A Generic Metamodel For Security Policies Mutation

Tejeddine Mouelhi, Franck Fleurey, Benoit Baudry

A Generic Metamodel For Security Policies Mutation

Tejeddine Mouelhi
IT-Telecom Bretagne
35576 Cesson Sévigné Cedex, France
tejeddine.mouelhi@telecom-bretagne.eu

Franck Fleurey
SINTEF
P.O. Box 124 Blindern
N-0314 Oslo, Norway
franck.fleurey@sintef.no

Benoit Baudry
IRISA- 35042 Rennes
Cedex, France
bbaudry@irisa.fr

Abstract. *We present a new approach for mutation analysis of Security Policies test cases. We propose a metamodel that provides a generic representation of security policies access control models and define a set of mutation operators at this generic level. We use Kermeta to build the metamodel and implement the mutation operators. We also illustrate our approach with two successful instantiation of this metamodel: we defined policies with RBAC and OrBAC and mutated these policies.*

1 Introduction

Access control policies are a among the most important security mechanisms necessary to increase the confidence in a system. Verifying that the implementation does not contain flaws or security breaches is thus a critical task. Testing security policies is a possible approach to fulfill this objective that requires generating efficient test cases. One strategy to evaluate the efficiency of these test cases is to perform mutation analysis [1] which has proved its effectiveness in many fields in the past.

The main idea behind mutation analysis is that a good set of test cases should be able to detect common faults that can occur in a program. When validating the efficiency of a set of test cases for a particular system under test, the analysis consists in injecting errors in the program to create mutant versions. All the test cases are then executed again each mutant and a mutation score is computed as the rate of mutants that are detected by one test case at least. An important assumption for this analysis is that the errors that are injected are relevant of most types of faults can occur. The different faults are modeled as mutation operators and systematically injected in the whole program when performing the experiment.

In recent works, mutation analysis has been applied to security policies testing [2, 3]. The main idea is to inject flaws into the security policy to get a set of mutant policies. Then, the efficiency of the security tests is evaluated by the rate of mutants that can reveal the injected flaws by distinguishing between the initial policy and mutant.

In this paper, we propose a generic metamodel for security policies formalisms. It captures the necessary concepts to express various rule-based security policy formalisms (*e.g.* R-BAC [4], MAC [5, 6] DAC [7] OrBAC [8]). This metamodel thus allows a modeler to design a new formalism and to model policies according to this formalism. Based on this generic definition of a security policy formalism, we express mutation operators that can apply to all rule-based formalisms. This generic definition of mutation operators will allow us to study mutation analysis for various security policy formalisms without implementing the same mutation operators as many times as there formalisms.

To validate this approach, we have implemented the generic security metamodel and the mutation operators within the Kermeta environment. We have then successfully instantiated the metamodel to define the OrBAC and RBAC security formalisms and to model a security policy for a library system using these two formalisms. We have also been able to automatically mutate both policies and produce a set of faulty policies. It is interesting to notice that the same mutation operator, depending on the formalism (RBAC, OrBAC etc.), produces very different mutants in terms of the simulated flaws.

The rest of the paper is organized as follows. In the next section, the existing approaches applying mutation testing to security policies are presented with a focus on the mutation operators. Section 3 presents the background and the generic metamodel. In Section 4, we show some examples highlighting how the mutation operators were implemented.

2 Background

In this section, we first discuss the previous approaches that applied mutation analysis for security policies. Then, we introduce metamodelling and the Kermeta tool used to build our generic metamodel.

2008 IEEE International Conference on Software Testing Verification and Validation Workshop (ICSTW'08)
978-0-7695-3388-9/08 $25.00 © 2008 IEEE

2.1 Mutation for security policies

Two different works have applied mutation analysis to security policies. We start by presenting the work of Xie et al. [2] who applied mutation to XACML. Then we summarize our previous work [3, 9] where we applied mutation analysis to OrBAC policies.

a Mutation applied to XACML

Xie et al. [2] applied mutation to XACML testing. XACML is an Oasis standard XML syntax for defining security policies. A framework is available that facilitates PDP (policy decision point) implementation. One of the difficulties of XACML is its complexity. They used mutation to evaluate different structural coverage criteria for XMACML policies tests generation and selection which they proposed in a previous work.

They proposed several mutation operators. The majority of these operators is platform dependant and is related to the way XACML expresses policies and rules. Here are some examples of operators:

- RTT: Policy Set Target True: Removes the target tag. The rule will be applied to all requests.
- CPC: Changes the combining algorithms (these algorithms allow to decide what rules/policies are applied).
- CRE: Changes the rule type (Deny becomes Allow and Allow becomes Deny).

These operators are efficient for revealing test cases weaknesses. In fact, a subset of operators (like RTT) emulates all possible syntactic faults w.r.t XACML syntax. In addition, other operators (CPC and CRE for example) emulate semantic faults.

The CRE which replaces permissions with prohibitions can be reused as a generic operator for security policies that have rule status (deny or accept).

In our approach, we tried to find out the operators that can be reused and included in the metamodel for security mutation operators. The CRE operator is one of these operators.

b Mutation applied to OrBAC

In a previous work [3, 9], we applied mutation analysis in order to qualify test cases for OrBAC (Organization Based Access Control) models.

An OrBAC security rule can be a permission, prohibition or obligation. A rule has 5 parameters (called entities): an organization, a role, an activity, a view and a context. To increase modularity for the definition of security rules, OrBAC enables the definition of hierarchies for entities. In that case, rules defined on high level entities are inherited by the sub-entities. An advantage of OrBAC is that is has a tool called MotOrBAC [10] that allows to define, administer policies and check conflicts.

We proposed a set of mutation operators that are adapted to OrBAC. Here are some examples:

- PPR: replaces permission with prohibition.
- CRD: replaces a rule context with a different one.
- APD: replaces a rule activity with one of its descendants.
- ANR: adds a new rule.

As highlighted in the previous section, some operators are related to the OrBAC model. For example APD and CRD cannot be applied to RBAC policies. Nevertheless, we can reuse some of the proposed operators and make them generic. For example, The ANR operator is an excellent candidate.

2.2 Metamodelling and Kermeta

This section summarizes the intents of metamodelling and how the Kermeta environment fits in this modelling activity.

a Metamodelling

Metamodelling [12, 13] consists in building a metamodel that defines a modeling language for a particular domain. The metamodel defines the concepts and relationships that describe the domain. A metamodel is a model itself that is expressed with a modeling language called the meta-metamodel. In the MDA context, the OMG has defined the MOF meta-metamodel [14] to define the basic structure of the metamodel. The OCL [15] can also be used to add constraints about the static semantics of the metamodel. These constraints define structural well-formedness rules that must be satisfied by any model that instantiates the metamodel. However, MOF and OCL are not designed for specifying the dynamic semantics of the language.

Concerning the dynamic semantics description of a metamodel, there is no standard language today. The Kermeta environment has been designed towards this purpose: it is an extension of MOF that allows defining operations in metamodels which instances are executable. Thus, using this metamodelling environment, it is possible to define metamodels which completely define a language for a particular domain. We have used this approach for implementing a requirement modeling language (presented in Section 4).

b Kermeta

Kermeta [11] is an open source metamodelling environment developed by the Triskell team at IRISA that is fully integrated with Eclipse. It has been designed as an extension to the meta-data language

2008 IEEE International Conference on Software Testing Verification and Validation Workshop (ICSTW'08)
978-0-7695-3388-9/08 $25.00 © 2008 IEEE

EMOF [14] with an action language that allows specifying semantics and behavior of metamodels. The action language is imperative and object-oriented and is used to provide an implementation of operations defined in metamodels. A more detailed description of the language is presented in [16].

The Kermeta action language has been specially designed to process models. It includes both OO features and model specific features. Convenient constructions of the Object Constraint Language (OCL) such as closures (e.g. each, collect, select) are also available in Kermeta. The action language offered by Kermeta is well adapted to model-oriented activities such as:

- Specification of abstract syntax, static semantic (OCL) and dynamic semantics.
- model and metamodel simulation and prototyping
- model transformation
- aspect weaving

3 A generic framework for security policies

In the literature, several security formalisms such as RBAC or OrBAC are based on the definition of security rules. Depending on the formalism, the type of rules and the entity they manipulate are different. The idea of the framework proposed in this section is to support the definition of security policies from these various formalisms in order to allow for the implementation of generic tools to manipulate the security models. In order to support this idea, the framework must thus allow defining all the different formalisms, and expressing models using those formalisms.

The framework is built around a metamodel which allows representing both particular rule-based security formalism (such as RBAC or OrBAC) and instances of this formalism that model actual security policies. The metamodel was defined using the Eclipse Modelling Framework and implemented within the Kermeta environment.

3.1 Generic security metamodel

Figure 1 presents the generic security metamodel. On the diagram all classes include a name attribute. In the implementation this attribute is factorized in a common super-class *NamedElement*.

The metamodel is divided in two parts: the three top classes (*PolicyType*, *ElementType* and *RuleType*) are the general concepts that allow defining any rule-based security policy formalism. The class PolicyType is the root class for the definition of security formalism. Security formalism consists of a set of element types (*ElementType*) and a set of rule types (*RuleType*). Each rule type has a set of parameters which are typed by element types. In the following we present how these three classes can be instantiated to represent RBAC and OrBAC formalisms.

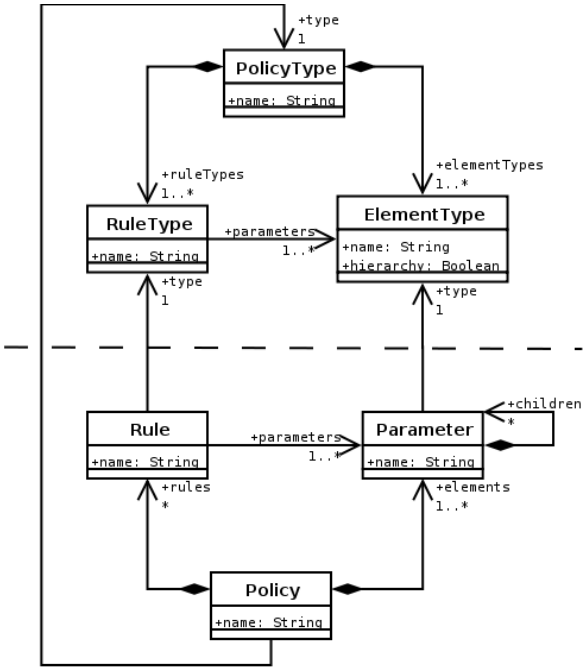

Figure 1 – The meta-model for rule-based security formalisms

The three bottom classes (*Policy*, *Rule* and *Element*) on the diagram in Figure 1 allow defining actual security policies using a formalism defined with the three top classes. The class Policy is the root class to instantiate in order to create a security policy. Each policy must have a type (which is an instance of class *PolicyType* discussed in the previous paragraph) and contains elements and rules. The type of a policy constrains the types of elements and rules it can contain. Each element has a type which must belong to the element types of the policy type. If the *hierarchy* property of the element type is true, then the element can contain children of the same type as itself. This is used for example to define hierarchies of roles in OrBAC. Finally, rules can be defined by instantiating the Rule class. Each rule has a type which should again belong to the policy type. Each rule has a set of parameters which types should match the types of the parameters of the type of the rule.

In practice the metamodel was defined as an Ecore model in the EMF framework. The advantages of using EMF are:

2008 IEEE International Conference on Software Testing Verification and Validation Workshop (ICSTW'08)
978-0-7695-3388-9/08 $25.00 © 2008 IEEE

- It generates editors that can be directly used to define security types and policies.
- It integrates smoothly in the Kermeta environment which is used to implement the mutation tool.

3.2 Definition of types of security policies

Figure 2 shows how the metamodel was instantiated to model OrBAC security policies. This figure is a snapshot of the model editor generated by EMF from the metamodel presented in the previous section. In OrBAC there are five types of entities: organizations, roles, activities, views and contexts. All these types were defined by instances of the ElementType class. Among these types of element, only Roles can be organized hierarchically.

OrBAC defines three types of rules: permission, prohibition and obligation. All three types of rule have the same five parameters: an organisation, a role, an activity, a view and a context.

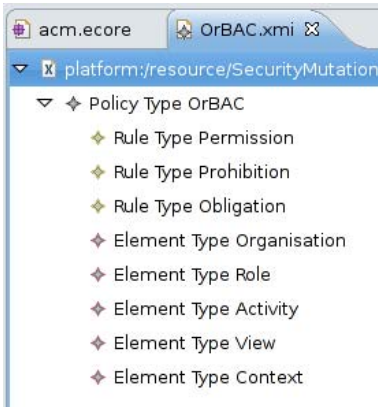

Figure 2 – OrBAC model

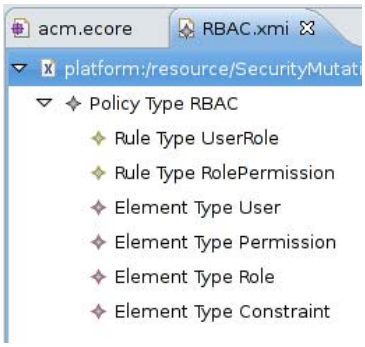

Figure 3 – RBAC model

Figure 3 presents the RBAC model. In the same way as for the OrBAC model, the PolicyType class is instantiated to model RBAC. RBAC defines four types of entities: users, permissions, roles and constraints. RBAC associates users with roles on one hand and roles with permissions on the other hand. Two types of rules have to be defined:

- UserRole rules which have two parameters: a user and a role.
- RolePermission rules which have three parameters: a role, a permission and a constraint.

The examples of OrBAC and RBAC show how these two existing rule based security mechanisms can be modelled within the proposed framework. The next subsection shows an example of how actual policies can be modelled based on the definitions of these formalisms.

3.3 Definition of actual security policies

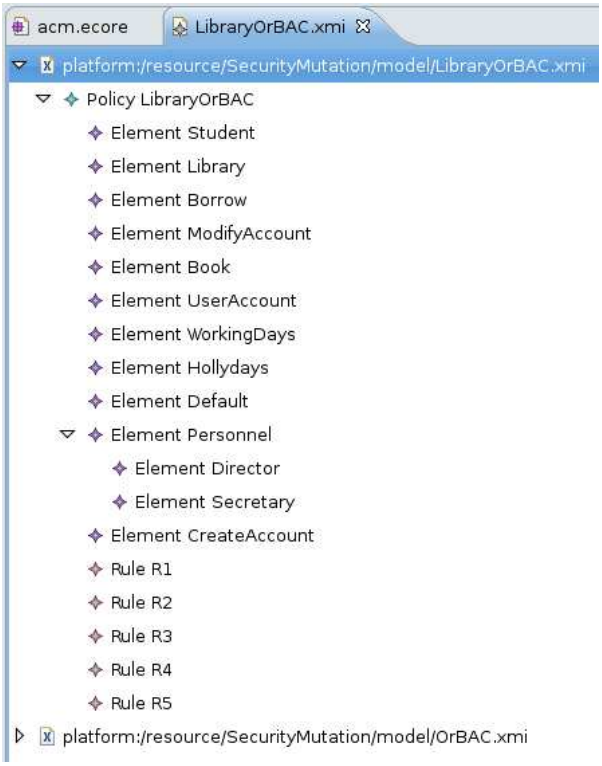

Figure 4 – OrBAC security policy for a library application

To illustrate the paper, we use the example of a library management system. Basically, the library system has various types of users: students, secretaries and a director. The students can borrow books from the library, the secretary manages the accounts of the students but only the director can create accounts. In the paper we only use a simplified version of the application with just a few security rules.

2008 IEEE International Conference on Software Testing Verification and Validation Workshop (ICSTW'08)
978-0-7695-3388-9/08 $25.00 © 2008 IEEE

In order to validate the proposed metamodel, we have modelled equivalent security policies for the application using both OrBAC and RBAC. Figure 4 shows a snapshot of the OrBAC policy in the editor generated by EMF. The model defines:

- One organization : Library
- Four roles: Student, Personnel, Secretary and Director. Among these, the roles Secretary and Director are in fact sub-roles of Personnel.
- Three activities: Borrow, ModifyAccount and CreateAccount.
- Two views: Book and UserAccount.
- Two contexts: WorkingDays and Holidays.

To illustrate the paper we have modelled five security rules (R1 to R5) for the library:

POLICY LibraryOrBAC (OrBAC)
R1 -> Permission(Library Student Borrow Book WorkingDays)
R2 -> Prohibition(Library Student Borrow Book Holidays)
R3 -> Prohibition(Library Secretary Borrow Book Default)
R4 -> Permission(Library Personnel ModifyAccount UserAccount WorkingDays)
R5 -> Permission(Library Director CreateAccount UserAccount WorkingDays)

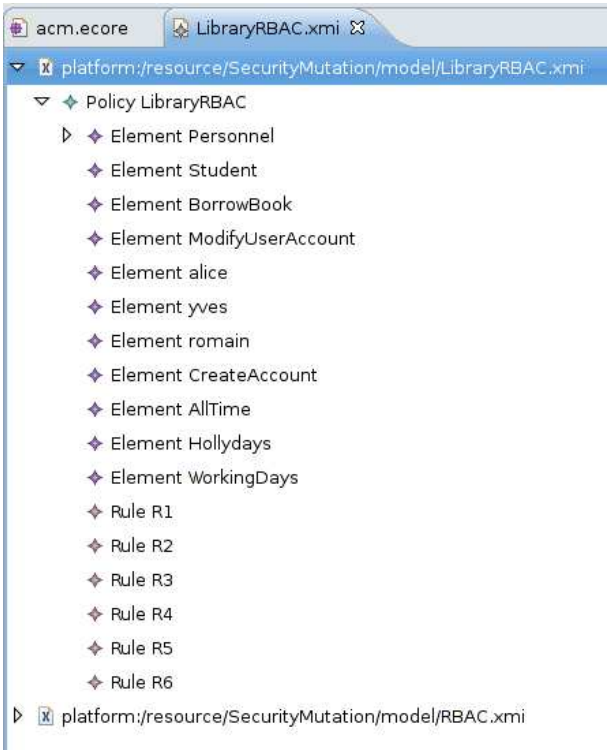

Figure 5 – RBAC security policy for a library application

A similar security policy was modeled based on RBAC. Figure 5 presents a snapshot of this model. It includes:

- Three users: alice, yves and romain.
- The same four roles as the OrBAC model.
- Three permissions: BorrowBook, ModifyUserAccount and CreateUserAccount.
- Two constraints: WorkingDays and Holidays.

Six rules were defined to associate users with roles on one hand and associate permissions with roles on the other hand:

POLICY LibraryRBAC (RBAC)
R1 -> UserRole(romain Student)
R2 -> UserRole(yves Director)
R3 -> UserRole(alice Secretary)
R4 -> RolePermission(Student BorrowBook WorkingDays)
R5 -> RolePermission(Personnel ModifyUserAccount WorkingDays)
R6 -> RolePermission(Director CreateAccount AllTime)

The next section reuses these examples to show how the generic mutation operators we propose can apply on both OrBAC and RBAC security policies.

4 The mutation operators and their implementation

This section proposes mutation operators defined on the generic metamodel proposed in the previous section. Section 4.1 presents the specifications of the operators, Section 4.2 details their implementation within the Kermeta environment and Section 4.3 shows how they can be applied to the library example.

4.1 Mutation operators

We propose four mutation operators. Each operator inherits from the SPMutator class and implements the mutate method which returns a set of mutants. The SPMutator class is related to the SecurityPolicy class from the generic security framework. This association allows the operator classes to manipulate the security models.

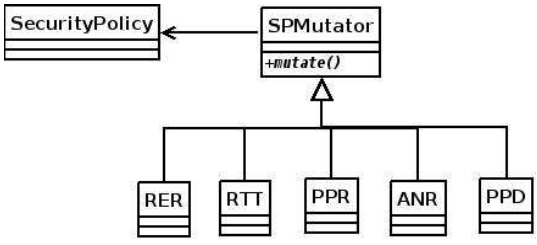

Figure 6. The mutation operator classes

2008 IEEE International Conference on Software Testing Verification and Validation Workshop (ICSTW'08)
978-0-7695-3388-9/08 $25.00 © 2008 IEEE

The following table shows the details about each operator.

Table 1- The mutation operators

Operator Name	Definition
RTT	Rule type is replaced with another one
PPR	Replaces one rule parameter with a different one
ANR	Adds a new rule
RER	Removes an existing rule
PPD	Replaces a parameter with one of its descending parameters

It is worth noting that these operators are defined at the generic level independently from any security formalism (it can be based on RBAC, OrBAC or anything else). In fact, these operators are defined based only the metamodel classes:

RTT: Finds a first rule type that has the same parameter as the type of another rule type. Then it replaces the rule parameter of one rule having the first rule type with the other rule type.

PPR: Chooses one rule from the set of the rules, and then replaces one parameter with a different parameter. It uses the knowledge provided by the metamodel (by ruleType and parameterType classes) about how rules are constructed.

ANR: Uses the knowledge about the defined parameters and the way rules are built. Then it adds a new rule that is not specified.

RER: Chooses one rule and removes it.

PPD: Chooses one rule that contains a parameter that has descendant parameters (based on the parameters hierarchies that is defined) then replaces it with one of the descendants. The consequence here is that the derived rules will be deleted and only the rule with the descendant parameter remains.

4.2 Implementation in Kermeta

Figure 7 shows the Kermeta code for the RER operator. The operator iterates on the set of rules and picks a rule to produce a mutant policy.

The Kermeta syntax and the use of metamodel allowed us to easily write the code for mutation operators that manipulate complex data structures such as security policy models. This is the case for the PPR operator shown in 8.

```
class RER inherits SPMutator {

    method mutate(p : Policy) : set Policy[*] is do
        var mutant : Policy
        result := Set<Policy>.new
        // loop on rules
        p.rules.each{ r |
            // create mutated policy
            mutant := p.copy
            mutant.name := p.name + "-RER-" + r.name
            // remove one rule
            mutant.rules.remove(mutant.rules.detect{x | x.name == r.name})
            // adds the mutant policy
            result.add(mutant)
        }
    end
}
```

Figure 7 – The RER operator

```
class PPR inherits SPMutator {

    method mutate(p : Policy) : set Policy[*] is do
        var mutant : Policy
        result := Set<Policy>.new
        // loop on rules
        p.rules.each{ rule |
            // loop on paramters
            rule.parameters.each{ param |
                // select a diffrent parameter having the same type
                p.elements.select{ e | e != param and e.type == param.type}.each { e |
                    mutant := p.copy
                    mutant.name := p.name + "-PPR-" + rule.name + "-" + param.name + "-" + e.name
                    var r : Rule init mutant.rules.detect{ x | x.name == rule.name }
                    // the rule has now a diffrent parameter
                    r.replaceParameter(r.parameters.detect{ u | u.name == param.name}, e)
                    result.add(mutant)
                }
            }
        }
    end

}
```

Figure 8 – The PPR operator

The PPR operator replaces rule parameters. It takes into account the type of parameter and produces all possible mutants by replacing on rule parameter with all possible parameters.

4.3 Examples

We show here some examples of mutants obtained for both an OrBAC and RBAC policy:

The original OrBAC policy:

2008 IEEE International Conference on Software Testing Verification and Validation Workshop (ICSTW'08)
978-0-7695-3388-9/08 $25.00 © 2008 IEEE

POLICY LibraryOrBAC (OrBAC)
R1 -> Permission(Library Student Borrow Book WorkingDays)
R2 -> Prohibition(Library Student Borrow Book Holidays)
R3 -> Prohibition(Library Secretary Borrow Book Default)
R4 -> Permission(Library Personnel ModifyAccount UserAccount WorkingDays)
R5 -> Permission(Library Director CreateAccount UserAccount WorkingDays)

Examples of mutants:

RER Mutant

POLICY LibraryOrBAC-RER-R1 (OrBAC)
R2 -> Prohibition(Library Student Borrow Book Holidays)
R3 -> Prohibition(Library Secretary Borrow Book Default)
R4 -> Permission(Library Personnel ModifyAccount UserAccount WorkingDays)
R5 -> Permission(Library Director CreateAccount UserAccount WorkingDays)

RTT mutant

POLICY LibraryOrBAC-RTS-R4-Prohibition (OrBAC)
R1 -> Permission(Library Student Borrow Book WorkingDays)
R2 -> Prohibition(Library Student Borrow Book Holidays)
R3 -> Prohibition(Library Secretary Borrow Book Default)
R4 -> Prohibition(Library Personnel ModifyAccount UserAccount WorkingDays)
R5 -> Permission(Library Director CreateAccount UserAccount WorkingDays)

Next, we present some examples of mutants related to an RBAC policy. The initial RBAC policy is presented bellow:

POLICY LibraryRBAC (RBAC)

R1 -> UserRole(romain Student)
R2 -> UserRole(yves Director)
R3 -> UserRole(alice Secretary)
R4 -> RolePermission(Student BorrowBook WorkingDays)
R5 -> RolePermission(Personnel ModifyUserAccount WorkingDays)
R6 -> RolePermission(Director CreateAccount AllTime)

Here are some examples of the generated mutants

RER mutant:

POLICY LibraryRBAC-RER-R5 (RBAC)
R1 -> UserRole(romain Student)
R2 -> UserRole(yves Director)
R3 -> UserRole(alice Secretary)
R4 -> RolePermission(Student BorrowBook WorkingDays)
R6 -> RolePermission(Director CreateAccount AllTime)

PPR mutant:

POLICY LibraryRBAC-RDD-R1-Student-Personnel (RBAC)
R1 -> UserRole(romain Personnel)
R2 -> UserRole(yves Director)
R3 -> UserRole(alice Secretary)
R4 -> RolePermission(Student BorrowBook WorkingDays)
R5 -> RolePermission(Personnel ModifyUserAccount WorkingDays)
R6 -> RolePermission(Director CreateAccount AllTime)

It is important to notice that the impact of the mutation operator depends on the access control formalism used to define a policy. The errors that are simulated are very different as shown in the examples. The same operators emulate very different flaws in the policies. For instance, the ANR operator applied to RBAC simulate the adding a new permission, while for OrBAC it will simulate adding a new prohibition or a new permission. In addition, the RER operator simulates adding a new prohibition when used for an RBAC policy, but may lead to removing a permission when used with an OrBAC policy. The impact of the operator depends on the semantic and the logic of the access control model.

5 Conclusion and further work

We presented a new approach that uses a generic metamodel for security policy mutation. This metamodel captures the concepts that are necessary to model different rule-based security formalisms. Based on this metamodel, we have modelled generic mutation operators. These operators can be applied to simulate flaws in security models expressed in the various rule-based formalisms defined with our metamodel.

We studied the feasibility of this generic approach by providing an implementation of the metamodel and the mutation operators within the Kermeta environment. The tool allowed us to define the OrBAC and RBAC security formalisms, to model security policies with these formalisms and to mutate these models by running the generic operators.

There are two main tracks for future work. The first one, that directly follows this initial study, consists in validating the mutation operators on other formalisms. In particular, we will focus on applying our approach to produce XACML mutants and analyse the difference with the mutants generated with dedicated operators. The XACML syntax can be integrated and expressed using our metamodel.

We will also study other access control models (like MAC and DAC) and check whether we can define them using our metamodel.

The second track for further work is to leverage the generic framework for security formalisms definition. It is now possible to experiment and develop other tools in addition to the mutation tool. For example, the framework eases experiments about the translation from one security formalism to another.

Acknowledgments: This work is a part of the *Politess* project (ANR-05-RNRT-01301), granted by the French National Research Agency (ANR).

2008 IEEE International Conference on Software Testing Verification and Validation Workshop (ICSTW'08)
978-0-7695-3388-9/08 $25.00 © 2008 IEEE

6 References

1. R. DeMillo, R. Lipton, and F. Sayward, *Hints on Test Data Selection : Help For The Practicing Programmer.* IEEE Computer, 1978. **11**(4): p. 34 - 41.
2. E. Martin and T. Xie *A Fault Model and Mutation Testing of Access Control Policies*. In Proceedings of *International Conference on World Wide Web*, p. 667-676, 2007.
3. T. Mouelhi, Y. Le Traon, and B. Baudry. *Mutation analysis for security tests qualification*. In Proceedings of *Mutation'07 workshop*, 2007.
4. D. F. Ferraiolo, R. Sandhu, S. Gavrila, D. R. Kuhn, and R. Chandramouli, *Proposed NIST standard for role-based access control.* ACM Transactions on Information and System Security, 2001. **4**(3): p. 224–274.
5. D. E. Bell and L.J. LaPadula. *Secure computer systems: Unified exposition and multics interpretation*. In Proceedings of *Tech. Rep. ESD-TR-73-306, The MITRE Corporation*, 1976.
6. K.J. Biba. *Integrity consideration for secure computer systems*. In Proceedings of *Tech. Rep. MTR-3153, The MITRE Corporation,*, 1975.
7. B. Lampson. *Protection*. In Proceedings of *5th Princeton Symposium on Information Sciences and Systems,*, p. 437-443, 1971.
8. A. Abou El Kalam, R. El Baida, P. Balbiani, S. Benferhat, F. Cuppens, Y. Deswarte, A. Miège, C. Saurel, and G. Trouessin. *Organization Based Access Control*. In Proceedings of *IEEE 4th International Workshop on Policies for Distributed Systems and Networks*, 2003.
9. Y. Le Traon, T. Mouelhi, and B. Baudry. *Testing Security Policies: Going Beyond Function Testing*. In Proceedings of *International Symposium on Software Reliability Engineering*, p. 93-102, 2007.
10. MotOrBAC. *The MotOrBAC Project Home Page*. Available from: http://motorbac.sourceforge.net/index.php?page=home&lang=en.
11. Kermeta. *The KerMeta Project Home Page*. 2005. Available from: http://www.kermeta.org.
12. Metamodel. *Community site for meta-modeling and semantic modeling*. 2006. Available from: http://www.metamodel.com/.
13. T. Clark, A. Evans, P. Sammut, and J. Willans, *Applied Metamodelling: A Foundation for Language Driven Development*. 2004. 189.
14. OMG. *MOF 2.0 Core Final Adopted Specification.* 2004. Available from: http://www.omg.org/cgi-bin/doc?ptc/03-10-04.
15. OMG. *UML 2.0 Object Constraint Language (OCL) Final Adopted specification*. 2003. Available from: http://www.omg.org/cgi-bin/doc?ptc/2003-10-14.
16. P.-A. Muller, F. Fleurey, and J.-M. Jézéquel. *Weaving executability into object-oriented meta-languages*. In Proceedings of *MoDELS'05*, p. 264 - 278. Montego Bay, Jamaica, October 2005.

2008 IEEE International Conference on Software Testing Verification and Validation Workshop (ICSTW'08)
978-0-7695-3388-9/08 $25.00 © 2008 IEEE