A Model-Based Approach to Automated Testing of Access Control Policies

Dianxiang Xu, Lijo Thomas, Michael Kent
National Center for the Protection of the Financial Infrastructure, Dakota State University
Madison, SD 57042, USA
+1 605 256 5694
{dianxiang.xu, lthomas, mjkent}@dsu.edu

Tejeddine Mouelhi, Yves Le Traon
Interdisciplinary Centre for Security, Reliability and Trust, University of Luxembourg, Campus Kirchberg
L-1359, Luxembourg, Luxembourg
+352 46 66 44 5840
{tejeddine.mouelhi, Yves.LeTraon}@uni.lu

ABSTRACT
Access control policies in software systems can be implemented incorrectly for various reasons. This paper presents a model-based approach for automated testing of access control implementation. To feed the model-based testing process, test models are constructed by integrating declarative access control rules and contracts (preconditions and post-conditions) of the associated activities. The access control tests are generated from the test models to exercise the interactions of access control activities. Test executability is obtained through a mapping of the modeling elements to implementation constructs. The approach has been implemented in an industry-adopted test automation framework that supports the generation of test code in a variety of languages, such as Java, C, C++, C#, and HTML/Selenium IDE. The full model-based testing process has been applied to two systems implemented in Java. The effectiveness is evaluated in terms of access-control fault detection rate using mutation analysis of access control implementation. The experiments show that the model-based tests killed 99.7% of the mutants and the remaining mutants caused no policy violations.

Categories and Subject Descriptors
D.2.5 [**Testing and debugging**]: Testing tools (e.g., data generators, coverage testing). D.4.6 [**Security and protection**]: Access Controls

General Terms
Reliability, Security, Verification.

Keywords
Access control, software testing, model-based testing, Petri nets, mutation analysis.

1. INTRODUCTION
Access control is a fundamental mechanism for providing security-intensive software with first-level security by regulating user access to resources. An access control policy is usually expressed in terms of declarative rules, defining the conditions to which the access to resources can be granted and to whom. Although the specification of an access control policy can be supported by powerful verification techniques, the specified policy and its mechanism may not be implemented correctly for various reasons, such as programming errors, omissions, and misunderstanding of the policy specification. The flaws in an incorrect implementation may result in serious violations of access control policy, such as unauthorized accesses and escalation of privileges. Therefore, it is important to reveal the potential discrepancy between the policy specification and the actual implementation.

Software testing is a major means for software quality assurance. It aims at finding errors by executing a program with test cases, including test inputs and test oracles (expected results). To reveal access control violations, one approach is to devise test cases for individual access control rules. The main issue of testing individual rules, however, is that it cannot see the forest for the trees because access control rules are often related to each other. In a library management system, for example, access control rules may be defined for such activities as borrow and return, where a precondition of return is that there is a borrowed book. Testing the individual borrow and return rules would lead to duplicated tests – testing the return activity typically involves a borrow activity. In addition, it is difficult to cover all the interactions among access control activities by testing individual rules.

To address the above issue, this paper presents a model-based approach to testing access control policies. Model-based testing uses models of a system under test (SUT) for generating test cases. It is an appealing approach to software testing because of several potential benefits [1]. First, the modeling activity helps clarify requirements and enhances communication between developers and testers. Without a good understanding about the SUT, testers would not be able to perform effective testing. Second, automated test generation enables more test cycles and assures the required coverage of test models. Third, model-based testing can help improve fault detection capability due to the increased number and diversity of test cases [2]. Nevertheless, studies have shown that the tester's ability to build quality models or required expertise in rigorous modeling is a major barrier to the effective application of model-based testing [3]. There is little work on how to build access control test models in a structured, repeatable process. Existing literature typically focuses on what modeling notation is used and how tests are generated and executed. Another issue is that abstract tests generated from models need to be transformed into concrete tests for execution, which can be a time-consuming process. As will be detailed in the related work section, these two issues remain largely open.

SACMAT'12, June 20–22, 2012, Newark, New Jersey, USA.

The approach in this paper generates executable access control tests from a MID (Model-Implementation Description) specification, which consists of an access control test model and a MIM (Model-Implementation Mapping) description. The underlying test model, represented by a Predicate/Transition (PrT) net [4][5][6], is constructed from the given access control rules and functional requirements according to which the SUT is designed and implemented. PrT nets are high-level Petri nets, a well-studied formal method for system modeling and verification. We use contracts (preconditions and post-conditions) to construct test models for two considerations. First, design by contracts [7] is a widely accepted approach to functional specification. Second, access control rules as security constraints on system functionality cannot be tested without involving system functionality. Access control testing requires understanding of the preconditions and post-conditions of the related activities. Consider testing the rule that a student is allowed to return books on working days. The test cannot be performed unless the functional precondition *"book is borrowed"* is satisfied. The accurate test oracle cannot be determined without knowing its post-condition "*book becomes available*". For test generation purposes, we integrate declarative access control rules and contracts into an operational PrT net. For code generation purposes, we create the MIM description by mapping the elements in a test model to the implementation constructs based on the SUT's programming interface. The generated code can then be executed with the SUT.

Our approach has been implemented in MISTA (formerly ISTA)[1], a framework for automated generation of test code in a variety of languages, including Java, C, C++, C#, and HTML/Selenium IDE (a Firefox plugin for testing web applications) [6][8]. We have conducted case studies using two Java applications, LMS (a library management system) and ASMS (an auction sale management system) [9][10]. To assess the fault detection capability of our approach, we applied mutation analysis of access control implementation. Mutants were created by seeding faulty rules in policy implementation. A mutant is said to be killed or detected if a failure is reported during at least one test execution. Mutation analysis is a widely applied method for evaluating the effectiveness of software testing techniques. Since the injected faults would represent the defects that likely occur in software implementation, the percentage of mutants killed by the test cases created from a testing technique is often a good indicator of how effective the testing technique is (For further information on mutation testing, a survey can be found in [11]). For each case study, we constructed the access control test models in the subject program, generated executable tests from the test models, and executed the tests against the mutants. Our experiments show that our approach is highly effective in detecting policy violations since the generated tests killed a large percentage of mutants.

The contribution of this paper is threefold. First, we formalize several desired characteristics of role-based access control rules (consistency, non-redundancy, and completeness) and deal with incomplete specification of access control rules. Incompleteness of specification is a norm in real-world software development and the undefined situations more likely lead to security holes in the implementation. Second, we present an automated process for constructing operational test models by integrating declarative access control rules and contracts into PrT nets. The test models can cover all access control rules and contexts. Third, we generate executable test code automatically to cover the access control rules and their contexts. Once the MID specification is completed, test generation and execution would need no human intervention. To the best of our knowledge, none of the above aspects has been addressed in the literature on model-based access control testing.

The remainder of this paper is organized as follows. Section 2 introduces the role-based access control model used in this paper, formalizes the desired characteristics of access control rules, and deals with incomplete access control rules. Section 3 describes how test models are constructed from access control rules and contracts. Section 4 discusses how executable test code is generated from test models. Section 5 presents the case studies. Section 6 reviews and compares our approach to the related work. Section 7 concludes this paper.

2. The Role-Based Access Control Model

2.1 Role-based Access Control

Our approach is based on role-based access control (RBAC) extended with the contexts and prohibition rules. An access control policy consists of the following elements:

- A set of roles *R,*
- A role hierarchy *H*,
- A set of objects (or resources) *O*,
- A set of contexts *C*,
- A set of operations *A* (called activities in this paper),
- A set of authorization types *{Permission, Prohibition}*,
- A set of subjects (human users or computer agents) *Sub*,
- A role assignment $Sub \rightarrow 2^R$ (one subject may play a set of roles), and
- A set of role-based access control rules $\Re$. $\Re(r)$ is the set of access control rules defined for role *r*.

Definition 1 (Access control rule). An access control rule is a 5-tuple $<r, o, a, c, \tau>$, where $r \in R$, $o \in O$, $a \in A$, c is a Boolean expression representing the policy's context, and $\tau \in \{$ *Permission, Prohibition*$\}$. It means that role *r*'s activity *a* on object *o* is permitted (when τ=*Permission*) or prohibited (when τ=*Prohibition*) when context *c* holds.

Table 1. Access control rules for borrower in LMS

No	Object	Activity	Context	Auth_Type
1	Book	GiveBackBook	day(HD)	Prohibition
2	Book	BorrowBook	day(HD)	Prohibition
3	Book	BorrowBook	day(WD)	Permission
4	Book	GiveBackBook	day(WD)	Permission
5	Book	ReserveBook	day(HD)	Prohibition
6	Book	ReserveBook	day(WD)	Permission

A role hierarchy $H \subseteq R \times R$ is a partial-order relation on *R*. Given $(r', r) \in H$, *r'* is said to be a direct super-role of *r*, and *r* is a direct sub-role of *r'*. Role *r* is called a primitive role if *r* is a leaf in the role hierarchy. In LMS, for example, the set of roles is *{student, teacher, director, secretary, admin, borrower, personnel}*, the role hierarchy is *{<borrower, student>, <borrower, teacher>, <personnel, director>, <personnel, secretary>}* (*borrower* is the super-role of *student* and *teacher*, whereas *personnel* is the super-role of *director* and *secretary*), the set of objects is *{book, borrowerAccount, personnelAccount}*, and the set of activities is

[1] The beta release of MISTA can be downloaded at:

http://www.homepages.dsu.edu/dxu/research/MBT.html.

{BorrowBook, ReserveBook, GiveBackBook, AdminActivity, ManageAccess, CreateAccount, Modify Account, DeliverBook, FixBook, ConsultBorrowerAccount}. Table 1 shows the rules specified for the *borrower* role. *day(HD)*, *day(WD)*, and *day(MD)* denote holiday, working day, and maintenance day, respectively. *day(HD)* can also be interpreted as $day(d) \wedge d=HD$, where *d* is a variable. According to rule 1, a borrower is not allowed to give back books on holidays. According to rule 3, a borrower is allowed to borrow books on working days.

In a role hierarchy, each role inherits all rules from its super roles. According to this semantics, we can flatten a role hierarchy. For each primitive role $r \in R$, the set of all defined access control rules with respect to *r,* denoted by $\wp(r)$, includes and only includes the access control rules defined for role *r* and its super roles in $\Re$. If roles are allowed to override the inherited rules, the overriding can also be handled in the flattening process. Therefore, without loss of generality, this paper focuses on the access control rules of primitive roles after the hierarchy is flattened. In LMS, *student*, as a sub-role of *borrower*, inherits all the access control rules in Table 1. Suppose there is no other rule defined with respect to the *student* role in $\Re$. The rules in Table 1 are all the rules defined for *student*, i.e., $\wp(student)$ ={rules 1-6 in Table 1}.

2.2 Characteristics of Access Control Rules

In the following, we formalize several characteristics required of a good access control policy. They provide a basis for building sound test models.

Definition 2 (Consistency). A set of access control rules $\wp$ is said to be consistent if, for any $r \in R$, there do not exist conflicting rules in $\wp(r)$. Two rules for the same role, object, and activity, $<r, o, a, c_1, \tau_1>$ and $<r, o, a, c_2, \tau_2>$, are said to conflict with each other if $\tau_1 \neq \tau_2$ (one of τ_1 and τ_2 is *Permission* and the other is *Prohibition*) and $c_1 \wedge c_2$ is satisfiable (may evaluate to true).

For example, *<student, book, borrow, day(WD), Permission> and <student, book, borrow, true, Prohibition >*are inconsistent. The former implies that student is allowed to borrow books on working days. The latter says that student is prohibited from borrowing books on any day.

Definition 3 (Non-redundancy). A set of access control rules $\wp$ is said to be non-redundant if there do not exist two rules for the same role, object, and activity such that one rule's context subsumes the other rule's context. Formally, there do not exist two rules $<r, o, a, c_1, \tau>$and $<r, o, a, c_2, \tau>$ in $\wp(r)$ such that $c_1 \rightarrow c_2$. (c_1 implies c_2)

For example, *{<student, book, borrow, true, Permission>, <student, book, borrow, day(WD), Permission>}* is redundant because the first rule subsumes the second one.

Definition 4 (Completeness). A set of access control rules $\wp$ is said to be complete if and only if $\wp$ provides an authorization definition for any role, object, activity, and context. Formally, for any $r \in R$, $o \in O$, $a \in A$, $\wp(r)$ must contain one or more rules, say $<r, o, a, c_1, \tau_1>, \ldots, <r, o, a, c_k, \tau_k>(k \geq 1)$, such that $c_1 \vee \ldots \vee c_k = true$ (tautology).

Consider rules 2 and 3 in Table 1. They are the only rules related to activity *BorrowBook* for *student*. Their contexts are *day(HD)* and *day(WD)*. They do not cover maintenance days (*MD*). $day(HD) \vee day(WD)$ is not tautology. In other words, $\neg day(HD) \wedge \neg day(WD)$ is satisfiable: $\neg day(HD) \wedge \neg day(WD) = day(MD)$. Thus, the rules in Table 1 are incomplete.

Consistency, non-redundancy, and completeness can be checked automatically. Dealing with inconsistent and redundant specifications is beyond the scope of this paper. In the following, we discuss how to deal with incomplete rules.

2.3 Dealing with Incomplete Rules

Given a set of access control rules $\wp$, we obtain a complete set of access control rules $\wp'$ as follows.

- We extend the authorization types from *{Permission, Prohibition}* to *{Permission, Prohibition, Undefined}*. "*Undefined*" means that authorization is not defined for the given role, activity, object, and context. We also initialize $\wp'$ as $\wp$.
- For each $r \in R$, $o \in O$, $a \in A$, if there is no such rule $<r, o, a, c, \tau> \in \wp(r)$, then we add rule *<r, o, a, true, Undefined>* to $\wp'(r)$
- For each $r \in R$, $o \in O$, $a \in A$, if $\wp(r)$ contains k ($k \geq 1$) consistent rule(s), $<r, o, a, c_1, \tau_1>, \ldots, <r, o, a, c_k, \tau_k>$, such that $c_1 \vee \ldots \vee c_k$ is not tautology or $\neg c_1 \wedge \ldots \wedge \neg c_k$ is satisfiable, we add rule $<r, o, a, \neg c_1 \wedge \ldots \wedge \neg c_k,$ *Undefined*> to $\wp'(r)$

We can prove that the addition of these new rules does not cause inconsistency or redundancy if $\wp$ is consistent and non-redundant. In LMS, because borrowing books on a maintenance day is not defined, we add rule *<student, Book, BorrowBook, day(MD), Undefined>* to $\wp'(student)$. This is similar for *ReserveBook* and *GiveBackBook*. Therefore, we have rules 7-9 in Table 2. Consider *FixBook* for *student*. $\wp(student)$ does not contain any rule for *FixBook* under any context. So we add rule *<student, Book, FixBook, day(d), Undefined>* to $\wp'(Student)$. Here *day(d)* is true for any $d \in \{HD, WD, MD\}$. This is similar for *DeliverBook*. Therefore we have rules 10-11 in Table 2. Here we apply all activities to each role. It may require a large number of access control rules to complete the specification. To deal with complex models, our approach allows tests to be generated with respect to various coverage criteria and can reduce the search space by using partial ordering and pairwise combination techniques. This will be discussed in Section 4.1.

Table 2. Access control rules added to $\wp'(student)$

No	Object	Activity	Context	Auth_Type
7	Book	BorrowBook	day(MD)	Undefined
8	Book	ReserveBook	day(MD)	Undefined
9	Book	GiveBackBook	day(MD)	Undefined
10	Book	FixBook	day(d)	Undefined
11	Book	DeliverBook	day(d)	Undefined

According to the security design principle "secure by default", a secure system should prohibit the activity from being performed under an unspecified context. If this security principle is followed, the effect of an activity under an unspecified context is similar to prohibition. This paper takes a more general approach - we differentiate the undefined contexts from prohibition contexts so that test models can be independent of implementation choices. In LMS and ASMS, for example, prohibition due to a prohibition context and prohibition due to an undefined context have different effects. The attempt of an activity under an undefined context will

lead to an exception of *UndefinedSecuritPolicyException*, whereas the attempt of a prohibited activity will result in *SecuritPolicyViolationException*.

3. Construction and Analysis of Test Models

In this section, we first give an introduction to the PrT nets used in this work. Then, we describe how to integrate access control rules and contracts into PrT nets. We also discuss how test models can be analyzed through simulation and verification.

3.1 PrT Nets

The PrT nets in this paper, as in the previous work [5][8], are a lightweight version of the original PrT nets [4]. They have both operational and declarative semantics. The operational semantics refers to removal and addition of tokens when transitions are fired. The declarative semantics interprets each transition as a first-order logic formula and transition firing as logical inference. The declarative semantics ensures the correctness of transforming declarative access control rules and contracts (preconditions and post-conditions in first-order logic) into a PrT net. The operational semantics provides a basis for the generation of test sequences (firing sequences) from PrT nets.

A PrT net consists of places (data and conditions), transitions (activities), normal and bidirectional arcs between places and transitions (input and output conditions of activities), inhibitor arcs from places to transitions (negative input conditions), and initial markings (states). A transition can be associated with a guard condition. An arc can be labeled by a list of arguments (constants and variables). If an arc is not labeled, the default label is the zero-argument tuple <>. In Figure 1, *available*, *day,* and *borrowed* are places (circles); *BorrowBook* and *GiveBackBook* are transitions (rectangles). The guard condition of *BorrowBook* is *d=WD*. An arrow (e.g., from *available* to *BorrowBook*) represents a normal arc. A bi-directional arc (arc without arrow) between *n1* and *n2* represents two arcs: one from *n1* to *n2* and the other from *n2* to *n1*. A marking is a set of tokens in all places. A token in *p* is a tuple of constants *<X1, ..., Xn>,* also denoted as *p(X1, ..., Xn)*. The zero-argument tuple is denoted as <>. For token <> in *p*, we also denote it as *p*. We associate a transition with a list of variables as formal parameters, if any. Multiple initial states can be associated with the same net for generating multiple test suites.

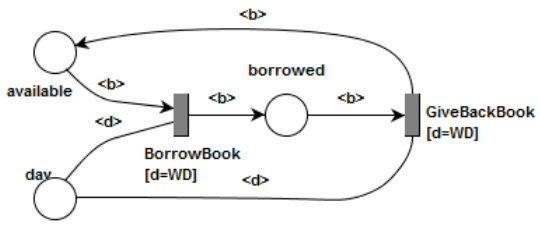

Figure 1. A simple net

Place *p* is called an input (or output) place of transition *t* if there is a normal or bi-directional arc from *p* to *t* (or from *t* to *p)*. *p* is called an inhibitor place if there is an inhibitor arc between *p* and *t*. Let *x/V* be a variable binding (variable *x* is bound to value *V)*. A substitution is a set of variable bindings. For example, *{b/B1, d/WD}* is a substitution where *b* and *d* are bound to *B1* and *WD*, respectively. Let θ be a substitution and *l* be an arc label. l/θ denotes the tuple (or token) obtained by substituting each variable in *l* for its bound value in θ. For instance, if *l* = *<b>* and θ = *{b/B1, d/WD},* then l/θ = *<B1>*

Transition *t* is enabled by substitution θ under a marking if the following conditions are satisfied:

- Each input place *p* has a token that matches l/θ, where *l* is the label of the input arc from *p* to *t*;
- Each inhibitor place *p* has no token that matches l/θ, where *l* is the label of the inhibitor arc between *p* and *t*;
- The guard condition evaluates to true according to θ.

Suppose *{available(B1), day(WD), day(MD)}* is an initial marking for the net in Figure 1. A simple net *BorrowBook* is enabled by θ *={b/B1, d/WD}* because token *<B1>* in the input place *book* matches $<b>/\theta$, token *<WD>* in the input place *day* matches $<d>/\theta$, and the guard *d=WD* is true according to θ. Only enabled transitions can be fired.

Firing an enabled transition *t* with substitution θ under marking M_0^k removes the matching token from each input place and adds new token l/θ to each output place, where *l* is the label of the arc from *t* to the output place. This leads to new marking M_1^k. We denote a firing sequence as M_0^k $[t_1\theta_1> M_1^k$ ... $[t_n\theta_n> M_n^k$, where $t_i(1 \le i \le n)$ is a transition, $\theta_i(1 \le i \le n)$ is the substitution for firing t_i, and M_i^k $(1 \le i \le n)$ is the marking after t_i fires, respectively. A marking *M* is said to be reachable from M_0^k if there is such a firing sequence that transforms M_0^k to *M*.

The PrT nets can be interpreted in terms of logic formulas and inference. Given a net, each input (output) place *p,* together with the associated arc label $<x_1, ...x_n>$ is corresponding to an input (output) predicate $p(x_1, ...x_n)$; each inhibitor place *p,* together with the associated arc label $<x_1, ...x_n>$,is corresponding to a negative predicate $\neg p(x_1, ...x_n)$ (called inhibitor predicate). Each transition can be captured by logic formula $P \rightarrow Q$, where precondition *P* is the conjunction of the inhibitor predicates, input predicates, and guard condition, and post-condition *Q* is the conjunction of the output predicates and negation of each input predicate. *P* and *Q* are universally quantified. For *BorrowBook* in Figure 1, P= *available (b)*$\wedge$ *day(d)* $\wedge d=WD$ and $Q=\neg$*available(b)*$\wedge$ *borrowed(b)*$\wedge$*day(d)*. This lays a theoretical foundation for the transformation of access control rules and contracts.

3.2 Construction of Test Models

Contracts are in the form of *precondition* $\rightarrow$ *post-condition*. Suppose *available(b)* means book *x* is available and *borrowed(b)* means book *b* is borrowed. The contract of *GiveBackBook(b)* is for *any b*, *borrowed(b)* $\rightarrow$ *available(b)*. An activity can be associated with multiple contracts, representing different situations. A general precondition in the disjunctive form $P_1 \vee ... \vee P_n$ can be represented by multiple contracts $P_1 \rightarrow Q_1$..., $P_n \rightarrow Q_n$. For example, the contract of *BorrowBook(b)* is for any *b, available(b)* $\rightarrow$ *borrowed(b)*$\wedge$ $\neg$*available(b)* or for any *b, reserved(b)* $\rightarrow$ *borrowed(b)*$\wedge$ $\neg$*reserved(b),* where *reserved(b)* means book *b* is reserved. In this paper, the preconditions and post-conditions are not necessarily accurate specifications of activity's semantics. They may represent ordering constraints for testing the activities involved in access control rules.

The process for transforming access control rules together with the contracts of relevant activities is as follows. First, we partition the complete rule set $\wp'$ into a number of subsets in terms of roles

and relevant activities so that a PrT net will be constructed for each subset. In LMS, *student* and *teacher* are independent roles although they have similar activities. So we group the access control rules for *student* and *teacher* into different subsets. Second, each subset together with the contracts of the relevant activities is integrated into a PrT net. This is done by converting each rule and the contract of the corresponding activity into a net and composing the nets of all rules into a single net. Third, we define test data and system settings as initial markings of the PrT net so that it can be analyzed for correctness and then used for test generation.

Suppose the access control rules with respect to activity $a \in A$ are $<r, o, a, c_1, \tau_1>, <r, o, a, c_2, \tau_2>, \ldots, <r, o, a, c_m, \tau_m>$ and $p_1(x_1) \wedge \ldots \wedge p_n(x_n) \rightarrow q_1(y_1) \wedge \ldots \wedge q_k(y_k) \wedge \neg p_1(x_1) \ldots$ is a contract of activity a. Here $m>0$ because $\wp'(r)$ is a complete set of access control rules. We handle each rule $<r, o, a, c_i=r_1 \wedge \ldots \wedge r_u, \tau_i>$ $(1 \le i \le m)$ as follows:

- If τ_i=*Permission,* we first convert the contract into a net with one transition named after activity a. Generally, predicates $p_1(x_1), \ldots, p_n(x_n)$ in the precondition are corresponding to the input places of the transition if they are not built-in functions such as arithmetic and relational operations (e.g., $z=x+y$ and $x>y$). Built-in predicates are transformed into part of the transition's guard condition. Predicates $q_1(y_1), \ldots, q_k(y_k)$ in the post-condition are corresponding to the output places of the transition. The input/output arcs are labeled by the arguments of the corresponding predicates. The input arc for p_j is bi-directional if its negation $\neg p_j$ does not appear in the post-condition. As the context in an access control rule is an additional precondition of the activity in the rule, the predicates $r_1, \ldots, r_u$ in the context lead to additional input places for the transition. The arc labels depend on the corresponding arguments. If $r_i(z_i)$ does not have negation and z_i is a variable, then the arc label is $<z_i>$. If $r_i(Z_i)$ does not have negation and Z_i is a constant, then the arc label is $<z_i>$, and $z_i=Z_i$ is added to the guard condition of the transition[2]. If $r_i(Z_i)$ is a negative predicate and Z_i is a constant, then the arc label is $<z_i>$, and $z_i \neq Z_i$ is added to the guard condition of the transition. The arcs are bi-directional unless the activity negates the context. Figure 2 shows the net, where the arc between p_1 and a is directed because $p_1(x_1)$ is negated in the post-condition; the arc between p_n and a is bi-directional as we assume that $\neg p_n(x_n)$ does not appear in the post-condition.

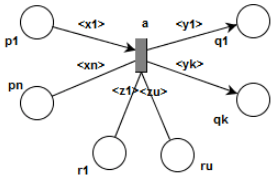

Figure 2. PrT net for a permission rule

- If τ_i= *Prohibition,* we convert the precondition of the contract into a net with one transition named *Pa* ("*P*" denotes "prohibition"). The post-condition of the contract is not used because the activity is prohibited. The predicates in the precondition are corresponding to input places and the arcs are labeled by the corresponding arguments. The arcs are all bidirectional because, when the prohibited activity is attempted under the specified context, it should not change the system's state. The context is handled in the same way as τ_i=*Prohibition*. Figure 3 shows the net.

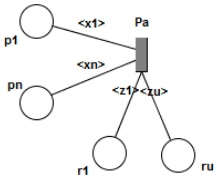

Figure 3. PrT net for a prohibition rule

- If τ_i= *Undefined,* the transformation is similar to that for τ_i= *Prohibition.* The only difference is that the transition is named as *Ua ("U"* denotes *"Undefined").*

Consider rules 2, 3, and 7 in Tables 1 and 2. For contract *available(b)*→*borrowed(b)*∧¬*available(b)* of *BorrowBook,* we transform rule 2, together with the contract, into transition *BorrowBook*, as shown in Figure 4(A). Its input places are *available* (resulted from the precondition) and *day* (resulted from the context in rule 2), its output places are *borrowed* (resulted from the post-condition) and *day* (resulted from the context in rule 2), and its guard condition is *d=WD* (resulted from the context in rule 2). Then we transform rule 3, together with the precondition of the contract, into transition *PBorrowBook*. Its input and output places are *available* (resulted from the precondition) and *day* (resulted from the context in rule 3) and its guard condition is *d=HD*. Likewise, we transform rule 7 into transition *UBorrowBook*.

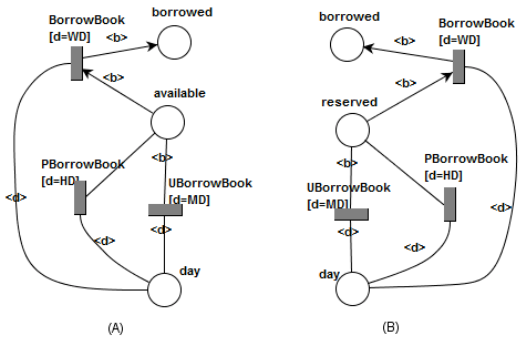

Figure 4. Composition of nets

Similarly, *reserved(b)*→*borrowed(b)*∧¬*reserved(b),* with rules 2, 3, and 7, can be transformed into the net in Figure 4 (B). It is the same as (A) except that *available* is replaced by *reserved*. We compose multiple nets into one net through place fusion - places with the same name in different nets become one place in the composed net. However, transitions with the same name in different nets become different transitions in the composed net (each of them is assigned a unique internal identity). In Figure 4 (A) and (B) share places *borrowed* and *day* after they are composed. The resultant net can further be composed with the nets obtained from other access control rules and contracts. Figure 5 shows the net that covers all the rules in Tables 1 and 2. For

[2] $r_i(Z_i)$ is equivalent to $r_i(z_i) \wedge z_i = Z_i$. Alternatively, $<Z_i>$ can be used directly as the arc label, with no change to the guard condition.

clarity, an annotation is used to specify *day* as a global predicate, meaning that there is a bidirectional arc between *day* and each transition

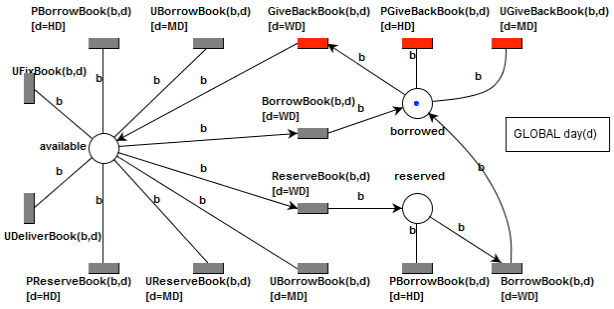

Figure 5. Access control model of the student role

According to the semantics of the PrT nets, we can prove that the above transformations have preserved the semantics of contracts and access control rules. For the sake of simplicity, the above discussion focuses on rules for individual roles. We can build a test model that involves multiple roles. The transformation of access control rules into a net essentially depends on the given subset of access control rules and the contracts of involved activities. If the given rules involve different roles, then the net captures the behaviors of different roles. In this case, we enhance the net with a new place, named *role*, new arcs from place *role* to each transition labeled with <*r*>, and additional guard condition for each transition (e.g., *r=Student*). In essence, this is to use *role(r)* as an additional precondition for each activity. In ASMS, for example, the complete auction process involves various activities (e.g., creation of a sale, change of the state of a sale for auction, comments and bids by buyers) performed by different roles (e.g., *seller*, *admin*, and *buyer*). We can build the test models based on the auction process, rather than individual roles.

3.3 Analysis of Test Models

After the structure of a PrT net is constructed, we define its initial markings by specifying test data (e.g., actual arguments of the activities) and test configurations (e.g., system settings and contexts in the access control rules). Consider the net in Figure 5. Let M_0={m_0}, where m_0={*available(B1), day (WD), day(HD), day(MD)*}. The net and M_0 form a test model for the *student* role. In m_0, test data *available(B1)* can reach the activities of *BorrowBook*, *ReserveBook* and *GivebackBook*. *day(WD), day(HD),* and *day(MD)* represent all possible contexts in the rules so that the test model can cover all the contexts.

Our approach provides three techniques for analyzing and debugging the specifications of test models – verification of transition reachability, verification of state reachability, and model simulation. In a test model, each transition is corresponding to an access control rule under a certain condition of the involved activity. Thus, all transitions should be reachable from some initial state. If there is one transition that is unreachable from the given initial states, then the transition will not be covered by any tests to be generated from the specified test model. In this case, either the net or the initial states is specified incorrectly. Suppose M_0={m_1}, where m_1= {*available(B1), day(WD), day(MD)*}. *UBorrowBook* is not reachable from m_1 in the net in Figure 5. In this case, M_0 is not specified properly.

If a goal state is known to be reachable (or unreachable), but the verification reports that it is unreachable (or reachable), then the net or the initial states is specified incorrectly. For example, {*reserved(B1)*} is a state reachable from m_0={*available(B1), day(WD), day(MD), day(HD)*}.It can be reached by transition firings *ReserveBook(b/B1, d/WD)*. However, if the arc from transition *ReserveBook* to place *reserved* is missing, the verification would report that the above state is not reachable.

Our approach also provides an animator for stepwise simulation of test models. At each state, the animator shows the number of tokens in each place and highlights the enabled transitions. The user can choose to manually fire one enabled transition at a time or continuously fire randomly selected enabled transitions. This can help find out whether the expected behaviors are specified correctly in a test model. Suppose M_0={m_1}, where m_1= {*available(B1), day(WD), day(MD), day(HD)*} for the net in Figure 5. We choose to fire *BorrowBook(b/B1, d/WD)* under m_1, which results in ={*borrowed(B1), day(WD), day(MD), day(HD)*}. At this state, the animator should highlight three enabled transitions: *GiveBackBook*, *PGiveBackBook*, and *UGiveBackBook*. If any one of them is not highlighted, we can check if the transition's associated arcs are described correctly.

4. Generation of Access Control Tests

In this section, we first describe how MISTA generates model-level access control tests from a test model. Then we discuss how to create a MIM specification so that executable test code can be generated.

4.1 From Transition Firings to Access Control Tests

Definition 5 (Model-level access control test). Given an access control test model represented by a PrT net, a test case is a firing sequence $< M_0^k [t_1\theta_1> M_1^k, \ldots, [t_n\theta_n> M_n^k >$ in the PrT net, where

- M_0^k is the initial setting of the test,
- Transition firings $t_1\theta_1,\ldots,t_n\theta_n$ are ***test inputs***, i.e., calls to the activities in access control rules. Suppose transition t_i is corresponding to activity $a(x_1,\ldots,x_m)$ and substitution θ_i={x_1/u_1, ..., x_m/u_m}. Then $t_i\theta_i$ ($1 \le i \le n$) represents component call $a(u_1,\ldots,u_m)$, where u_j ($1 \le j \le m$) is x_j' actual argument.
- $M_1^k,\ldots,M_n^k$ are ***test oracles*** for respective test inputs $t_i\theta_i$ ($1 \le i \le n$). For each place $p \in P$ and each token $<v_1,\ldots,v_m> \in M_i^k(p)$, proposition $p(v_1,\ldots,v_m)$, when used as an ***oracle value***, is expected to evaluate to true in the SUT.

For example, m_0, *Reserve(b/B1, d/WD)*, m_1, *Borrow (b/B1, d/WD)*, m_2, *UGiveBackBook(b/B1, d/HD)*, m_3 is a firing sequence in the test model in Figure 5. Access control model of the student role where M_0={m_0} and m_0={*available(B1), day(WD), day(HD), day(MD)*}. Thus:

m_1={*reserved(B1), day(WD), day(HD), day(MD)*},

m_2={*borrowed(B1), day(WD), day(HD), day(MD)*}

m_3={*borrowed(B1), day(WD), day(HD), day(MD)*}

The firing sequence is a test case that exercises three access control rules: reserve books on working days (permitted), borrow books on working days (permitted), and give back books on holidays (prohibited). The states of book *B1*, *reserved(B1), borrowed(B1),* and *borrowed(B1)*, represent the expected results

of these activities. We assume that a prohibited activity, such as *PGiveBackBook(b/B1, day/HD)*, should not change the system state. Here *day(WD), day(HD), day(MD)* are not used as test oracles because they represent different system settings for access control contexts.

Therefore, test generation from a test model in our approach is to produce firing sequences from the test model according to a certain strategy (e.g., to achieve a coverage criterion). MISTA supports automated test generation for several coverage criteria, such as reachability tree coverage, state coverage, and transition coverage. In an access control test model, a transition is corresponding to one access control rule. A test suite is said to meet transition coverage if each transition is covered by at least one test. A test suite is said to meet state coverage if each state is covered by at least one test. A test suite is said to meet reachability tree coverage if each edge in the reachability graph (i.e., each transition firing under each reachable marking) is covered by at least one test. Reachability tree coverage subsumes transition coverage and state coverage because the reachability tree includes each reachable transition and each reachable state. The case studies in this paper use the reachability tree coverage.

In MISTA, test cases are structured as a test tree, where each path from an initial marking to a leaf is corresponding to a firing sequence (i.e., test case). Figure 6 shows portion of the test tree generated for the reachability tree coverage of the test model in Figure 5. Node "*1 new*" represents the initial marking, i.e., the initial setting of each test. The path *1→1.1→1.1.2* exercises two access control rules. It first borrows book *B1* on a working day, which should be permitted, and attempts to return the book on a holiday, which should be prohibited.

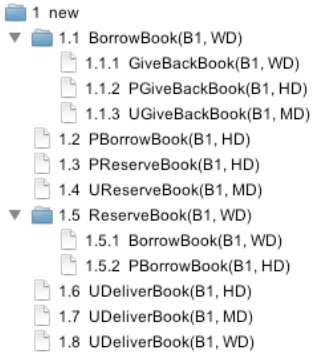

Figure 6. Portion of a test tree

In order to represent tests generated from multiple initial markings that represent different sets of test data and system settings, the test tree uses an invisible root node whose child nodes are corresponding to the initial markings. The test tree for reachability tree coverage is constructed as follows. The nodes of initial markings are put into a stack (if depth-first search is used) or queue (if breadth-first search is used) for expansion. When a node is expanded, all possible transition firings (including all substitutions for each transition) under the current marking are computed and a child node is created for each possible firing. The child node will also be expanded if the new marking has not expanded before. Due to the combinatorial nature of transition firings, the test tree for a complex model may have a large number of tests. MISTA provides two effective techniques for reducing the number of tests: partial ordering and pairwise combination. The total ordering of *n (n>1)* independent or concurrent transition firings yield *n!* sequences, where the partial ordering only produces one sequence. When there are more than two inputs, the pairwise technique covers all pairs of inputs, rather than all combinations of inputs. Suppose each of 10 input variables has 10 values. There are 10^{10} combinations of these variables. In MISTA, however, 120 combinations can cover all pairs of the variables.

4.2 Building MIM Specification for Test Code Generation

The model-level tests in *Definition 5* are not executable because they are generated solely from a given test model, which can be independent of the SUT. The test model of *student* does not specify how *BorrowBook* can be performed against the SUT. In our approach, a MIM specification for a test model can be created so that all model-level tests can be converted into test code automatically. A MIM specification maps the elements in a test model into corresponding constructs in the SUT. It consists of the following main components: object function f_o, method function f_c, accessor function f_o, mutator function f_m, a list of setting predicates l_s, and helper code function f_h. Table 3 presents an example of these components in LMS.

Object function f_o maps objects in the test model to objects in the SUT. In LMS, book *B1* in the test model of *student* is corresponding to a named constant, referring to a book titled "Software Security". Method function f_c maps activities in the test model to test operations in the SUT. For example, the implementation of *BorrowBook* is a method *doPermittedBorrow*. Accessor functionf_a maps predicates in the test model to accessors in the SUT. It is used for verifying oracle values. For example, book *b* is borrowed on day *d* in a test case, i.e., *borrowed(b, d)*, can be verified by method *isBookBorrowed(b)*. Mutator function f_a maps the system setting predicates in l_s to operations in SUT so that the SUT can be configured to a specific state. For example, predicate *day* in LMS is a system setting. As an access control precondition, it must be set correctly because the individual activities can be called. Setting LMS to a working day, i.e., making *day(WD)* true, can be done by the following statement: *ContextManager.currentContext=Context Manager. workingday;* Helper code function f_h includes header code (e.g., package and import statements in Java), constant and variable declarations, setup, teardown, and methods for testing individual activities. All of this code will be included in the test code. The methods for testing individual activities depend on how the SUT is implemented, e.g., what types of security exceptions will be reported. In the case studies, the exceptions for prohibited activities and undefined activities are *SecuritPolicyViolationException* and *UndefinedSecuritPolicyException*, respectively. Thus, a test for a permitted activity fails if the SUT throws an exception of *SecuritPolicyViolationException* or *UndefinedSecurit PolicyException*. A test for a prohibited activity fails if no exception is thrown or the thrown exception is not *SecuritPolicyViolationException*. A test for an undefined activity fails if no exception is thrown or the thrown exception is not *UndefinedSecuritPolicyException*.

Table 3. Sample MIM specification

MIM	Model element	Implement ation element	Notes
f_o	B1	Book1Title	Book1Title is a named constant in the helper code

f_c	BorrowBook(b,d)	doPermitedBorrow(b)	doPermittedBorrow is a test method in the helper code. It fails if an exception is thrown.
f_o	borrowed(b,d)	isBookBorrowed(b)	isBookBorrowed is a query method for verifying whether the status of the book is borrowed by the borrower
f_m	day(WD)	ContextManager.currentContext = ContextManager.workingday;	It sets the concurrent context to working day.
l_s	day		day is a system setting
f_h(PACKAGE)	package com.library.test.software.modeltest;		Helper code for the package statement of Java test code
f_h(CODE)	private final String Book1Title = "Software security"; …		Declarations and methods to be included in the test code

Given a complete MID specification (i.e., test model and MIM), the executable test code for all model-level tests can be generated automatically. Each model-level test is corresponding to a test method. For the aforementioned test: m_0, *ReserveBook(b/B1, d/WD),* m_1, *BorrowBook(b/B1, d/WD),* m_2, *PGiveBackBook(b/B1, d/HD),* m_3, the Java test method is as follows:

```
public void test12() throws exception {
  System.out.println("Test 12");
  ContextManager.currentContext=ContextManager.workingday;
  doPermittedReserve(Book1Title);
  assertTrue(isBookReserved(Book1Title));
  ContextManager.currentContext = ContextManager.workingday;
  doPermittedBorrow(Book1Title);
  assertTrue(isBookBorrowed(Book1Title));
  ContextManager.currentContext = ContextManager.holiday;
  doProhibitedGiveBack(Book1Title);
  assertTrue(isBookBorrowed(Book1Title));
}
```

5. EMPIRICAL STUDIES

5.1 Experiment Setup

Our case studies are based on two Java programs, LMS and ASMS [9][10]. Table 4 presents the main parameters, where *R* is the number of roles, *PR* is the number of primitive roles, *O* is the number of objects, *A* is the number of activities, and *RL* is the total number of specified rules for the primitive roles. The mutants of the access control rules were created by the MutaX tool[3] using five types of mutation operators [9][10]: replacing permission rule with prohibition, replacing prohibition rule with permission, changing role, changing context, and adding a rule. They were created before this work was initiated. To evaluate the proposed approach, the following mutants were excluded:

- Mutants related to non-implemented activities are not used because the tests could not be performed.
- Mutants with inconsistent access control rules are not used because our approach assumes that the given access control rules are consistent. These mutants are typically created by the operator that adds new rules..
- Equivalent mutants, which have the same behavior as the original version because of the implementation issues in the original version (e.g., some access control contexts, hard-coded in the implementation, are not affected by mutation of access control rules).

[3]https://sites.google.com/site/servalteam/tools/mutax

Table 4. Subjects in the empirical studies

	LOC	Classes/ Methods	R	PR	O	A	RL
LMS	3,204	62/335	7	5	4	12	33
ASMS	10,703	134/797	8	6	6	23	107

The protocol of our experiment is as follows. First, we specify the contracts of the activities involved in access control rules, and construct and analyze the test models. Second, we create the MIM specification for each test model as described in Section 4. Thus complete MID specifications are obtained for test code generation. Third, we use MISTA to generate test code from the MID specifications. Fourth, we execute the generated test code against the original version such that no test fails (the original version is considered as the correct version). If there is a failure, then the previous steps need to be repeated. Finally, we run the test code against each mutant.

5.2 Results

The results of our experiments are summarized in Table 5, where *T* is the total number of transitions, *P* is the total number of places (they reflect the complexity of test models), *TC* is the number of test cases, *LOC* is the number of lines of code generated, *M* is the total number of mutants, *K* is the number of mutants killed by the test, and *FDR* is the fault detection rate (i.e., number of mutants killed/total mutants tested also called mutation score).

Table 5. Results of the empirical studies

	Models		Tests		Mutation Analysis		
	T	***P***	***TC***	***LOC***	***M***	***K***	***FDR***
LMS	73	27	207	3,086	243	233	95.9%
ASMS	126	30	179	4,680	914	914	100%

For LMS, 207 test cases in 3,086 lines of non-comment code were generated. They killed 233 out of 243 mutants, with an overall detection rate of 95.9%. The 10 remaining mutants not killed by the tests have the same nature – they contain a new rule created by the adding-rule operator but can never cause security problems because the functional precondition of the activity in the added rule is not satisfiable. These mutants do not violate the required security policies. Consider a mutant with the following added rule that allows the *admin* role to return books on any day: *(admin, Book, GiveBackBook, true, Permission)*. According to the required access control policies, none of the *Borrower*'s activities, *BorrowBook*, *ReserveBook*, and *GiveBackBook*, is intended for use by the *admin* role (no access control rules with respect to these activities are specified for *admin*). The above rule can never enable the *admin* role to return books because the precondition of *GiveBackBook*- "the book is borrowed" (by the same person) - is unsatisfiable. This precondition can only be fulfilled by *BorrowBook*. In the mutant, however, *Admin* is not able to borrow

books (*BorrowBook* is undefined for *admin*). It is worth pointing out that our approach killed the mutant with the following added rule that allows *admin* to borrow books: *(admin, Book, BorrowBook, true, Permission)*. In ASMS, 179 tests in 4,680 lines of code were generated. They killed all of the 914 mutants.

There are two main reasons for the near-perfect mutation scores in our case studies. First, our approach for dealing with incomplete access control specification makes it possible to reveal all undefined situations. The tests generated to cover these situations are not only necessary but also powerful for revealing potential policy violations in an implementation. Second, the tests generated for the reachability graph coverage can cover all access controls, objects, activities, and contexts. This was feasible thanks to the automation of both test generation and test execution. In comparison, transition coverage (or rule coverage) has a low fault detection capability. As an initial experiment, the application of transition coverage to the *student* role in LMS only killed about 50% of the mutants. The reason was that many objects and contexts were not exercised. Thus, we decided not to continue the empirical evaluation of the transition coverage.

5.3 Threats to Validity

The main result of the case studies is that our approach is highly effective in detecting violations of access control rules. The key aspects that have led to this result include the access control model, the formalization of contracts, generation of access control tests with the reachability tree coverage, generation of executable test code, and mutation analysis of access control rules. In the following, we discuss how these aspects can be affected when our approach is applied to real-world software where access control is an important security mechanism.

First, our approach focuses on testing of access control rules. It does not cover every aspect of RBAC, such as role assignment and session management. Second, due to the small sizes of the subject programs, we were able to generate tests to cover all rules and combinations of objects, resources, and contexts. Generation of such comprehensive tests may not be feasible for complex or large-scale software systems where access control is involved in very large state space. In order to deal with such systems, testers may try to divide the system to several small components/modules and apply our approach to each one in an independent way. This will reduce the complexity and the testing effort. Since our technique was not applied to such systems, we cannot evaluate the effectiveness of this strategy. Third, we were able to generate executable test code by completing the MIM specifications of the test models. In the MIM specifications, invocations to individual activities and verification of test oracles are programmed. For real-world software, the individual activity tests and the test oracles may not be completely programmable. Fourth, although our approach is applicable to a variety of languages and applications (Java, C, C++, C#, VB, HTML/Selenium IDE) supported by MISTA, the subject programs were limited to Java applications. Finally, the evaluation of fault detection capability is based on the mutation analysis of access control rules. Although the mutants were created by five different operators, they do not necessarily represent all possible access control flaws in real-world software.

6. RELATED WORK

Software security testing involves two different perspectives - testing of security policies and performing security attacks (penetration testing) to identify vulnerabilities. This paper focuses on policy testing.

In recent years, Le Traon's group have investigated various issues of testing access control policies, such as test criteria of access control policies [9], test generation from access control models [10], mutation analysis of access control policies [12], and selection and transformation of functional tests for policy testing [13]. In the above work, test generation was not automated. Pretschner, in collaboration with Le Traon's group, has proposed a model-based approach [14], where the access control model consists of a role hierarchy, a permission hierarchy, and a context hierarchy. They use a combinatorial testing technique to derive test targets, i.e., combinations of roles, permissions, and contexts. Each test target is relevant to one access control rule. In comparison, this paper focuses on automatic test code generation from models that capture the interactions of access control rules. A test is usually a sequence of access control requests for exercising multiple access control rules.

Masood et al. [15][16] have investigated a state-based approach to test generation for RBAC policies. They first construct a finite state machine (FSM) of the RBAC policy and then derive tests from the FSM. This model essentially captures the behaviors of role assignment, rather than access control rules. Different from FSMs, PrT nets can capture both control flows and data flows (e.g., test data and contexts). Based on the Assurance Management Framework (AMF), Hu and Ahn have proposed an approach to the generation of conformance tests of access control policies through constraint verification [17]. Test cases are derived through verification by either removing or negating the security constraints. Our approach focuses on test generation with respect to coverage criteria and transformation of model-level tests into executable code. Mallouli et al. proposed a model-based approach for integrating OrBAC (Organizational Based Access Control) rules into an initial functional model represented by an extended finite state machine [18]. Test sequences generated from the integrated model will be able to exercise the OrBAC rules. Different from this approach, we do not assume the availability of the full functional model. We integrate the access control rules with the contracts of the activities involved in the given rules. This integration can handle incomplete specification of access control rules. Jürjens has developed an approach for testing security-critical systems based on UMLsec models [19]. Test sequences for access control properties are generated from UMLsec models to test the implementation for vulnerabilities. Li et al. proposed an approach to test generation from security policies specified as OrBAC rules [20]. It focuses on generation of test purposes from individual OrBAC rules. In comparison to these two approaches, our work integrates access control rules into an operational model and generates tests to cover different access control rules. Julliand et al. have proposed an approach to generating security tests in addition to functional tests by re-using the functional test model together with a new model of security properties defined by a security engineer [21]. The security properties describe tortuous situations that could violate security policies. They did not use an explicit access control model. Different from this work, our approach not only provides a process for building the model of access control rules, but also generates tests to exercise all access control rules and contexts. Generally, the above model-based approaches focus on generating model-level tests, not executable tests. Our approach can produce executable test code by using a flexible mechanism that maps modeling elements into implementation constructs.

Martin et al. have investigated techniques for test generation from access control policy specifications written in XACML [22][23]. They have defined policy coverage criteria and developed a

mutation-testing framework for XACML policies. To generate tests from policy specifications, they synthesize inputs to a change-impact analysis tool. Different from this work, our approach does not focus specifically on XACML and targets both PDP (policy decision point, where access control decisions are made) and the enforcement mechanisms inside the system.

7. CONCLUSIONS

We have presented a new model-based approach for automated testing of access control policies. It provides a tool-supported process for building access control test models from contracts and access control rules. Access control tests can then be generated and converted into executable code based on a MIM specification that maps the modeling elements into implementation constructs. By using mutation analysis of access control implementation, our empirical studies have demonstrated that our approach is highly effective in detecting violations of access control policies.

In the case studies, tests were generated using the reachability tree coverage, which subsumes both transition coverage and state coverage. In addition, this paper has focused on access control rules, which define role and permission authorization. When a role is involved in a test case, a subject is created and assigned to that role. Our future work will deal with other RBAC features, such as role assignment and session management.

8. ACKNOWLEDGMENTS

This work was supported in part by NSF under grants CNS 1004843 and CNS1123220.

9. REFERENCES

[1] Pretschner, A., Prenninger, W., Wagner, S., Kühnel, C., Baumgartner, M., Sostawa, B., Zölch, R. and Stauner, T. 2005. One evaluation of model-based testing and its automation. In *Proc. of the 27th International Conf. on Software Engineering (ICSE'05)*, 392-401.

[2] Pretschner, A., Slotosch, O., Aiglstorfer, E. and Kriebel, S. 2004. Model-based testing for real - The inhouse card case study. *J. Software Tools for Technology Transfer* 5(2-3): 140-157.

[3] Zander, J., Schiefewrdecker, I., and Mosterman, P. J. (eds.). 2011. *Model-Based Testing for Embedded Systems*, CRC Press.

[4] Genrich, H.J. 1987. Predicate/transition nets. *Petri Nets: Central Models and Their Properties*, 207–247.

[5] Xu, D. and Nygard, K.E. 2006. Threat-driven modeling and verification of secure software using aspect-oriented Petri nets, *IEEE Trans. on Software Engineering*, vol. 32, no. 4, 265-278.

[6] Xu, D. 2011. A tool for automated test code generation from high-level Petri nets. In *Proc. of Petri Nets'11*, LNCS 6709, 308-317, Newcastle upon Tyne, UK, June 2011.

[7] Meyer, B. 1997. *Object-Oriented Software Construction*, 2nd Edition, Prentice-Hall PTR.

[8] Xu, D., Tu, M., Sanford, M., Thomas, L., Woodraska, D., and Xu, W. 2012. Automated security test generation with formal threat models. *IEEE Trans. on Dependable and Secure Computing*. In press.

[9] Le Traon, Y., Mouelhi, T., Pretschner, A., and Baudry, B. 2008. Test-driven assessment of access control in legacy applications. In *Proc. of the First IEEE International Conference on Software, Testing, Verification and Validation (ICST'08)*, Norway, 238-247.

[10] Mouelhi, T., Fleurey, F., Baudry, B., and Le Traon, Y. 2008. A model-based framework for security policy specification, deployment and testing. In *Proc. of the ACM/IEEE 11th International Conf. on Model Driven Engineering Languages and Systems (MODELS'08)*, Toulouse, France.

[11] Jia, Y. and Harman, M. 2010. An analysis and survey of the development of mutation testing. *IEEE Trans. on Software Engineering*, vol. 37, no. 5, 649-678.

[12] Le Traon, Y., Mouelhi, T., and Baudry, B. 2007. Testing security policies: going beyond functional testing. In *Proc. of the IEEE International Symposium on Software Reliability Engineering (ISSRE'07)*, Sweden.

[13] Mouelhi, T., Le Traon, Y., and Baudry, B. 2009. Transforming and selecting functional test cases for security policy testing. In *Proc. of the Second International Conf. on Software Testing Verification and Validation (ICST'09)*. Denver, USA.

[14] Pretschner, A. Le Traon, Y., and Mouelhi, T. 2008. Model-based tests for access control policies. In *Proc. of the First IEEE International Conference on Software, Testing, Verification and Validation (ICST'08)*. Norway.

[15] Masood, A. Bhatti, R., Ghafoor, A., Mathur, A. 2009. Scalable and effective test generation for role-based access control systems. *IEEE Trans. on Software Engineering,* vol. 35, no. 5, 654-668.

[16] Masood, A., Ghafoor, A., Mathur, A. 2010. Conformance testing of temporal role-based access control systems. *IEEE Trans. on Dependable and Secure Computing*, vol. 7, no. 2, 144-158.

[17] Hu, H. and Ahn, G. 2008. Enabling verification and conformance testing for access control model. In *Proc. of the 13th ACM Symposium on Access Control Models and Technologies (SACMAT'08)*, 195–204.

[18] Mallouli, W., Orset, J.M., Cavalli, A., Cuppens, N., Cuppens, F. 2007. A formal approach for testing security rules. In *Proc. of the 12th ACM Symposium on Access Control Models and Technologies (SACMAT'07),* 127-132.

[19] J. Jürjens, 2008. Model-based security testing using UMLsec. *Electronic Notes in Theoretical Computer Science (ENTCS)*, 220(1): 93-104.

[20] Li, K., Mounier, L., Groz, R. 2007. Test generation from security policies specified in Or-BAC. In *Proc. of the 31st Computer Software and Applications Conference (COMPSAC'07)*, 255-260.

[21] Julliand, J., Masson, P.A., Tissot, R. 2008. Generating security tests in addition to functional tests. In *Proc. of the Workshop on Automation of Software Test (AST'08)*, 41–44.

[22] Martin, E. and Xie, T. 2006. Defining and measuring policy coverage in testing access control policies. In *Proc. of the 8th International Conference on Information and Communications Security*, 139-158.

[23] Martin, E. and Xie, T. 2007. A fault model and mutation testing of access control policies. In *Proc. of WWW'07*, 667-676.