A model-based framework for security policy specification, deployment and testing

Tejeddine Mouelhi[1], Franck Fleurey[3], Benoit Baudry[2] and Yves Le Traon[1]
[1] IT- Telecom Bretagne, France. [2]INRIA/IRISA, France. [3]SINTEF, Norway.

Abstract. In this paper, we propose a model-driven approach for specifying, deploying and testing security policies in Java applications. First, a security policy is specified independently of the underlying access control language (OrBAC, RBAC). It is based on a generic security meta-model which can be used for early consistency checks in the security policy. This model is then automatically transformed into security policy for the XACML platform and integrated in the application using aspect-oriented programming. To qualify test cases that validate the security policy in the application, we inject faults into the policy. The fault model and the fault injection process are defined at the meta-model level, making the qualification process language-independent. Empirical results on 3 case studies explore both the feasibility of the approach and the efficiency of a full design & test MDE process.

Keywords: Metamodeling, Model-driven engineering methodology, Security.

1. Introduction

Security is not only a keyword, it is currently a critical issue that has to be embraced by modern software engineering (SE) techniques. From this SE point of view, having confidence in the implemented security mechanisms is the key objective when deploying a security concern. This objective can be reached by improving the design and implementation process via automation, such as security code generation from models, and by qualification criteria. Qualification criteria can be used for *a priori* verification of the specified security models' consistency, and for *a posteriori* validation of the quality of test cases executed on the final implementation. Qualification criteria should be independent of the many languages used to model security and – if possible – from the specificity of implementation languages. The same principles and same criteria should be applied to qualify the security mechanisms for various kinds of systems. Although such common principles and criteria are highly desirable, they are useless without the support of effective tools that make the qualification approach applicable in a final system implementation.

In this paper we focus on the confidence of an *access control policy* proposing a bi-dimensional approach, which is (1) an automated MDE process for deriving security components, (2) a qualification process which is security-language independent.

The first dimension of the approach thus targets the automation of the process transforming a security policy into security components. This process is based on a standard architecture that involves designing a dedicated security component, called the policy decision point (PDP), which can be configured independently from the rest of the implementation containing the business logic of the application. The execution of functions in the business logic includes calls to the PDP (called PEPs – policy enforcement points), which grant or deny access to the protected resources/functionalities of the system.

The proposed MDE process is based on a domain-specific language (DSL) in order to model security formalisms/languages as well as security policies defined according to these formalisms. This DSL is based on a generic metamodel that captures all the necessary concepts for representing rule-based access control policies. The process relies on several automatic steps in order to avoid errors that can occur with repeated manual tasks. This includes the automatic generation of a specific security framework, the automatic generation of an executable PDP from a security policy and the automatic injection of PEPs into the business logic using aspect-oriented programming.

In this context, the challenge for qualification is to offer some guarantee that the PDP is consistent and that it interacts with the business logic as expected (e.g. a hidden security mechanism may bypass some PEPs [1]). Since this interaction of the PDP with the business logic can be faulty, a second dimension is needed to improve the confidence in security mechanisms.

This second dimension consists of a qualification environment which provides (1) *a priori* verifications of security models before PDP component and PEP generation, (2) *a posteriori* validation of the test cases for the implementation of the policy. This qualification environment is independent of security policy languages and is based on a metamodel for the definition of access control policies. It provides model transformations that make the qualification techniques applicable to several security modeling languages (e.g. RBAC, OrBAC [2, 3]).

Section 2 presents the MDE process. Section 3 details the framework, while section 4 explains how we exploit this metamodel for security code generation and V&V certification. Section 5 presents the empirical results on three Java case studies.

2. MDE for Security

Errors in the security policy can have several causes. They can be caused by an error in the policy definition, by an error when translating the policy into an executable PDP or by an error in the definition of PEPs (calls to the PDP at the wrong place in the business logic, calls to the wrong rule, missing call, etc.). In order to increase the efficiency of the validation and verification tasks, we investigate an integrated approach based on two facets. First, we want to ensure as much quality as possible by construction. For that purpose we propose an MDE process based on a specific modeling language for security policies and automatic transformations to integrate this policy into the core application. Second, we develop generic verification and validation techniques that can be applied early in the development cycle and that are independent of any particular security formalism.

Figure 1 presents an overview of the proposed approach. From top to bottom, the process starts with the requirements for the system and ensures the integration of the security concerns in the running code.

The first step of the approach (1) is to build the security model for the application. The security model is a platform independent model which captures the access control policies defined in the requirements of the system. To allow security experts to create, edit and check this security model, it is modeled in a security domain specific modeling language. This language is based on a generic meta-model which allows several types of rule-based access-control policy to be expressed. In practice, the meta-model allows the type of rules to be modeled as well as the rules themselves.

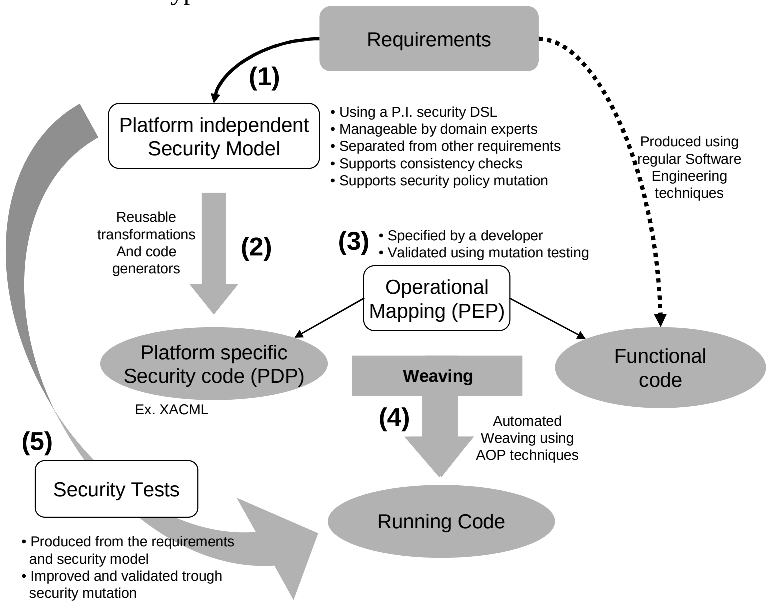
Requirements
(1)
Platform independent Security Model
• Using a P.I. security DSL
• Manageable by domain experts
• Separated from other requirements
• Supports consistency checks
• Supports security policy mutation
Produced using regular Software Engineering techniques
Reusable transformations And code generators
(2)
(3)
• Specified by a developer
• Validated using mutation testing
Operational Mapping (PEP)
Platform specific Security code (PDP)
Ex. XACML
Weaving
Functional code
(4)
Automated Weaving using AOP techniques
(5)
Security Tests
Running Code
• Produced from the requirements and security model
• Improved and validated trough security mutation

Figure 1 – Overview of the proposed approach

The main benefit of using a generic meta-model is that it allows the implementation of generic verifications and transformations and their application to all types of rule. Verifications are performed in order to check the soundness of the security policy and its adequacy with regards to the requirements of the application, in particular, the detection of conflicts between rules, which is a tedious task. A simple case of conflict occurs when a specific permission is granted by a rule and denied by another (possible with OrBAC language). These verifications also include checking that each business operation can be performed by at least one type of user or that a specific set of operations can only be performed by administrators. All these verifications are carried out by the security metamodel and are thus language independent.

After the platform independent security model has been validated, automated transformations are used to produce specific PDPs (2). The Policy Decision Point is the point where policy decisions are made. It encapsulates the Access Control Policy and implements a mechanism to process requests coming from the PEP and return a response which can be permit or deny In practice, the PDP is not usually fully generated but based on reusable security frameworks. These frameworks are configured with the specific security policy and connected to the application through the PEP. In the example presented in the figure, the output of the transformation is an XACML file that contains the security policy.

A critical remaining step for implementing the security of the application is to connect the security framework with the functional code of the application (3). In practice, this corresponds to adding the PEPs in the main code of the application. It is a critical step because any mistakes in the PEP can compromise the overall security of the application. To reduce the risk of mistakes, we use AOP, to make the security PEP introduction systematic (4), and mutation testing to ensure that the final running code conforms to the security model (5). An important reason to compose the security concern at the code level is to limit the assumptions on how the system is developed prior to this composition. Another reason is that this makes it possible to change the security policy after deployment.

The use of aspect-oriented programming allows a separation to be kept between the business logic of the application and the security code. The use of pointcuts allows calls to be introduced to the security code systematically without having to list all the specific locations. This makes the PEP easier to specify and understand, which avoids many potential mistakes. However, the use of AOP does not provide a warranty that the security policy is appropriately integrated into the code of the application. A careful validation of the resulting code with respect to the security model is required.

In the proposed approach, the validation is done by testing the final running code with security-specific test cases. To properly validate the security of the application, these test cases have to cover all the security features of the application. When testing, *test criteria* are used to ensure that the tests are “good enough” to assess the quality of the system under test. The test criteria we use are based on the mutation of the security model. Mutation testing is a test qualification technique introduced in [4] which has recently been adapted for security testing [5, 6].

The intuition behind mutation testing applied to security is that the security tests are qualified if they are able to detect any elementary modification in the security policy of the application (mutants). The originality of the proposed approach is to perform mutations on the platform independent security model using generic mutation operators. Since the transformation and weaving of the security policy in the application are fully automated, the tests can be automatically executed on the mutants of the application. If the tests are not able to catch a mutant then new test cases should be added to exercise the part of the security policy which has been modified to create this mutant. In practice the undetected mutants provide valuable information for creating new tests and covering all the security policies.

The objective of the test cases in a partly automated generation process (PDP and PEPs) is to check that the security policy of the application is fully synchronized with the security model. Indeed, in the mutation testing process, the model is modified and the tests are required to detect the modification in the running code. Thus, if any of the security rules are hard-coded in the application or if the application code bypasses the security framework, the modification will not be observable in the running code. In that case, the mutants corresponding to the hard-coded or bypassed security feature are not detected and the tester has to fix the application to make sure that all security rules are come from the security model.

Overall, the main benefit of the approach is to allow the security policy to be validated using verification on the security model and testing that the policy implemented in the application conforms to the security model. Because the testing is performed on the final running code it allows validation both that the PDP is according to the model but also that the PEP, i.e. the integration with the rest of the application, is correct. The following sections detail the main steps of the approach.

3. The access control metamodel

In the literature, several access control formalisms such as RBAC or OrBAC are based on the definition of security rules. All these formalisms allow the definition of rules that control the access to data, resources or any type of entity that should be protected. The formalisms differ according to the type of rules and entities they manipulate. Since we want to provide validation mechanisms that can be used with various formalisms, we have defined a metamodel that captures the common concepts of rules and entities that are necessary to define these formalisms. The metamodel also captures the concerns for the definition of access control policies using these formalisms.

In this section we introduce this metamodel and illustrate how it supports the definition of RBAC and OrBAC

3.1. Generic security metamodel

The metamodel, shown Figure 2, is divided into two parts, that correspond to two levels of instantiation:

- It is possible to define a security policy formalism with the classes POLICYTYPE, ELEMENTTYPE and RULETYPE. A POLICYTYPE defines a set of element types (ELEMENTTYPE) and a set of rule types (RULETYPE). Each rule type has a set of parameters that are typed by element types.
- Based on a security formalism defined by instantiating the above classes, it is possible to define a policy using the classes POLICY, RULE and PARAMETER. A POLICY must have a type (an instance of POLICYTYPE) and defines rules and parameters. The type of a policy constrains the types of paramters and rules it can contain. Each parameter has a type which must belong to the element types of the policy type. If the *hierarchy* property of the parameter type is true, then the parameter can contain children of the same type as itself. Policy rules can be defined by instantiating the RULE class. Each rule has a type that belongs to the policy type and a set of parameters whose types must match the types of the parameters of the type of the rule.

These two parts of the metamodel have to be instantiated sequentially: first define a formalism, then define a policy according to this formalism.

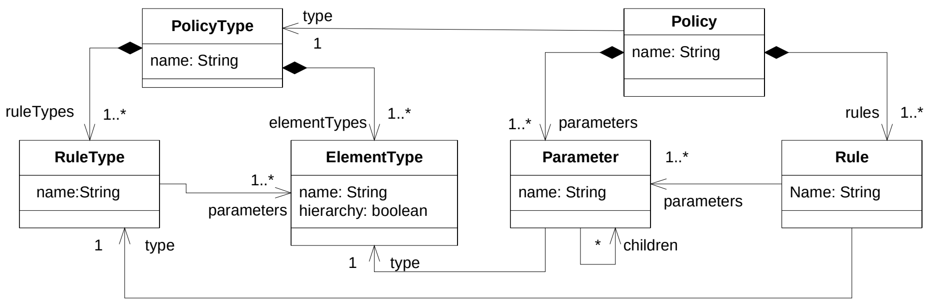

Figure 2 – The meta-model for rule-based security formalisms

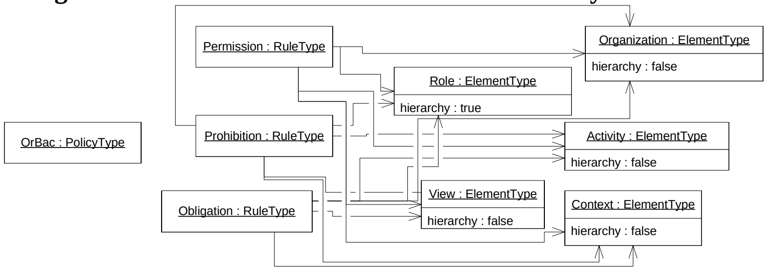

Figure 3 - The OrBAC security formalism

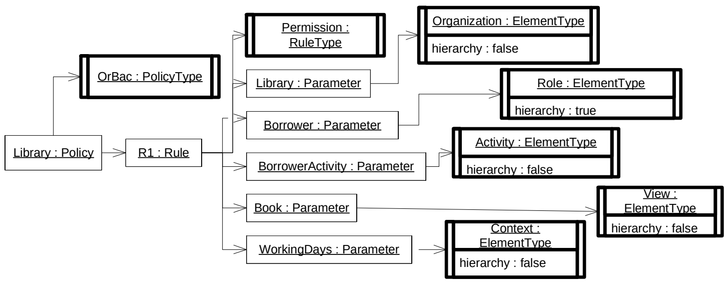

Figure 4 - Rule R1 for the library security policy

3.2. Instantiating the metamodel

The two parts of the metamodel are instantiated at different moments and by different people. The classes that capture the concepts for a security formalism have to be instantiated first. They are instantiated by language engineers, and this instance defines a modeling language that can be used to model a security policy for a particular system. The classes that capture the concepts to define a policy can only be instantiated if a formalism has been modeled.

To illustrate this process, we redefine the OrBAC formalism as an instance of the generic metamodel and then we define an OrBAC security policy. In OrBAC there are five types of entities: organizations, roles, activities, views and contexts. OrBAC defines three types of rules: permission, prohibition and obligation. All three types of rule have the same five parameters: an organization, a role, an activity, a view and a context. Figure 3 shows how the metamodel was instantiated to model OrBAC security policies.

To illustrate our approach, we use the example of a library management system. This system defines three 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 use only a simplified version of the application with just a five security rules defined below:

```
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 4 shows an instance of the metamodel for rule R1. The objects in bold are objects that are defined by the OrBAC formalism in Figure 3. Rule R1 is well-formed because, as we can see in Figure 4, the rule has five parameters, each of them of the correct type: Organization, Role, Activity, View, Context. `Borrower` and `BorrowerActicity` are hierarchies of parameters. Rule R1 uses the sub-parameter `Student` defined in the `Borrower` hierarchy and `Borrow` in the `BorrowerActivity` hierarchy.

4. Exploiting the security metamodel

This section introduces two mechanisms that we have developed around the generic metamodel. First, we illustrate the fault models that we have defined at the meta-level and that can be executed to inject errors into security policies. Second, we detail how we transform a security policy, defined as an instance of the security metamodel, into a platform-specific security policy that can be ‘hooked’ onto the business logic.

4.1. Mutation testing for security

Mutation analysis involves qualifying a set of test cases for a program under test (PUT) according to the rate of injected errors they can detect. The assumption is that if test cases can detect errors that have been injected on purpose, they will be able to detect actual errors in the PUT. The validity of mutation analysis greatly depends on the relevance of faults that are injected. Faults are modeled as mutation operators that reflect typical faults that developers make in a particular language or domain. Several works (Xie et al. [5], Le Traon et al. [6]) have proposed mutation operators to validate test cases for security policy.

In this paper, we define five mutation operators for security policy testing, shown in Table 1. These operators are defined only in terms of the concepts present in the security metamodel, which means that they are independent of a specific security formalism. Thus, these operators can be applied to inject errors into any policy expressed with any formalism defined as an instance of our metamodel. The definition of mutation operators at this meta-level is critical for us since it allows the qualification of test cases with the same standard, whatever the formalism used to define the policy.

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

In order to generate faulty policies according to these operators, we have added one class to the metamodel for each operator.

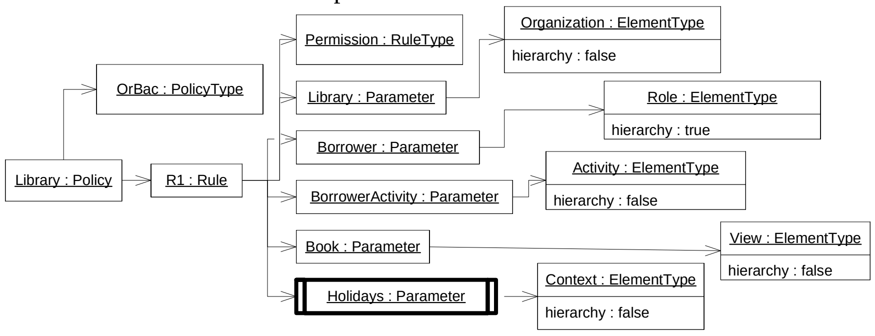

Figure 5 - Rule R1 mutated with PPR operator

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 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 parameter hierarchies that are 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.

As an example, Figure 5 shows a mutant obtained by applying the PPR operator on rule R1 of the policy for the library system. In bold we can see the parameter that has been changed: the context has been changed form WorkingDays to Holidays.

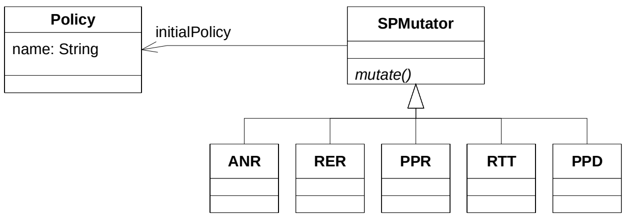

Figure 6. The mutation operator classes

Figure 6 shows the operator classes. The `mutate()` method is implemented in Kermeta. What is important to notice in this method is that it is defined only using concepts defined in the metamodel. Thus, this method can generate a set of mutated policies, completely independently of the formalism they are defined with. It has to be noted that some mutant policies can be strictly equivalent to the initial policy. In that case no test case can detect the mutant. For mutation analysis, these equivalent mutants have to be removed.

The next section details the deployment of a security policy according to a particular formalism. Then this platform-specific policy is linked to the business logic through the PEPs. Thus, once a policy is mutated at the meta level, the injected error is automatically transformed into an error in the final system that can then be detected by a test case.

4.2. PDP XACML code generation from the metamodel

We implemented a tool to generating XACML (Extended Access Control Markup Language) policies from our metamodel. The XACML file is added to the PDP. XACML is an OASIS standard dedicated to defining access control policies in XML files. There is a framework [7] that supports and helps to access, write and analyze XACML files. Writing XACML files manually is an error-prone task, due to the language complexity. In this paper, we used the existing XACML profiles for RBAC and OrBAC to generate XACML files from our metamodel. Due to space limitation, we do not detail these two profiles.

a) The PDP and the PEP

The PDP encapsulates the XACML policies and offers services to access the policy. The PEP is implemented in the applications and contains a mapping between the application concepts (user roles, methods, resources and contexts) and the OrBAC/RBAC policy concepts (roles, activities, views and contexts, or roles, permissions and constraints).

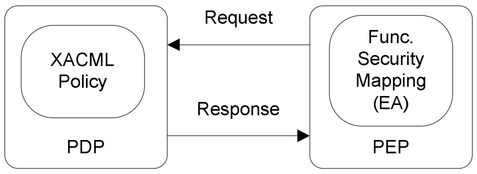

Figure 7 – PEP and PDP

As shown in Figure 7, a PEP uses the mapping between the application concepts (user roles, methods, resources and contexts) and the OrBAC abstract policy (roles, activities, views and context) to get the OrBAC abstract entities. Then it sends a request to the PDP containing the role, the activity, the view and the context. The PDP responds with permit or deny.

b) Limitations

Depending on the underlying model (OrBAC or RBAC), the appropriate profile is used to generate XACML files. Therefore the XACML generator is not generic and a specific generator is needed for each model. However, it can be reused because it is defined at the model level and thus independent of the specified generic policy.

It is important to note that the tool allows XACML files to be generated for both the actual policy and the mutant policies. These mutants are generated at the generic level and the generator tool is used to create XACML files for these mutant policies. Thanks to this tool and the PDP architecture created, the mutation analysis becomes a 'push-button' technology. The mutants are automatically generated at the metamodel level. Then they are exported to XACML to be used by the PDP to evaluate the existing security tests. Therefore, our approach offers a powerful framework for evaluating security policy tests using mutation independently of the underlying access control language.

5. Case studies and results

We applied the Verification (V), the automated deployment of PDP and PEPs and the Validation (&V) on three systems:

LMS: A Library Management System.

VMS: A Virtual Meeting System.

ASMS: An Auction Sale Management System.

Table 2 gives the size of the 3 applications (the number of classes, methods and lines of code LOC).

Table 2 – The size of the three applications

	# classes	# methods	LOC (executable statements)
LMS	62	335	3204
VMS	134	581	6077
ASMS	122	797	10703

5.1. The process

In this section, we detail the verification step and the creation of PEP by aspect weaving. The results concerning the validation of the three case studies will be synthesized in the next subsection.

a) The verification step

The main verification is offered by construction, when the conformity of a security model to its metamodel is checked. In this way, we have a minimum consistency that is obtained with the conformance relationship of a model to its metamodel. We also add some extensible verification functions. They constitute preconditions which are checked before deploying the policy and generating the mutants. Two verification functions are implemented:

1. `policy_is_conform()`: the policy conformance to the underlying policy type (OrBAC or RBAC). In order to guarantee that the defined rules meet the types of parameters of rules defined by the policy type.
2. `no_conflicts()`: checks the absence of conflicts. It essentially involves checking that there are no rules having the same parameters and having different types.

The conformance verification function detects simple erroneous rules such as wrong parameters or wrong numbers of parameters.

b) Aspect weaving of the PEP

An aspect is composed of two main parts: (1) advice that implements the behavior of the cross-cutting concern, and (2) the pointcut descriptor (PCD) that designates a set of joinpoints in the base program where the advice should be woven. The weaving is performed automatically by a specific compiler. In the context of the development of secured applications, we propose to define PEPs as aspects, as shown in the following listing. The call to the PDP is weaved before the execution of the method. If the access is granted, the execution continues, otherwise a security exception must be raised. The PEPAspect aspect (implemented in AspectJ in the following listing) defines an advice that is woven before each execution of BookService.borrowBook. This advice calls the checkSecurity method that will then call the PDP by sending the role, activity, view and context given in the parameter. Note that the advice throws a SecurityPolicyViolationException. This exception is raised by checkSecurity when the PDP responds with a prohibition.

```
public aspect PEPAspect {
// PEP Joinpoint for borrow
before(User user,Book book) throws
SecuritPolicyViolationException :
borrowBookCall(user,book)  {

// Call to check for security rule
checkSecurity(user.getRole(), BORROWMETHOD
,BOOKVIEW, getTemporalContext());
       }
}
```

5.2. Validation results

The validation of the PDP interacting with the business logic was done for the three case studies. The objective of these empirical studies is twofold:

1. Is the approach feasible and what results do we obtain when we modify the security policy language?
2. What are the differences between MDE and language specific approaches?

To answer both questions, we have an existing basis, which are the functional tests and the security tests which were generated to validate the specific OrBAC-based mutation approach [1, 8]. So we have two sets of test cases, which are *functional test cases* and *security policy test cases*.

System/functional tests: test cases generated based on the uses cases and the business models (e.g. analysis class diagram and dynamic views) of the system. Contrary to security tests, we call these tests functional.

Security policy tests: test cases generated specifically from a security policy. The objective of SP testing is to reveal as many security flaws as possible. In our case, the security test cases are generated from the OrBAC or RBAC policies. The test criterion we use requires that each access control rule must be tested by at least one test case. Since OrBAC requires more rules to specify a security policy equivalent to an RBAC one (prohibition rules can be explicitly specified), the test cases generated for OrBAC also satisfy the RBAC test criterion.

a) Process feasibility and results

For the first question, we applied the full generation process for the three case studies, with RBAC and OrBAC access control languages. The feasibility of the approach is thus demonstrated by these successful deployment and validation steps. The limit of this feasibility analysis is that the targeted systems are homogeneous, which means that the final execution platform is based only on Java. We believe that a sufficient number of systems are built with such platform assumptions, making the approach relevant at least for these systems. The number and categories of mutants that are generated for both security policy languages are given in Table 3. In this table we separate ANR mutants from non-ANR because they both generate very different types of mutant: non-ANR operators mutate the existing rules whereas ANR adds new rules. It is interesting to note that the number of mutants may vary significantly depending on the language chosen. However, based on this observation, we cannot conclude which language is likely to be error-prone.

Figure 8 presents the comparative mutation scores (percentage of mutants that are detected as faulty by the test cases) when we execute functional and security test cases on both implementations. We can thus compare how efficient each category of test generation is by using this qualification technique. Here, efficiency is evaluated by the ability of test cases at detecting specific types of faults. OrBAC non-ANR mutants (basic mutation operators) are more difficult to detect with functional test cases than RBAC mutants. This is interesting, since it shows that qualifying functional test cases using OrBAC may be a good choice. Clearly, security test cases are qualified for both languages. Second, concerning ANR mutants, which qualify whether the test cases exercise the "by default" policy of a system, it appears that functional test cases are not efficient, whatever the security policy language is. Circles pinpoint the very low mutation scores obtained especially with OrBAC policies. This is partly due to the fact that some ANR mutants we generate for OrBAC do not modify the behavior of the application, in other words, they are *equivalent mutants*. However, even after filtering equivalent mutants, both functional and security test cases obtain low scores. In the case of the LMS system, the differences in terms of quality between functional and security test cases for RBAC are quite visible (3% compared with 95 % mutation scores). Test cases sets that do not reach a high mutation score have to be completed to be qualified. In a previous work [1], we presented the real security faults we found in these cases studies, especially hidden security mechanisms. In summary, a same fault model leads to the injection of specific faults, which shows the interest of the technique.

Table 3 – Number of mutants generated for RBAC/OrBAC policies

	RBAC				OrBAC			
System\ mutants	rules	NON ANR	ANR	All	rules	NON ANR	ANR	All
LMS	23	437	257	694	42	330	714	1044
VMS	36	972	396	1368	106	426	1046	1572
ASMS	89	2937	647	3584	130	1138	1950	3088

Table 4 – OrBAC specific mutants vs Generic mutants

System	generic mutants	specific mutants
LMS	1044	371
VMS	1572	1426
ASMS	3088	2056

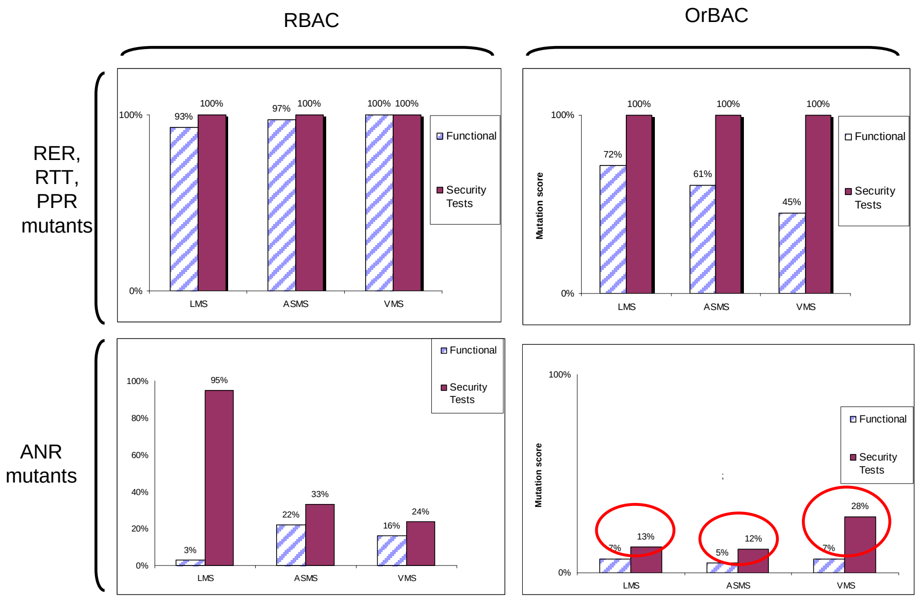

Figure 8 – Mutation scores per mutation operator and language

b) MDE approach versus language-specific approach

Table 4 compares the number of mutants we obtain with OrBAC policies using a specific approach instead of the generic approach presented in this paper. Table 5 is more important since it presents the mutation scores, with functional and security test cases, obtained with the generic and the specific approach. The specific approach has been presented in [6, 8] and it benefits from the language and the dedicated platform (MotOrBAC tool associated with the language) to generate mutants. The delta reveals that, for all cases, the variation of mutation scores is lower than 10%. This delta is due to the generation of more mutants with the generic approach, and reflects the proportion of equivalent mutants when using the generic approach. The lack of quality of some mutants generated using the MDE process is counter-balanced by the interest of having a certification process that is language-independent. Moreover, this distance separating generic and specific can be reduced by the addition of a mutant filtering function at the language level.

Table 5 – OrBAC mutation results vs. Generic mutation results

Mutants	**Basic Mutants (func. tests)**			**ANR mutants (sec. tests)**		
System	**LMS**	**VMS**	**ASMS**	**LMS**	**VMS**	**ASMS**
Generic mutants	72%	61%	45%	13%	12%	28%
Specific mutants	78%	69%	55%	17%	19%	33%
Delta	**-6%**	**-8%**	**-10%**	**-4%**	**-7%**	**-4%**

6. Related works

Previous work focused on providing model based methodologies for security. UMLsec [9] which is an extension of UML allows security properties to be expressed in UML diagrams. In addition, Lodderstedt et al. [10] propose SecureUML, which is close to our contribution, especially concerning the generation of security components from dedicated models. The approach proposes a security modeling language to define the access control model. The resulting security model is combined with the UML business model in order to automatically produce the access control infrastructure. More precisely, they use the Meta-Object facility to create a new modeling language to define RBAC policies (extended to include constraints on rules). They apply their technique in different examples of distributed system architectures including Enterprise Java Beans and Microsoft Enterprise Services for .net. The two processes differ since we do not merge the security and the business models but generate security components independently focusing on different aspects, even if SecureUML could be added for two reasons. First, we consider the functional design and the deployment of the access control model as independent processes which are finally merged. In addition, we consider the validation of the implementation and include it in our process. Moreover, our approach is generic and independent of the underlying access control model.

Previously, mutation has been applied to security policy testing. Xie et al. [5] proposed a mutation fault model specific to XACML policies. However, they had to deal with the problem of equivalent mutants due to the fact that mutants are seeded at code-level. Since Xie et al.'s work aims at testing the Policy Decision Point alone, implemented with XACML, their testing approach does not execute the business logic of the system. The same PDP-alone based testing approach is considered by Mathur et al. [11], who also mutate RBAC models by building an FSM model for RBAC and use strategies to produce test suites from this model (for conformance testing).

7. Conclusion

We have presented a new approach that uses a generic metamodel for security policy specification, deployment and testing. Our approach allows the access control model to be defined in a generic way, independently of the underlying access control language. The security policy is then exported to an XACML file that will be included in the PDP. Then, the PEP is implemented using AOP. Finally, we define mutation at a generic level and use it to qualify the security tests validating the implementation of the security policy. The feasibility of the approach and the interest of a common V&V are both illustrated by applying the approach to 3 case studies.

8. References

1. Y. Le Traon, T. Mouelhi, A. Pretschner, and B. Baudry, *Test-Driven Assessment of Access Control in Legacy Applications*, in *ICST 2008: First IEEE International Conference on Software, Testing, Verification and Validation*. 2008.
2. 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.
3. 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 *IEEE 4th International Workshop on Policies for Distributed Systems and Networks*. 2003.
4. 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.
5. E. Martin and T. Xie. *A Fault Model and Mutation Testing of Access Control Policies*. in *Proceedings of the 16th International Conference on World Wide Web*. 2007.
6. T. Mouelhi, Y. Le Traon, and B. Baudry, *Mutation analysis for security tests qualification*, in *Mutation'07 : third workshop on mutation analysis in conjuction with TAIC-Part*. 2007.
7. Sun's XACML implementation. http://sunxacml.sourceforge.net/. [cited.
8. Y. Le Traon, T. Mouelhi, and B. Baudry, *Testing security policies : going beyond functional testing*, in *ISSRE'07 : The 18th IEEE International Symposium on Software Reliability Engineering*. 2007.
9. J. Jürjens. *UMLsec: Extending UML for Secure Systems Development*. in *Proceedings of the 5th International Conference on The Unified Modeling Language*. 2002.
10. Torsten Lodderstedt, David Basin, and Jürgen Doser. *SecureUML: A UML-Based Modeling Language for Model-Driven Security*. in *Proceedings of the 5th International Conference on The Unified Modeling Language*. 2002.
11. Ammar Masood, Arif Ghafoor, and Aditya Mathur, *Scalable and Effective Test Generation for Access Control Systems that Employ RBAC Policies*. 2006.