clavel et al. 2009, Dania & Clavel 2013), which allows one to transform questions about FGAC policies into satisfiability problems in first-order logic. Although this mapping does not cover the complete OCL language, our experience shows that the kind of OCL expressions typically used for specifying invariants and authorization constraints are covered by our mapping. More intriguing is, however, the issue about the effectiveness of SMT solvers for automatically reasoning about FGAC policies. Although our experience so far is extremely encouraging (all problems are solved in less than a second), we should not forget that our results completely depend on the interaction between (i) the way our mapping translates into first-order logic the relevant OCL expressions (invariants and authorization constraints) and (ii) the heuristics implemented in the SMT solver. We are currently analyzing this interaction in depth to better understand its scope and limitations. Ultimately, we know that there is a trade-off when using SMT solvers. On the one hand, they are necessarily incomplete and their results depend on heuristics, which may change. In fact, we have experienced (more than once) that two different versions of Z3 may return 'sat' and 'unknown' for the very same problem. This is not surprising (since two versions of the same SMT solver may implement two different heuristics) but it is certainly disconcerting. On the other hand, SMT solvers are capable of checking, in a fully automatic and very efficient way, the satisfiability of large sets of complex formulas. In fact, we have examples, involving more than a hundred non-trivial OCL expressions, which are checked by Z3 in just a few seconds.

Finally, as part of our future work, we plan to define formal mappings between the FGAC languages and frameworks supported by commercial DBMS (e.g., Oracle, IBM/DB2, Microsoft SQL Server and Teradata) and SecureUML. These mappings will allow us to apply our methodology also when reasoning about FGAC policies in commercial DBMS.

References

ActionGUI (2012). `http://actiongui.org/`, see ActionGUI project.

Appel, A. W. & Felten, E. W. (1999), Proof-carrying authentication, *in* J. Motiwalla & G. Tsudik, eds, 'ACM Conference on Computer and Communications Security', ACM, pp. 52–62.

Bacon, J., Moody, K. & Yao, W. (2002), 'A model of OASIS role-based access control and its support for active security', *ACM Trans. Inf. Syst. Secur.* **5**(4), 492–540.

Barrett, C. W., Sebastiani, R., Seshia, S. A. & Tinelli, C. (2009), 'Satisfiability modulo theories.', *Handbook of satisfiability* **185**, 825–885.

Basin, D. A., Clavel, M. & Egea, M. (2011), A decade of model-driven security, *in* 'SACMAT 2011', Vol. 1998443, New York, NY, USA, Innsbruck, Austria, pp. 1–10.

Basin, D. A., Clavel, M., Egea, M., de Dios, M. A. G. & Dania, C. (2014), 'A model-driven methodology for developing secure data-management applications', *IEEE Trans. on Software Engineering* **40**(4), 324–337.

Basin, D., Doser, J. & Lodderstedt, T. (2006), 'Model driven security: from UML models to access control infrastructures.', *ACM Trans. on Software Engineering and Methodology* **15**(1), 39–91.

Bertino, E., Catania, B., Ferrari, E. & Perlasca, P. (2003), 'A logical framework for reasoning about access control models', *ACM Trans. Inf. Syst. Secur.* **6**(1), 71–127.

Clavel, M., Egea, M. & de Dios, M. A. G. (2009), 'Checking unsatisfiability for OCL constraints', *Electronic Communications of the EASST* **24**, 1–13.

Dania, C. & Clavel, M. (2013), OCL2FOL+: coping with undefinedness, *in* J. Cabot, M. Gogolla, I. Ráth & E. D. Willink, eds, 'OCL@MoDELS', Vol. 1092 of *CEUR Workshop Proceedings*, pp. 53–62.

de Moura, L. M. & Bjørner, N. (2008), Z3: an efficient SMT solver, *in* C. R. Ramakrishnan & J. Rehof, eds, 'Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS 2008, Proceedings', Vol. 4963 of *LNCS*, Springer, pp. 337–340.

Egea, M., Dania, C. & Clavel, M. (2010), 'MySQL4OCL: a stored procedure-based MySQL code generator for OCL', *Electronic Communications of the EASST* **36**.

Ferraiolo, D. F., Sandhu, R. S., Gavrila, S., Kuhn, D. R. & Chandramouli, R. (2001), 'Proposed NIST standard for role-based access control', *ACM Trans. Inf. Syst. Sec.* **4**(3), 224–274.

Halpern, J. Y. & Weissman, V. (2008), 'Using first-order logic to reason about policies', *ACM Trans. Inf. Syst. Secur.* **11**(4).

Helil, N. & Rahman, K. (2010), 'Extending XACML profile for RBAC with semantic concepts'.

Huey, P. (2014), 'Oracle database security guide', `http://docs.oracle.com/database/121/`.

IBM (2013), 'IBM DB2. Database security guide', `http://www-01.ibm.com/support/docview.wss?uid=swg27038855`.

Jackson, D. (2002), Alloy: a new technology for software modelling, *in* J. Katoen & P. Stevens, eds, 'Tools and Algorithms for the Construction and Analysis of Systems, 8th International Conference, TACAS 2002, Proceedings', Vol. 2280 of *LNCS*, Springer, p. 20.

Jajodia, S., Samarati, P., Sapino, M. L. & Subrahmanian, V. S. (2001), 'Flexible support for multiple access control policies', *ACM Trans. Database Syst.* **26**(2), 214–260.

Kuhlmann, M., Sohr, K. & Gogolla, M. (2011), Comprehensive two-level analysis of static and dynamic RBAC constraints with UML and OCL, *in* 'Fifth International Conference on Secure Software Integration and Reliability Improvement, SSIRI 2011', IEEE, pp. 108–117.

Kuhlmann, M., Sohr, K. & Gogolla, M. (2013), 'Employing UML and OCL for designing and analysing role-based access control', *Mathematical Structures in Computer Science* **23**(4), 796–833.

Massacci, F. (1997), Reasoning about security: a logic and a decision method for role-based access control, *in* D. M. Gabbay, R. Kruse, A. Nonnengart & H. J. Ohlbach, eds, 'Qualitative and Quantitative Practical Reasoning, First International Joint Conference on Qualitative and Quantitative Practical Reasoning ECSQARU-FAPR'97, Proceedings', Vol. 1244 of *LNCS*, Springer, pp. 421–435.

OASIS (2010), 'XACML core and hierarchical role-based access control', `http://docs.oasis-open.org/xacml/3.0/`.

OASIS (2013), 'Extensible access control markup language (XACML)', `http://docs.oasis-open.org/xacml/3.0/`.

Object Management Group (2014), Object Constraint Language specification, Technical report, OMG. `http://www.omg.org/spec/OCL/2.4`.

Ramli, C. D. P. K., Nielson, H. R. & Nielson, F. (2014), 'The logic of XACML', *Sci. Comput. Program.* **83**, 80–105.

SecProver (2014). `http://actiongui.org/`, see SecProver project.

SQL (2012), 'Microsoft SQL Server 2012. Implementing row- and cell-level security in classified databases', `http://msdn.microsoft.com/en-us/library/bb545450.aspx`.

Teradata (2014), 'Teradata database. Security administration', `http://www.info.teradata.com/`.

Woo, T. Y. C. & Lam, S. S. (1993), 'Authorizations in distributed systems: A new approach', *Journal of Computer Security* **2**(2-3), 107–136.

Zhao, C., Heilili, N., Liu, S. & Lin, Z. (2005), Representation and reasoning on RBAC: a description logic approach, *in* D. V. Hung & M. Wirsing, eds, 'Theoretical Aspects of Computing - ICTAC 2005, Second International Colloquium, Proceedings', Vol. 3722 of *LNCS*, Springer, pp. 381–393.