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 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 logic.

Copyright ©2015, Australian Computer Society, Inc. This paper appeared at the 11th Asia-Pacific Conference on Conceptual Modelling (APCCM 2015), Sydney, Australia, January 2015. Conferences in Research and Practice in Information Technology (CRPIT), Vol. 165, Henning Köhler and Motoshi Saeki, Ed. Reproduction for academic, not-for-profit purposes permitted provided this text is included.
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:

<table>
<thead>
<tr>
<th>Resource</th>
<th>Actions</th>
</tr>
</thead>
<tbody>
<tr>
<td>Entity</td>
<td>create, delete</td>
</tr>
<tr>
<td>Attribute</td>
<td>read, update</td>
</tr>
<tr>
<td>Association-end</td>
<td>read, create, delete</td>
</tr>
</tbody>
</table>

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.](image)

(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, e1, e2 and e3, and e1 is supervised by e2, e2 is supervised by e3, and e3 has no supervisor at all. Moreover, e1 has role Worker and both e2 and e3 have role Supervisor. Instance 2b has also three employees, e1, e2 and e3, but this time e1 is supervised by e2 and e2 is supervised by itself, and e3 has no supervisor at all. As before, e1 has role Worker and both e2 and e3 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→isEmpty, →includes, and →excluding, as well as operators to iterate over collections, such as →forAll, →exists, and →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.

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 desugared 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 desugaring a SecureUML+ComponentUML model $S$:
- Role hierarchies. Let act be an action and let $r$ and $r'$ be two roles. Suppose that $r$ is a subrole of $r'$ in $S$, and that there is a permission in $S$ for $r'$ to execute act under the constraint auth. Then, when desugaring $S$, we add a new permission to $S$ 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 $S$ for a role $r$ to execute act under the constraint auth. Then, when
de-sugaring $S$, for every association-end $assoc$ owned by $entity$, we add to $S$ a new permission for $r$ to execute $\text{delete}(assoc)$ under the same constraint $auth$.

- **Opposite association-ends.** Let $assoc$ and $assoc'$ be two opposite association-ends. Let $act$ be the action $\text{create}(assoc)$. Suppose that there is a permission in $S$ for a role $r$ to execute $act$ under the constraint $auth$. Then, when de-sugaring $S$, we add to $S$ a new permission for the role $r$ to execute $\text{create}(assoc')$ 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 $\text{delete}(assoc)$.

- **Denying by default.** Let $r$ be a role and let $act$ be an action. Suppose that there is no permission in $S$ for the role $r$ to execute $act$. When de-sugaring $S$, we add to $S$ 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 $S$ for the role $r$ to execute $act$. When de-sugaring $S$, 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 $\text{Auth}(S, r, act)$ the authorization constraint assigned, in the de-sugared presentation of the ComponentUML+SecureUML model $S$, to the role $r$’s permission for performing the action $act$.

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

$\text{Auth}(\text{Empl.stm, Worker, update(salary)}) = false$, by the rule “denying by default”.

$\text{Auth}(\text{Empl.stm, Supervisor, update(salary)}) = self.supervisedBy = \text{caller or false}$, by the combination of the rules “denying by default”, “role hierarchies”, and “disjunction of authorization constraints”.

$\text{Auth}(\text{Empl.stm, Worker, read(salary)}) = \text{caller = self}$.

$\text{Auth}(\text{Empl.stm, Supervisor, read(salary)}) = \text{caller = self or true}$, 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 (Huynh 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 ocl2fol$_{def}$; 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 $D$ be a ComponentUML model, with invariants $\text{expr}_1, \ldots, \text{expr}_n$, and let $\text{expr}$ be a boolean expression. Then, $\text{expr}$ evaluates to $true$ in every valid instance of $D$ if and only if

$$\bigcup_{i=1}^{n} \text{ocl2fol}_{def}(\text{expr}_i) \cup \{\text{oocl2fol}(\text{expr})\}$$

is unsatisfiable.

Next, we will explain, and illustrate with examples, the main ideas behind the maps ocl2fol$_{def}$ 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 ocl2fol_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 $D$ be a ComponentUML model. $ocl2fol_def(D)$ returns the axioms formalizing the properties of the predicates and functions that represent the entities, attributes and association-ends in $D$, 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, $ocl2fol_def(EmplBasic.dtm)$ returns the axiom

$\forall(x)(\text{Employee}(x) \Rightarrow \neg(\text{isNull}(x) \lor \text{isInvalid}(x)))$, 

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

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

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

The following remark formalizes the main property of the map $ocl2fol_def$.

Remark 2 Let $D$ be a ComponentUML model. Then, there is a one-to-one correspondence between the instances of $D$ and the first-order models that satisfy $ocl2fol_def(D)$.

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 $ocl2fol_def$ (see below).

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

$\exists(e)(\text{Employee}(e) \land \text{isNull}(\text{supervisedBy}(e))$ 

$\land \forall(e')(\text{Employee}(e') \land \text{isNull}(\text{supervisedBy}(e'))$ 

$\Rightarrow e' = e)$,

and $ocl2fol(noSelfSuper)$ returns the formula

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

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

Remark 3 Let $D$ 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 $D$ in which the $expr$ evaluates to true and the first-order models that satisfy $ocl2fol_def(D) \cup (ocl2fol(expr))$.

3.3 The map ocl2fol_def (expressions)
Often, OCL expressions include subexpressions that will evaluate to collections: e.g., those which are built using $\rightarrow select$, $\rightarrow collect$, or $\rightarrow excluding$. 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 $ocl2fol_def$, are given by the map $ocl2fol_def$.

Example 8 Consider the expression

Employee.allInstances() $\rightarrow select(e|e.supervises$ $\rightarrow notEmpty()$).

This expression, which we refer to as $colOfSuper$, will evaluate to a set containing only those employees whose list of supervises is not empty. Then, $ocl2fol_def(colOfSuper)$ returns the following axiom,

$\forall(x)(P_{colOfSuper}(x) \iff (\text{Employee}(x) \land \exists(y)(\text{supervises}(x, y))))$, 

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 $ocl2fol_def$ over expressions of type set.

Remark 4 Let $D$ 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 $D$ and the first-order models that satisfy $ocl2fol_def(D) \cup ocl2fol_def(expr)$ for which the following holds: $expr$ evaluates to $\{o_1, \ldots, o_n\}$ in $D$ 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 $D$ contains invariants $expr_1, \ldots, expr_n$, we will consider that $ocl2fol_def(D)$ includes also the formulas $\bigcup_{i=1}^n ocl2fol_def(expr_i)$.

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 supervises? Let us formalize the property that no employee is supervised by their own supervises as follows:

Employee.allInstances() $\rightarrow forAll(e|e.supervises$ $\rightarrow excludes(e.supervisedBy))$. 

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

\[
\text{ocl2fol}(\text{Empl1.dtm}) \cup \neg(\text{ocl2fol}(\text{noMixSuper}))
\]

is satisfiable. Indeed, consider, for example, an instance of \text{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 \text{supervisedBy} with \(e_4\), and similarly \(e_2\) with \(e_3\), and \(e_2\) with \(e_3\). Suppose also that \(e_1\) is of role \text{Worker}, and \(e_2,\ e_3,\) and \(e_4\) are of role \text{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 \text{supervisedBy} with \(e_3\) (since \(e_3\) is linked through \text{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 \textit{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 \textit{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 \text{Empl1.dtm} (Example 4). Recall that this model's underlying data model is the ComponentUML model \text{Empl1.dtm} (Example 3), which includes the invariants \textit{oneBoss, noSelfSuper, roleSuper,} and \textit{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:

\[
\text{ocl2fol}(D) \cup \{\exists(caller)\exists(self)\exists(target)\exists(value) (\text{ocl2fol}(caller.role = r) \land \text{ocl2fol}([\text{Auth}(\mathcal{S}, r, act)])))
\]

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

\[
\text{Auth}(\text{Empl1.stm, Worker, update(salary)}) = \text{false}.
\]

According to Remark 1, the answer to this question is ‘Yes’, since the following set of formulas is clearly unsatisfiable:

\[
\text{ocl2fol}(\text{Empl1.dtm}) \cup \{\exists(caller)\exists(self) (\text{ocl2fol}(\text{caller.role = Worker}) \\
\land \text{ocl2fol}(\text{false}))
\]

(Note that ocl2fol(\text{false}) returns \text{false}.) Indeed, there is no scenario in which the expression false can evaluate to true.

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

\[
\text{Auth}(\text{Empl1.stm, Supervisor, update(salary)}) = \\
(\text{self.supervisedBy = caller or false}).
\]

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

\[
\text{ocl2fol}(\text{Empl1.dtm}) \cup \{\exists(caller)\exists(self) (\text{ocl2fol}(\text{caller.role = Supervisor}) \\
\land \text{ocl2fol}(\text{self.supervisedBy = caller or false}))
\]

(Note that ocl2fol(\text{self.supervisedBy = caller}) returns \text{false}.) Clearly, for \text{caller = e2} and self = \text{e1}, the expression self.supervisedBy = caller evaluates to true in this scenario.

Example 12 Consider the following question: \textit{Is there any scenario in which someone with role \text{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\text{(i.e., the employee whose salary is to be updated)} and caller\text{(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:

\[
\text{ocl2fol}(\text{Empl1.dtm}) \cup \{\exists(caller)\exists(self) (\text{ocl2fol}(\text{caller.role = Supervisor}) \\
\land \text{ocl2fol}(\text{self.supervisedBy = caller or false}))
\]

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: \textit{Is there any scenario in which someone with role \text{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:

\[
\text{ocl2fol}(\text{Empl1.dtm}) \cup \{\exists(caller)\exists(self) (\text{ocl2fol}(\text{caller.role = Supervisor}) \\
\land \text{ocl2fol}(\text{self.supervisedBy = null or self.supervisedBy = caller or false}))
\]
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:

\[
\text{ocl2fol}(D) \cup \{ \exists(caller) \exists(self) \exists(target) \exists(value) \exists(\text{Auth}(S,r, act)) \}.
\]

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:

\[
\text{ocl2fol}(\text{Empl1.dtm}) \cup \{ \exists(caller) \exists(self) \exists(target) \exists(value) \exists(\text{Auth}(S,r, act)) \}.
\]

Consider, for example, a scenario with just three employees, e₁, e₂, and e₃ such that e₁ is linked with e₂ through the association-end supervisedBy, and similarly e₂ with e₃; but e₁ is not linked with e₃ through the association-end supervisedBy. Suppose that e₂ and e₃ have role Supervisor and e₁ has role Worker. Clearly, for caller = e₃ and self = e₁, 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:

\[
\text{ocl2fol}(D) \cup \{ \exists(caller) \exists(target) \exists(value) \exists(\text{Auth}(S,r, act)) \}.
\]

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 is satisfiable:

\[
\text{ocl2fol}(\text{Empl1.dtm}) \cup \{ \exists(caller) \exists(target) \exists(value) \exists(\text{Auth}(S,r, act)) \}.
\]

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 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:

\[
\text{ocl2fol}(D) \cup \{ \forall(caller) \exists(target) \exists(value) \exists(\text{Auth}(S,r, act)) \}.
\]

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:

\[
\text{ocl2fol}(\text{Empl1.dtm}) \cup \{ \forall(caller) \exists(target) \exists(value) \exists(\text{Auth}(S,r, act)) \}.
\]

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:

\[
\text{Employee.allInstances() \rightarrowForAll\{e | \not(e.supervisedBy.ooclIsUndefined())\}}.
\]

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 $\text{Forms(Ex 15)}$ 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 $\text{supervisedBy}$, and similarly $e_2$ with $e_3$ and $e_3$ with $e_1$. Suppose also that $e_2$ and $e_3$ have role $\text{Supervisor}$, but $e_1$ has role $\text{Worker}$. (Notice that, since $\text{roleSuper}$ is not included in $\text{Empl3.dtm}$, nothing prevents $e_1$ from not having the role $\text{Supervisor}$, despite the fact that it is linked with $e_3$ through the association-end $\text{supervises}$.) Clearly, for self $= e_2$, for every caller of role $\text{Supervisor}$, namely, $e_2$ and $e_3$, the expression $\text{self.supervisedBy} = \text{caller}$ 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 $\text{supervisedBy}$, which employees have the role $\text{Worker}$, which employees have the role $\text{Supervisor}$, 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

<table>
<thead>
<tr>
<th>Ex.</th>
<th>Time</th>
<th>Ans.</th>
<th>Interpretation</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>0.078s</td>
<td>unsat</td>
<td>—–</td>
</tr>
<tr>
<td>11</td>
<td>0.107s</td>
<td>sat</td>
<td>$#\text{employees} = 3$ \text{supervisedBy} = {(e_1, e_2), (e_1, e_3)}</td>
</tr>
<tr>
<td>12</td>
<td>0.041s</td>
<td>unsat</td>
<td>—–</td>
</tr>
<tr>
<td>13</td>
<td>0.042s</td>
<td>unsat</td>
<td>—–</td>
</tr>
<tr>
<td>14</td>
<td>0.306s</td>
<td>sat</td>
<td>$#\text{employees} = 6$ \text{supervisedBy} = {(e_1, e_2), (e_2, e_3), (e_4, e_2), (e_5, e_3), (e_6, e_3)} \text{Worker} = {e_1, e_4, e_5, e_6} \text{Supervisor} = {e_2, e_3} \text{caller} = e_3, \text{self} = e_1</td>
</tr>
<tr>
<td>15</td>
<td>0.078s</td>
<td>sat</td>
<td>$#\text{employees} = 1$ \text{supervisedBy} = {} \text{Worker} = {e_1} \text{Supervisor} = {} \text{self} = e_1</td>
</tr>
<tr>
<td>16</td>
<td>0.485s</td>
<td>unsat</td>
<td>—–</td>
</tr>
<tr>
<td>17</td>
<td>0.060s</td>
<td>unsat</td>
<td>—–</td>
</tr>
<tr>
<td>18</td>
<td>0.506s</td>
<td>sat</td>
<td>$#\text{employees} = 15$ \text{supervisedBy} = {(e_1, e_2), (e_2, e_3), (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)} \text{Worker} = \text{all} \text{Supervisor} = {} \text{self} = e_2</td>
</tr>
</tbody>
</table>

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

Many proposals exist for reasoning about RBAC policies. In recent years, however, 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.

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 SMT solver to these questions.

1Currently, 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
cies, Ramli et al. (2014) propose to explore the use of spectral to methods for reasoning about XACML policies and advices in XACML, but they are not propose two new algorithms (one of which is very close to) formalize the majority of these algorithms, 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, Heli & Rahman (2010) propose an extension of Alloy which is formally indicated 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.

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 & Björner 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 investigating 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 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. 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 investigating 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 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. 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 investigating 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 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.
References


