Automated Model-Based Testing of Role-Based Access Control Using Predicate/Transition Nets

Dianxiang Xu, *Senior Member, IEEE*, Michael Kent, Lijo Thomas, Tejeddine Mouelhi, and Yves Le Traon

Abstract—Role-based access control is an important access control method for securing computer systems. A role-based access control policy can be implemented incorrectly due to various reasons, such as programming errors. Defects in the implementation may lead to unauthorized access and security breaches. To reveal access control defects, this paper presents a model-based approach to automated generation of executable access control tests using predicate/transition nets. Role-permission test models are built by integrating declarative access control rules with functional test models or contracts (preconditions and postconditions) of the associated activities (the system functions). The access control tests are generated automatically from the test models to exercise the interactions of access control activities. They are transformed into executable code through a model-implementation mapping that maps 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. The full model-based testing process has been applied to three systems implemented in Java. The effectiveness is evaluated through mutation analysis of role-based access control rules. The experiments show that the model-based approach is highly effective in detecting the seeded access control defects.

Index Terms—Access controls, security and privacy protection, testing tools, test design

1 Introduction

Role-based access control (RBAC) [3], [22] is a popular access control method for restricting system access to authorized users. It assigns users to specific roles and grant permissions to each role according to the role's job responsibilities. According to a 2010 report [18] on the economic values of RBAC, the benefits of RBAC include more efficient provisioning, more efficient policy administration in an era of increased regulation of internal controls, enhanced security and integrity, and enhanced organizational productivity. An RBAC policy consists of a set of declarative rules, defining which role is allowed to access what resources under which conditions. A correctly specified RBAC policy may be implemented incorrectly for various reasons, such as programming errors, omissions, misunderstanding of the requirements, and intricate interplay between business logic and access control policy. The defects in an incorrect implementation may result in serious security problems, such as unauthorized accesses and escalation of privileges. Therefore, it is important to reveal the potential discrepancy between the RBAC specification and the actual implementation.

To reveal access control defects in a system implementation, existing approaches to RBAC testing often focus on devising test cases with respect to individual RBAC rules. The main issue of testing individual rules, however, is that it cannot see the forest for the trees because access control activities are often interrelated to each other. In a library management system (LMS), for example, access control rules may be defined for such activities as borrow and return books, where a precondition of returning a book is that there is a borrowed book. It is difficult to cover all the interactions among access control activities by testing individual rules. Testing the individual borrow and return rules would also lead to duplicated tests—testing the return activity typically involves a borrow activity. In addition, it is important to test the way the system is able to correctly update the status of the objects. For instance, a book can be reserved, borrowed or returned. Updating object states correctly is crucial to triggering the appropriate access control rules. This task could only be performed by running particular testing scenarios that exercise interactions between the business logic and the access control mechanisms.

In this paper, we take benefit from the high-level nature of an RBAC policy to express it into a productive model aligned with functional requirements. This makes it feasible to apply a model-based approach to automated testing of RBAC policy. Model-based testing uses models of a system under test (SUT) for generating test cases [25]. It is an appealing approach to testing because of several potential benefits [21]. First, the modeling activity helps clarify test requirements and enhances communication between developers and testers. 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. Nevertheless, studies have shown that the tester's ability to build quality models and the

- *D. Xu is with the Department of Computer Science, Boise State University, Boise, ID 83725. E-mail: dianxiangxu@boisestate.edu.*
- *M. Kent is with the SDN Communications, Sioux Falls, SD 57104. E-mail: Michael.Kent@sdncommunications.com.*
- *L. Thomas is with the Cognizant Technology Solutions, Teaneck, NJ 07666. E-mail: llijo.thomas@cognizant.com.*
- *T. Mouelhi is with the itrust Consulting, Niederanven, Luxembourg. E-mail: mouelhi@itrust.lu.*
- *Y. Le Traon is with the Interdisciplinary Centre for Security, Reliability and Trust University of Luxembourg, Campus Kirchberg, L-1359, Luxembourg. E-mail: yves.leTraon@uni.lu.*

Manuscript received 22 Oct. 2013; revised 17 Sept. 2014; accepted 14 Nov. 2014. Date of publication 0 . 0000; date of current version 0 . 0000.
Recommended for acceptance by H. Jin.
For information on obtaining reprints of this article, please send e-mail to: reprints.org, and reference the Digital Object Identifier below.
Digital Object Identifier no. 10.1109/TC.2014.2375189

required expertise in rigorous modeling are major barriers to the effective application of model-based testing [32]. In particular, 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 access control test 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 issues remain largely open.

This paper presents a model-based testing approach for generating executable RBAC test code from a model-implementation description (MID), which consists of an RBAC test model and a model-implementation mapping (MIM). The test model is constructed from the given RBAC policies and functional requirements according to which the SUT is designed and implemented. It is represented by a Predicate/Transition (PrT) net [4], [31]. PrT nets are high-level Petri nets, a well-studied formal method for system modeling and verification. Since an RBAC test model specified by a PrT net captures both data and control flows of test requirements, our approach can generate complete model-based tests, including specific test inputs and test oracles (expected results). These model-level tests can further be converted into executable test code by using the given MIM, which maps the elements of the PrT net into the implementation constructs. Our approach has been implemented in MISTA, a framework for automated generation of test code in a variety of languages, including Java, C, C++, C#, PHP, and HTML [26] (MISTA is publicly available at http://cs.boisestate.edu/~dxu/research/MBT.html). The test code generated from the MID specification can be executed with the SUT to reveal potential access control defects.

To evaluate our approach, we have conducted empirical studies using three Java applications. To assess the fault detection capability, we used mutation analysis of RBAC rules, where mutants are created by injection of policy violations into the implementation. A mutant is an access control policy in which there is a fault in one of the rules. The test cases are executed against the faulty policy to check if the tests are able to detect the seeded fault. A mutant is considered to be killed when the tests report a failure. Mutation analysis is a commonly used method for evaluating the effectiveness of testing techniques [7]. Since the injected faults would represent the defects that are likely to occur in the implementation, the percentage of mutants killed by the tests created from a testing technique is often a good indicator of the testing effectiveness [7] in terms of fault-detection capability. Our experiments have shown that the test cases generated by our approach killed 99.5 percent of the 1,912 mutants and that 71 percent of the executable test code was generated automatically.

The remainder of this paper is organized as follows. Section 2 reviews the related work. Section 3 introduces the RBAC model in this paper. Section 4 elaborates on constructing test models. Section 5 discusses test generation from test models. Section 6 presents the transformation of tests into executable code. Section 7 describes the empirical studies. Section 8 concludes this paper.

2 Related Work

This paper is related to the research on model-based testing of access control policies and the research on modeling and analysis of access control with Petri nets. This section reviews the literature from these perspectives.

Masood et al. [14], [15] have investigated a state-based approach to test generation for RBAC policies. They first construct a finite state model of the RBAC policy and then derive tests from the state model. This model essentially captures the behaviors of role assignment, rather than access control rules as used in this paper. In addition, PrT nets in our approach not only capture control flows, but also data flows (e.g., test data and contexts). Based on the assurance management framework (AMF), Hu and Ahn [5] have proposed an approach to the generation of conformance tests of access control policies through constraint verification. Test cases are derived through verification by either removing or negating the security constraints. This approach is not concerned with the coverage criteria of access control rules. Our approach not only provides structured processes for building test models, but also generates tests to exercise all access control rules and their contexts. Mallouli et al. [12] proposed a model-based approach to testing access control policies by integrating Organizational Based Access Control (OrBAC) rules into an initial functional model represented by an extended finite state machine (EFSM). This approach has been validated only on a small program with no attempt to estimate the fault detection capability of their technique. The above work is not concerned about generation of executable test code.

Li et al. [11] proposed an approach to test generation from security policies specified as OrBAC rules. It consists of two steps: generation of test purposes from the OrBAC rules and generation of test cases from test purposes. This approach focuses on generation of test purposes from individual OrBAC rules. Our work integrates access control rules into an operational model and generates tests to cover different access control rules. Julliand et al. [8] 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. No explicit access control model was used. Jürjens [9] has developed an approach for testing security-critical systems based on UMLsec models. Test sequences for access control properties are generated from UMLsec models to test the implementation for vulnerabilities. To summarize, the existing work on model-based access control testing usually generates abstract model-level tests, not executable tests. Our approach produces executable test code by using a flexible mechanism for mapping modeling elements into implementation constructs. Pretschner et al. [20] have investigated a model-based approach to testing RBAC polices. This approach aims at selection of test targets for individual access control rule. Such test targets cannot be directly executed on the SUT. This paper captures the interactions of access control rules and fills the gap between test targets and executable test generation.

Martin and Xie [13] have investigated techniques for test generation from access control policy specifications written in XACML (OASIS eXtensible Access Control

Markup Language). This implementation-based approach targets individual XACML rules. Our model-based approach builds access control test models from functional models and access control rules. It is applicable no matter whether or not the SUT is based on XACML. In addition, our approach considers issues caused by erroneous interactions between the access control policy and the business logic. Blackburn et al. [1] have developed an approach to automated generation of functional security tests. Security properties are translated to a T-VEC test specification. T-VEC tools are then used to automatically generate test vectors and requirement-to-test coverage metrics. Our approach not only provides structured processes for building test models, but also automatically generates executable test code.

As a well-studied formal method, Petri nets have been applied to modeling and analysis of access control policies, which focuses on access control requirements and design [2], [6], [10], [17], [23]. Different from this literature, our work aims at finding access control defects in software implementation through model-based testing. Based on Colored Petri Net Processes (CPNPs), Huang and Kirchner [6] presented several composition operators that preserve the properties of sub-policies when they are composed. The properties, including completeness, termination, consistency, and confluence, are defined with respect to CPNPs. The CPNP of an access control policy is said to be complete if an access control decision is reachable from any initial marking. Because the possible initial markings for a CPNP can be infinite, it is unclear how a complete set of initial markings can be obtained. Our approach defines access control properties with respect to roles, activities, objects, and contexts of access control rules. Shafiq et al. [23] applied Colored Petri nets (CPNs) to the modeling of access control policies, which can capture such constraints as cardinality, separation of duty, precedence, and dependency constraints. Reachability analysis and a set of consistency rules were used to detect undesirable states that represent erratic behavior of the system. Deng et al. [2] applied PrT nets for the modeling and analysis of access control system architectures. Mortensen [17] used CPNs to specify an industrial access control system for the purposes of automatic generation of product code (not test code). A common characteristic of the above related work is their focus on the modeling of access control policies, not the interplay between access control policies and system functions. In our approach, however, the access control test models are constructed by integrating functional models and access control rules through structured processes. Knorr [10] discussed dynamic access control in Petri net-based workflows. Since functional activities and access control needs are both specified, unnecessary access can be eliminated by deriving access rights from the workflow.

TABLE 1
RBAC Rules for the Borrower/Student Role

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
7	Book	BorrowBook	day(MD)	Undefined
8	Book	ReserveBook	day(MD)	Undefined
9	Book	GiveBackBook	day(MD)	Undefined
10	Book	FixBook	day(d)	Undefined

3 The RBAC Model

The RBAC model in this paper follows the NIST RBAC model [3], [22] with a more general representation of role permission assignments (i.e., RBAC rules to be defined below). It consists of the following elements:

- A set of roles *R*,
- A role hierarchy $H \subseteq R \times R$, a partial order relation on *R*. $<r_1, r_2>$ denotes that r_1 is a direct super-role of r_2 or r_2 is a direct sub-role of r_1 (r_2 inherits all permissions of r_1),
- A set of subjects/users (human or computer agents) *Sub*,
- Role assignments $Sub \rightarrow 2^R$ (one subject can play a set of roles),
- A set of constraints on static separation of duties: $SSOD \subseteq R \times R$, where $<r_1, r_2> \in SSOD$ means that r_1 and r_2 cannot be assigned to the same subject,
- A set of constraints on dynamic separation of duties: $DSOD \subseteq R \times R$, where $<r_1, r_2> \in DSOD$ means that r_1 and r_2 assigned to the same subject cannot be activated within the same session, and
- A set of role permission/prohibition rules *R*.

Let *O* be a set of objects (or resources), *A* be a set of operations (called activities related to the resources), *C* be a set of contexts (representing Boolean expression constraints, for instance temporal contexts, location-based context etc.), and *{Permission, Prohibition}* be a set of authorization types.

Definition 1 (RBAC rule) *. An RBAC rule is a five-tuple* <r, o, a, c, τ>, *where* $\mathrm{r} \in \mathrm{R}, \mathrm{o} \in \mathrm{O}, \mathrm{a} \in \mathrm{A}, \mathrm{c} \in \mathrm{C}$, *and* $\tau \in$ {Permission, Prohibition}. *It means that role* r*'s activity* a *on object* o *is permitted (when* $\tau =$ Permission*) or prohibited (when* $\tau =$ Prohibition*) when context* c *holds.*

In a library management system, 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*, whereas *teacher* and *personnel* is the super-role of *director* and *secretary*), *SSOD = {<borrower, personnel>, < admin, borrower>}*, *DSOD = {<admin, director>}*, the set of objects is *{book, borrower Account, personnelAccount}*, and the set of activities is *{BorrowBook, ReserveBook, GiveBackBook, AdminActivity, ManageAccess, CreateAccount, ModifyAccount, DeliverBook, FixBook}*, and the set of contexts is *{day(WD), day(HD), day(MD)}*, where *WD, HD*, and *MD* refer to working day, holiday, and maintenance day, respectively. In Table 1, rules 1-6 are specified for the *borrower* role. *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.

Given a set of specified RBAC rules, there can be situations under which neither permission nor prohibition is specified. We treat these situations as "undefined conditions" and extend the set of authorization types to *{Permission, Prohibition, Undefined}*. From security assurance perspective, the undefined conditions must be tested because they likely lead to security holes in an implementation. To generate tests for these conditions, test modeling needs to cover both defined and undefined access control conditions. Our approach can automatically find such undefined conditions for a given set of RBAC rules. This paper will not elaborate on this due to the focus on test modeling and test generation. More details can be found in [29]. In the following, we discuss a few examples.

In Table 1, rules 1-6 are the specified access control conditions whereas rules 7-10 are added according to the undefined conditions. Among the specified rules 1-6, rules 2 and 3 are the only ones that are related to activity *BorrowBook* for *borrower*. Their contexts are *day*(*HD*) and *day*(*WD*). They do not cover maintenance days (*MD*)—whether a borrower can borrow books on maintenance days is not defined. Thus rule 7 is added. This is similar for *ReserveBook* (*rule 8*) and *GiveBackBook* (*rule 9*). Consider *FixBook* for *borrower*. There is no specified rule for *FixBook* under any context because it is a responsibility of *secretary*. From testing perspective, we need to test whether a borrower is allowed to perform *FixBook*. Thus we add rule 10, where *day*(*d*) is true for any *d* ∈ *{HD, WD, MD}*. Applying all activities to each role may require many rules to complete the specification. To deal with the complexity, 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 6.

The above RBAC specification in our approach supports all four levels of the NIST RBAC model [22], including flat, hierarchical, constrained, and symmetric RBAC. Flat RBAC has no role hierarchy, where hierarchical RBAC uses role hierarchies. Handling of role hierarchies in our approach will be discussed below. Constrained RBAC uses static and dynamic constraints to deal with separation of duties. In our approach, *SSOD* and *DSOD* specify the pairs of roles that cannot be assigned or activated together. Symmetric RBAC adds the notion of role permission review, which allows determining permissions of operations on objects assigned to specific roles. This is addressed by RBAC rules defined over roles, operations, and objects. As a more general formalism of permission specification, the RBAC rules also allow the specification of access contexts and prohibitions (i.e., negative permissions [22]).

In a role hierarchy, each role *r* inherits all permissions (i.e., RBAC rules) from its super roles. Let *S*(*r*) be the set of all super-roles of role *r*, and ℘(*r*) be the set of all rules with respect to *r*, including the rules defined for *r* and its super roles. $\wp(r) = \{<r, o, a, c, \tau>: \ <r, o, a, c, \tau> \in R\} \cup \{<r', o, a, c, \tau>: \ <r', o, a, c, \tau> \in R \wedge r' \in S(r)\}$. In this paper, we use ℘(*r*) to build role-permission test models that involve role *r* (refer to Sections 4.2, 4.3, and 4.4). In the above LMS example, *student*, as a sub-role of *borrower*, inherits all the RBAC rules in Table 1. These rules will be used to build the role-permission test model for *student* as a running example.

4 Constructing RBAC Test Models

RBAC testing involves testing of role-permission assignments (i.e., rules) and testing of user-role assignments with *SSOD* and *DSOD* constraints. We present two methods for constructing role-permission test models, discuss modeling of user-role assignments, and describe analysis of test models. Before describing test modeling and analysis, we first introduce PrT nets adapted from [28], [31].

4.1 PrT Nets for Test Modeling

Suppose constants start with an upper-case letter or a digit, and variables start with a lower-case letter. A term is a constant, a variable, or an expression $f(t_1, \ldots, t_n)$ of *n* arguments, where *f* is a function symbol and each t_i is a term. A term is called a ground term if it has no free variable. A label is a tuple of terms. Let Σ_l be a set of labels and Σ_f be a set of first-order logic formulas.

Definition 2 (PrT net). *A PrT net* N *is a tuple* <P, T, F, I, L, φ, M_0>*, where:*

1) P *is a finite set of places (also called predicates),*
2) T *is a finite set of transitions,*
3) F *is a finite set of normal arcs from places to transitions and from transitions to places, i.e.,* $F \subseteq P \times T \cup T \times P$,
4) I *is a finite set of inhibitor arcs from places to transitions (i.e.,* $l \subseteq F \times T$*),*
5) L: $F \cup I \rightarrow \Sigma_l$ *is a labeling function on arcs* $F \cup I$. L (f) *is the label for arc* $f \in F \cup I$. *When the label of an arc is not specified, the default label is a no-argument tuple* $< >$,
6) φ: $T \rightarrow \Sigma_f$ *is a guard function on* T. *The guard condition of transition* t*,* φ(t)*, is a first-order logical formula, which can evaluate true or false,*
7) M_0 *is a set of one or more initial markings.*

The PrT nets in this paper are a lightweight version of the original PrT nets [4] in that (a) arcs are labeled by tuples of terms, rather than formal sums $c_1l_1 + c_2l_2 + \cdots + c_nl_n$ (i.e., coefficient c_i of tuple l_i is 1 for all $1 \leq i \leq n$) and (b) tokens are tuples of ground terms rather than a formal sum. This simplification has resulted in efficient verification techniques [31]. We have successfully applied the above lightweight PrT nets to the modeling and analysis of various systems [28], [30], [31]. Also different from the Petri nets in the literature, we allow multiple initial states to be associated with the same net structure to represent different sets of test data. This makes it convenient to partition a large test data set into smaller ones so as to improve the effectiveness of test generation. Suppose $m_0 \in M_0$ is an initial marking. $m_0 = \{m_0(p) : p \in P\}$, where m_0 (*p*) is the set of tokens residing in place *p*. A token in *p* is a tuple of ground terms $<X1, \ldots, Xn>$, denoted as $p(X1, \ldots, Xn)$. The zero-argument token is denoted as $<>$. For token $<>$ in *p*, we simply denote it as *p*. We also associate a transition with a list of variables as formal parameters, if any.

Fig. 1 shows a sample PrT net, where *available*, *day*, and *borrowed* are places (circles); *BorrowBook* and *GiveBackBook* are transitions (solid rectangles). The guard condition of *BorrowBook* is $d = WD$. An arrow (e.g., from *available* to

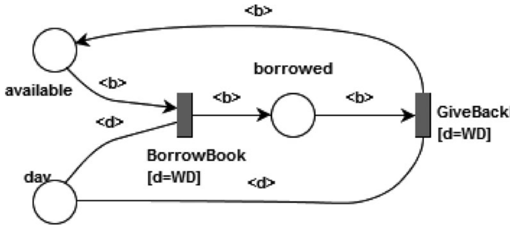


Fig. 1. A sample PrT net.

BorrowBook) represents a normal arc. A bi-directional arc (arc without arrow) between *n1* and *n2* (e.g., *day* and *BorrowBook*) represents two directed arcs: one from *n1* to *n2* and the other from *n2* to *n1*. MISTA also supports hierarchical PrT nets, where a transition can be corresponding to another PrT net (called subnet). To interpret a hierarchy of PrT nets, MISTA flattens the hierarchy into a single net by replacing each transition with the corresponding subnet.

Let *p* and *t* be a place and transition, respectively. *p* is called an input (or output) place of *t* if there is a normal 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 $\bullet t = \{p \in P : (p,t) \in F\}$, $t \bullet = \{p \in P : (t,p) \in F\}$ and ${}^*t = \{p \in P : (p,t) \in I\}$ be the input places, the output places, and the inhibitor places of transition t, respectively. Let *x*/*V* be a variable binding, meaning that variable *x* is bound to a ground term *V*. A variable 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 variable substitution and *l* be an arc label. *l*/θ denotes the token (tuple) obtained by substituting each variable in *l* for its bound value in θ. For instance, if $l = <b>$ and $\theta = \{b/B1, d/WD\}$, then $l/\theta = <B1>$.

Definition 3 (Enabledness). *Transition* t *in a given net is said to be enabled by substitution* θ *under a marking* m_0 *if:*

1) *For each input place* p *of* t *(i.e.,* $p \in \bullet t$*), there is a token that matches* l/θ*, where* l *is the label of the input arc from* p *to* t*,*
2) *For each inhibitor place* p *of* t *(i.e.,* $p \in {}^*t$*), there is no token that matches* l/θ*, where* l *is the label of the inhibitor arc between* p *and* t*, and*
3) *The guard* φ(t) *evaluates true with respect to* θ*.*

Suppose *{available*(*B1*)*, day*(*WD*)*, day*(*MD*)*}* is an initial marking for the net in Fig. 1. *BorrowBook* is enabled by $\theta = \{b/B1, d/WD\}$ because token <*B1*> in the input place *book* matches <*b*>/θ, token <*WD*> in the input place *day* matches <*d*>/θ, and the guard $d = WD$ is true according to θ. A marking m_i is said to be a deadlock or termination state if no transition is enabled under m_i.

Definition 4 (Transition Firing). *Firing transition* t *enabled by substitution* θ *under marking* $\mathrm{m_i}$ *removes the matching token from each input place and adds a new token to each output place. This leads to new marking* m_{i+1}*. Formally,* m_{i+1} *is defined as follows:*

1) *For each input place* p *of* t*,* $m_{i+1}(p) = m_i(p)l/\theta\}$*, where* l *is the label of the arc from* p *to* t *and* l/θ *is the matching token in* p*,*
2) *For each output place* p *of* t*,* $m_{i+1}(p) = m_i(p) \cup \{l/\theta\}$*, where* l *is the label of the arc from* p *to* t*, and* l/θ *is the new token added to* p*.*

In Definition 4, if *p* is both input and output place of *t* with the same arc label or the arc between *p* and *t* is bidirectional, the matching token remains in *p*. If the new marking is not a deadlock or termination state, an enabled transition can be fired according to Definition 5. Thus we can have sequences of transition firings. We denote a firing sequence as $m_0\,[t_1\theta_1 > m_1 \ldots m_i\,[t_{i+1}\theta_{i+1} > m_{i+1} \ldots\ [t_n\theta_n > m_n$, where $t_i\,(1 \leq i \leq n)$ is a transition, $\theta_i (1 \leq i \leq n)$ is the substitution for firing t_i, and $m_i (1 \leq i \leq n)$ is the marking after t_i fires, respectively. A marking m_n is said to be reachable from m_0 if there is such a firing sequence that transforms m_0 to m_n.

The above operational semantics of PrT nets provides a formal basis for the generation of tests when PrT nets represent test models (Section 5). The PrT nets in this paper also have declarative semantics—each transition is a first-order logic formula and each transition firing is an application of logical inference. Given a PrT net, each input (output) place *p*, together with the associated arc label $<x_1, \ldots x_n>$, is corresponding to an input (output) predicate $p(x_1, \ldots x_n)$; each inhibitor place *p*, together with the associated arc label $<x_1, \ldots x_n>$, is corresponding to a negative predicate $p(x_1, \ldots x_n)$ (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 postcondition Q is the conjunction of the output predicates and negation of each input predicate. P and Q are universally quantified. For *BorrowBook* in Fig. 1, $P =$ *available* (*b*) $\wedge$ *day* (*d*) $\wedge$ *d* = *WD* and *Q* = $\neg$*available*(*b*) $\wedge$ *borrowed*(*b*) $\wedge$ *day*(*d*). This lays the theoretical foundation for transforming declarative rules and contracts (preconditions and postconditions in first-order logic) into a PrT net (Section 4.3).

4.2 Building Role-Permission Test Models from Functional Test Models

RBAC rules are security constraints on system functions. If a functional test model is already available, we can integrate in it RBAC rules as constraints for the purposes of access control testing. Our previous work has demonstrated that PrT nets can be used to build test models for automated functional testing of various applications [26]. One approach to building a functional test model as a PrT net (referred to as functional net) is to formulate a test design (or workflow) by using the building blocks of PrT nets, including sequence, condition, repetition, concurrency, and modularity/hierarchy. This is similar to programming, which transforms a program design into code by using the building blocks (sequence, if-then-else, for/while/repeat, multi-threading, classes/function calls) of the given programming language. More details on the methodology for building functional test models with PrT nets can be found in [27].

Let us consider building a functional net for a subset of the student role activities in LMS—borrow, reserve, and return book. A student may borrow an available book and return a borrowed book. This consists of a sequence of two activities. A student may reserve a book and then borrow it. This is also a sequence of activities. When a borrowed book

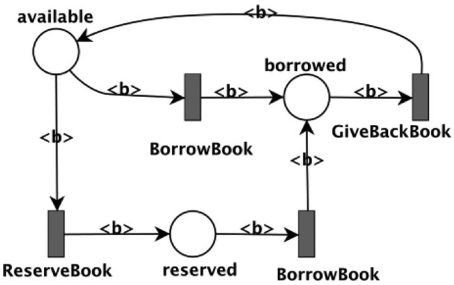


Fig. 2. Partial functional net of student activities in LMS.

is returned, it can be borrowed again—this implies a loop structure. When interested in a book, a student may borrow or reserve the book—this is a condition structure. Putting the above structures together would result in the functional net shown in Fig. 2.

Now we discuss how to build a role-permission test model by integrating RBAC rules into a functional net. For the sake of simplicity, let us first assume that a functional net involves the activities of a single role. An RBAC rule $<r, o, a, c, \tau>$ is related to a functional net only if activity *a* appears as a transition in the functional net. For example, rules 1-9 in Table 1 for the activities *BorrowBook*, *GiveBackBook*, and *ReserveBook* are related to the functional net in Fig. 2. Suppose the RBAC rules in $\wp(r)$ related to a functional net are $<r, o_1, a_1, c_1, \tau_1>$, $<r, o_2, a_2, c_2, \tau_2>, \ldots, <r, o_m, a_m, c_m, \tau_m>$. We integrate each RBAC rule $<r, o_i, a_i, c_i, \tau_i>$ (1) into the functional net as follows:

- If $\tau_i = Permission$ and $c_i = true$. Nothing is needed in that the rule is already represented by the activity transition a_i,
- If $\tau_i = Permission$ and predicates in c_i have corresponding places in the net (new places may be created for the predicates if necessary). Add a bi-directional arc between each place and the activity transition a_i (because the access does not change the context) and add c_i to the guard of a_i. Consider rule 3 in Table 1—student is allowed to borrow book only on a working day. The context $day(d) \wedge d = WD$ becomes an additional constraint of *BorrowBook* in the functional net. As shown in Fig. 3, we create a place *day*, add a bidirectional arc between place *day* and transition *BorrowBook*, label the arc with variable *d*, and add $d = WD$ to the guard condition of *BorrowBook*.
- If $\tau_i = Prohibition$, we add to the net a new transition Pa_i, which means a_i is prohibited. This transition shares the input and inhibitor places (i.e., functional preconditions) with a_i. If the predicates in c_i are corresponding to places, we add a bi-directional arc between each place and transition Pa_i (because it does not change the context) and add c_i to the guard of a_i. Consider rule 2 in Table 1—student is not allowed to borrow books on holiday. This is represented by transition *PBorrowBook* in Fig. 3.
- If $\tau_i = Undefined$, we handle the same way as Prohibition except that the new transition is named Pa_i. Consider rule 7 in Table 1—borrowing books on a maintenance day is undefined. This is represented by transition *UBorrowBook* in Fig. 3.

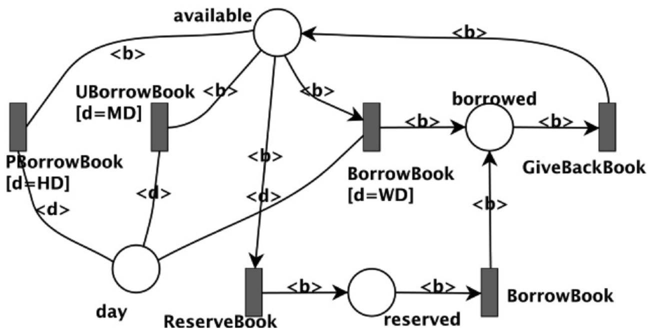


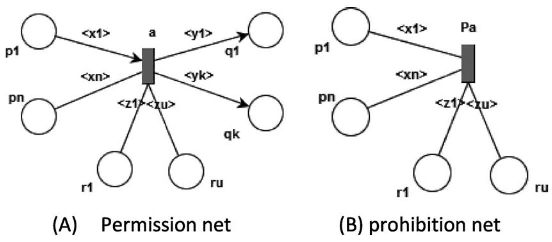
Fig. 3. Partial role-permission model of student activities in LMS.

The role-permission test model in Fig. 3 results from integrating rules 2, 3, and 7 in Table 1 into the functional net in Fig. 2. Other rules can be handled similarly except for rule 10 whose activity *Fixbook* does not appear in the functional net in Fig. 2. Since *Fixbook* is an activity of the *secretary* role, we can integrate rule 10 into the functional net of *secretary*. To represent multiple roles in a model, we use a global place *role*, which is connected to each transition with a bi-directional arc labeled by a role variable $<r>$. Which role can or cannot perform an activity is then represented by such a guard condition as $r = R$ or $r! = R$, where R is a particular role. For example, rule 10 can be integrated into the *secretary* test model by using a new transition *UFixBook*, whose guard includes $r = student$. This transition means that *fixbook* is undefined for *student*.

4.3 Building Role-Permission Test Models from Contracts

When a functional net is not available, we can construct role-permission test models from contracts (preconditions and postconditions). We have developed this method for two considerations. First, design by contract [16] is a widely accepted approach to functional specification. Second, RBAC rules as security constraints on system functionality cannot be tested without involving system functionality. Access control testing requires understanding of the preconditions and postconditions of the related activities. Consider testing the RBAC rule that a student is allowed to borrow books on working days. The test cannot be performed unless the functional precondition *"book is available"* is satisfied. The accurate test oracle cannot be determined without knowing its postcondition.

Contracts in the form of *precondition* → *postcondition* capture the dependencies among activities. 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*) → *available*(*b*). An activity can be associated with multiple contracts, representing different situations. A general precondition in the disjunctive form $P_1 \vee \ldots \vee P_n$ can be represented by multiple contracts $P_1 \rightarrow Q_1, \ldots, P_n \rightarrow Q_n$. For example, the contract of *BorrowBook*(*b*) is for any *b*, *available*(*b*) → *borrowed*(*b*) ∧ ¬*available*(*b*) or for any *b*, *reserved*(*b*) → *borrowed*(*b*) ∧ ¬*reserved*(*b*), where *reserved*(*b*) means book *b* is reserved. In this paper, the preconditions and postconditions are not necessarily accurate specifications of activity's semantics. Instead, they focus specifications of test requirements. For example, they may represent the ordering constraints on the activities involved in the rules.



(A) Permission net (B) prohibition net

Fig. 4. PrT nets for permission and prohibition rules.

The process for building permission test models by integrating RBAC rules $\wp(r)$ with the contracts of related activities is as follows. First, we partition $\wp(r)$ into a number of subsets in terms of roles or relevant activities. In LMS, student and teacher are independent roles although they have similar activities. So we group the RBAC rules for student and teacher into different subsets. Second, each subset of the RBAC rules 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 PrT net and composing the PrT nets of all rules into a single PrT net.

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

- If $\tau_i = Permission$, 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 postcondition 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 postcondition. As the context in an RBAC 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 $z_i \neq Z_i$ is added to the guard condition of the transition. 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. Fig. 4A shows the net, where the arc between p_1 and *a* is directed because $p_1(x_1)$ is negated in the postcondition; the arc between p_n and *a* is bi-directional as $\neg p_n(x_n)$ does not appear in the postcondition.
- If $\tau_i = Prohibition$, we convert the precondition of the contract into a net with one transition named *Pa* ("*P*" denotes "prohibition"). The postcondition 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 $\tau_i = Prohibition$. Fig. 4B shows the net.
- If $\tau_i = Undefined$, the transformation is similar to that for $\tau_i = Prohibition$. However, the transition is named by *Ua* ("*U*" denotes "*Undefined*").

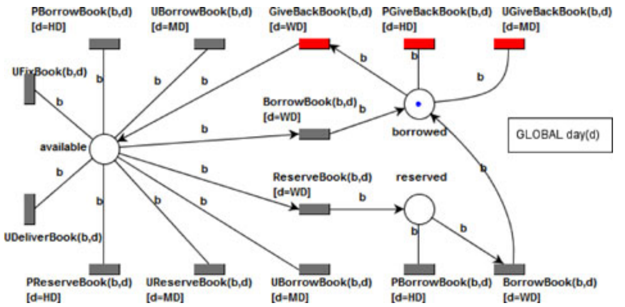


Fig. 5. RBAC test model for the student role.

The PrT nets for multiple rules and contracts are composed into one net through place fusion—places with the same name in different nets become one place in the composed net. Transitions with the same name in different nets become different transitions in the composed net (each of them is assigned a unique internal identity). The composed net can further be integrated with the nets from other RBAC rules and contracts.

Fig. 5 shows the PrT net that covers all the rules in Table 1 (except for transition *UDeliverbook,* which resulted from an additional rule). For clarity, an annotation is used to specify *day* as a global predicate, meaning that there is a bidirectional arc between *day* and each transition.

According to the semantics of the PrT nets, the above transformations have preserved the semantics of contracts and RBAC rules. For the sake of simplicity, the above discussion focuses on the rules for individual roles. It is easy to deal with multiple roles. 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, we use *role*(*r*) as an additional precondition for each activity. In the ASMS case study (Section 7), 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 build the test models based on the auction process (objects and activities), rather than individual roles. In brief, the transformation of RBAC rules into a PrT net essentially depends on the given subset of rules and the contracts of involved activities. If the given rules involve multiple roles, then the resultant net captures the behaviors of these roles.

4.4 Discussions on Role-Permission Test Models

After the structure of a PrT net for a role-permission test model is constructed, we define its initial markings by specifying test data (e.g., actual arguments of the activities) and

TABLE 2
Sample Contracts in LMS

Activity	*Contracts (Precondition→ Postcondition)*

BorrowBook	available(b) → borrowed(b) ∧ ¬available(b)
	reserved(b) → borrowed(b) ∧ ¬reserved(b)

ReserveBook	available(b) →reserved(b) ∧ ¬available(b)
GiveBackBook	borrowed(b) → available(b) ∧ ¬borrowed(b)

test configurations (e.g., system settings and contexts in the RBAC rules. Consider the net in Fig. 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.

Sections 4.2 and 4.3 present two different methods for constructing role-permission test models from functional nets and contracts, respectively. The two methods do not conflict with each other. According to the declarative semantics of PrT nets, a contract can be converted into a PrT net. Consider the contract $p_1(x_1) \wedge \ldots \wedge p_n(x_n) \rightarrow q_1(y_1) \wedge \ldots \wedge q_k(y_k) \wedge \neg\, p_1(x_1) \ldots .$ in Section 4.3. Its corresponding net is similar to the net in Fig. 4A, except for the places $r_1, \ldots, r_u$ introduced for the RBAC rules. Similarly, the corresponding PrT nets for individual contracts can be composed into a single PrT net. Let *M₁*(*N*, ℘) denote the test model obtained from functional net *N* and RBAC rules ℘ in Section 4.2 and *M₂*(*C*, ℘) denote the test model obtained from a set of contracts *C* and RBAC rules ℘ in Section 4.3. It is not difficult to prove that $M_1(N, \wp) = M_2(C, \wp)$ if *N* is the corresponding PrT net of *C*.

Consider the contracts of *BorrowBook*, *ReserveBook*, and *GiveBackBook* in Table 2. The composed PrT net for these contracts is the same as the net in Fig. 2. In other words, applying the method in Section 4.3 to the contracts in Table 2 and RBAC rules 2, 3, and 7 in Table 1 would result in the RBAC test model in Fig. 3, which was obtained by integrating the net in Fig. 2 with the same RBAC rules. Nevertheless, both methods are useful. They represent different modeling paradigms and build test models from different functional perspectives. Generally, functional nets are procedural whereas contracts are declarative. The former can be used to capture test workflows and the latter can specify logical dependencies between activities.

4.5 Building User-Role Assignment Test Models

A test model of user-role assignments specifies the test requirements related to assigning/deassigning users to roles, and activating/deactivating roles assigned to users. The assignment and activation must satisfy the static and dynamic constraints on separation of duties, i.e., *SSOD* and *DSOD*. As shown in Fig. 6, PrT nets can be used to formalize the above test requirements. In Fig. 6, places *user* and *role* represent users and roles, respectively. Places *assignedRole* and *activatedRole* represent the roles that are assigned to users and the roles that are activated, respectively. Places *ssod* and *dsod* represent the role pairs in *SSOD* and *DSOD*, respectively. Two *"assign"* transitions intend to assign roles to users. The lower *"assign"* transition assigns role r_2 to user *u* if *u* is not yet assigned to any role. The upper *"assign"* transition assigns role r_2 to user *u* which already plays role r_1 only when $<r_1, r_2> \notin SSOD$ (i.e., the inhibitor arc from *ssod* to assign) and $r_1 \neq r_2$ (i.e., the guard condition). Similarly, the lower *"activate"* transition activates role r_2 assigned to user *u* when *u* has no activated role yet. The upper *"activate"* transition activates role r_2 assigned to user *u* when *u* has an activated role r_1, $r_1 \neq r_2$, and $<r_1, r_2> \notin DSOD$. In addition, transitions *deassign* and *deactivate* remove role assignment and activation relations, respectively.

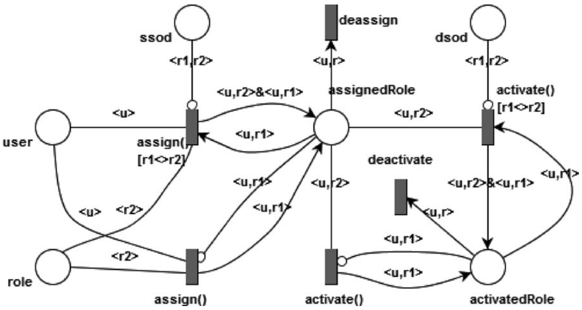


Fig. 6. A test model for user-role assignments.

4.6 Analyzing Test Models

Ensuring correctness of a test model is critical to model-based testing. Here we brief introduce our techniques for analyzing access control test models, including verification of transition/state reachability, verification of deadlock states, verification of assertions, and simulation.

Verification of transition reachability is to check if all transitions of a given PrT net are reachable from some given initial state. In an access control test model, each transition is corresponding to an access control activity. Thus, all transitions should be reachable from some given 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 set of initial states is specified incorrectly. Suppose $M_0 = \{m_1\}$, where *m₁* = {*available*(*B1*), *day*(*WD*), *day*(*MD*)}. *UBorrowBook* is not reachable from m_1 in the net in Fig. 5. In this case, M_0 is not specified properly.

Verification of goal reachability is to check if a given goal state is reachable from some initial state of the given PrT net. 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 set of initial states is specified incorrectly. In Fig. 5, 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 reachability verification would report that the above state is not reachable. A goal state is not limited to a specific marking. Generally, it is specified by a logical formula $P \rightarrow Q$. The reachability analysis of $P \rightarrow Q$ checks to see if there exists a reachable marking that satisfies $P \rightarrow Q$. For example, *"tokenCount*(*reserved*, *x*) ∧ *x* > *0"* refers to states where the place *reserved* has at least one token. This is expected to be reachable from the initial state {*available*(*B1*), *day*(*WD*), *day* (*MD*)} in the net in Fig. 5.

Verification of deadlock states is to check if the given PrT net can reach any deadlock state under which no transition is firable. If the verification result is different from our expectation, then the given net is specified incorrectly. For example, the net in Fig. 5 with the initial state {*available*(*B1*), *day* (*WD*), *day*(*MD*)} does not reach any deadlock states.

Verification of assertion $P \rightarrow Q$ is to check if the given assertion is satisfied by all states reachable from the given initial states. Consider the net in Fig. 5 with the initial state {*available* (*B1*), *day*(*WD*), *day*(*MD*)}. Place *reserved* should never have more than one token. This requirement can be represented by an assertion "$tokenCount\,(reserved, x) \wedge x < 2$". Generally, verification of goal reachability aims at the analysis of existential properties ("there exists") whereas verification of assertions targets the analysis of universal properties ("for all").

MISTA also offers a visual simulator for stepwise execution of test models. At each state, the simulator 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 let MISTA continuously fire randomly selected enabled transitions. This is useful for finding out unexpected behaviors in a test model.

5 Generating RBAC Tests

This section describes how to generate model-level access control tests from RBAC test models.

Definition 5 (Model-level RBAC test). *Given an RBAC test model represented by a PrT net, a test case is a firing sequence* $m_0[t_1\theta_1 > m_1, \ldots, [t_n\theta_n > m_n$ *in the PrT net, where*

1) *m_0 is the initial setting of the test,*
2) *Transition firings $t_1\theta_1, \ldots, t_n\theta_n$ are* test inputs, *i.e., calls to the activities in RBAC rules or related to role assignment (assignment, de-assignment, activation, and de-activation). Suppose transition t_i is corresponding to activity $a(x_1, \ldots, x_m)$ and substitution $\theta_i = \{x_1/u_1, \ldots, x_m/u_m\}$. Then $t_i\theta_i$ $(1 \leq i \leq n)$ represents method call $a(u_1, \ldots, u_m)$, where u_j $(1 \leq j \leq m)$ is x_j' actual argument,*
3) *$m_0, \ldots, m_n$ are* oracle values *for respective test inputs $t_i\theta_i (1 \leq i \leq n)$. For each place $p \in P$ and each token $< v_1, \ldots, v_m > \in M^k o(p)$, proposition $p(v_1, \ldots, v_m)$, when used as an oracle value, is expected to evaluate to true in the SUT.*

For example, suppose $M_0 = \{m_0\}$, m_0 = {available(B1), day (WD), day(HD), day(MD)} *for the model in Fig. 5.* <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, where:*

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 RBAC 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 system settings for access control contexts.

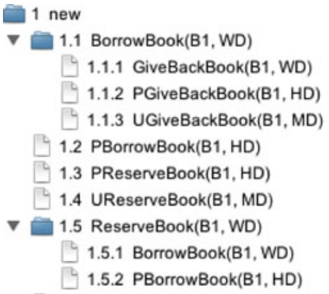

1 new
1.1 BorrowBook(B1, WD)
1.1.1 GiveBackBook(B1, WD)
1.1.2 PGiveBackBook(B1, HD)
1.1.3 UGiveBackBook(B1, MD)
1.2 PBorrowBook(B1, HD)
1.3 PReserveBook(B1, HD)
1.4 UReserveBook(B1, MD)
1.5 ReserveBook(B1, WD)
1.5.1 BorrowBook(B1, WD)
1.5.2 PBorrowBook(B1, HD)

Fig. 7. Portion of a test tree.

Therefore, test generation from an RBAC test model in our approach is to produce firing sequences (test cases) from the RBAC test model according to a certain strategy (e.g., to achieve a coverage criterion). The 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). Fig. 7 shows portion of the test tree generated for the reachability coverage of the test model in Fig. 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 RBAC 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. In order to represent tests generated from multiple initial markings (i.e., 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.

MISTA supports automated test generation for several coverage criteria, such as reachability coverage, state coverage, and transition coverage. In a role permission test model, a transition is corresponding to one RBAC rule. In a role assignment test model, a transition is corresponding to role assignment, de-assignment, activation, or de-activation. 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 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 coverage subsumes transition coverage and state coverage because the reachability coverage includes each reachable transition and each reachable state. The case studies in this paper use the test generator for reachability coverage as described in Algorithm 1. It extends the previous implementation [26] with two techniques: partial ordering of concurrent firings and pairwise combinations of transition inputs. This extension can reduce the number of test sequences so as to deal with more complex test models.

After initialization (line 1), Algorithm 1 creates a node for each initial marking and adds the node to the queue for expansion (lines 2-5). While the queue is not empty (line 6), it takes a node from the queue for expansion (lines 7-41). To expand a node, the algorithm first computes and collects eligible firings for all transitions (lines 8-29) and then creates a new child node for each of the eligible firings (lines 30-40). If the resultant marking of a firing has not appeared before, the new node for the firing is also added to the queue for further expansion (lines 37-39). In Algorithm 1, *currentNode. marking* refers to the marking in *currentNode*. *allSubstitutions* (*t*, *currentNode. marking*) denotes all substitutions that enable *t* under *currentNode.marking* (line 13). According to the definition of transition enabledness (Definition 3), Substitutions for a transition are obtained by unifying the arc label of each input or inhibitor place with the tokens in this place and evaluating the guard condition. The collection of all substitutions is computed through backtracking—after a substitution is obtained, backtracking is applied to the unification process until all substitutions are found. In this case, all possible combinations of tokens in the input places are covered. Different from *allSubstitutions*(*t*, *currentNode. marking*), *pairwiseSubstitutions*(*t*, *currentNode. marking*) (line 11) denotes all pairwise substitutions that enable *t* under *currentNode.marking*. Pairwise substitutions are substitutions where pairwise combinations of tokens in the input places are applied. Suppose there are 10 input places and each of them has 10 values (single argument tokens). There are 10^{10} combinations of these values. Using pairwise combination, however, 120 combinations can cover all pairs of the values. Algorithm 1 also offers an option of using partial ordering of concurrent firings (lines 16-29). The total ordering of *n* ($n > 1$) concurrent firings yield *n!* sequences, where the partial ordering only produces one sequence. For a group of concurrent firings, only one of them is selected to create new node for expansion (lines 20-27). Because the selected firing does not disable other concurrent firings in the group, its concurrent firings will remain enabled at the resultant marking and thus will be expanded only in the next levels.

In addition, Algorithm 1 allows partitioning of test data due to the support of multiple initial states. We can divide a large set of test data into multiple initial states. Consider a transition with three input variables and each of them has 10 values. If all of the input values are specified in one initial state, the transition can be fired by 1,000 different combinations of the inputs. If the input values are specified into two initial states and each initial state contain five values of each input variable, then the possible firings of the transition are 250 (5*5*5*2).

The complexity of Algorithm 1 is exponential to the size of the test model because it covers all the states and state transitions of the test model. The test tree can be very large although pairwise combination, parting ordering, and partitioning of test data can significantly reduce the number of tests.

6 Generating Executable Test Code

In the previous discussions, the RBAC test models can be independent of the system implementation. Thus, tests generated from an RBAC test model are not immediately executable with the SUT. For example, the RBAC test model of the *student* role does not specify how *BorrowBook* can be performed against the SUT. Our approach allows a MIM specification to be created for converting all model-level tests into executable code automatically. Although our approach supports the generation of test code in a variety of programming and scripting languages, this paper focuses on Java/JUnit (JUnit is a test framework for writing and executing Java test code).

Algorithm 1: Test generation for reachability coverage

```
Input: PrT net (P, T, F, I, L, $\varphi$, $M_0$).
Output: transition tree with dirty tests.
Declare: root, newNode, currentNode are nodes;
         queue is a queue of nodes;
         firings is a list of firings;
         newMarking is a marking;
1.  initialization: queue ← ø ; root ← create a node
2.  for each initial marking $m_0 \in M_0$, do
3.     create the initial state node as a child of the root
4.     add the node into queue
5.  end for
6.  while queue ≠ ø do
7.     currentNode ← first node in queue;
8.     firings ←ø ;
9.        for each transition t ∈ T, do
10.          if use pairwise combination
11.             firings ← firings ∪ {(t, $\theta$): $\theta$ ∈ pairwiseSubstitutions(t, currentNode.marking)};
12.       else
13.          firings ← firings ∪{(t, $\theta$): $\theta$ ∈ allSubstitutions(t, currentNode.marking)};
14.       end if
15.    end for
16.    if use partial ordering
17.       tmpFirings ← firings;
18.       firings ← ø ;
19.       while tmpFirings ≠ ø do
20.       f ← first firing in tmpFirings;
21.       firings ← firings ∪ {f};
22.       remove f from tmpFirings;
23.       for each f′ ∈ tmpFirings do
24.          if f and f′ are concurrent
25.             remove f′ from tmpFirings;
26.          end if
27.       end for
28.    end while
29. end if
30. for each (t, $\theta$) ∈ firings, do
31.    newMarking ← the marking of firing t with $\theta$
32.    newNode.parent ← currentNode;
33.    newNode.marking ← newMarking;
34.    newNode.transition ← t;
35.    newNode.substitution ← $\theta$;
36.    add newNode to currentNode.children;
37.    if newMarking has not occurred in the tree
38.       add newNode to queue;
39.    end if
40.    end for
41. end while
42. return root
```

Definition 6 (MIM) . *A MIM specification for a PrT net is a 7-tuple* $<\mathrm{ID}, \mathrm{f_o}, \mathrm{f_c}, \mathrm{f_a}, \mathrm{f_m}, \mathrm{l_s}, \mathrm{h}>$, *where*

1) ID *is the identity of the SUT tested against the PrT net,*

TABLE 3
Portion of the MIM Specification for the RBAC Test Model of Student in LMS

MIM	*Model element*	*Implementation element*	*Notes*
f_o	B1	Book1Title	Book1Title is a named constant in the helper code (see below)
f_c	ReserveBook (b, d)	doPermittedReserve(b)	doPermittedReserve is a test method in the helper code. It fails if an exception (particularly security-related exception) is thrown.
	BorrowBook (b,d)	doPermittedBorrow(b)	doPermittedBorrow is a test method in the helper code. It fails if an exception (particularly security-related exception) is thrown.
	PGiveBack-Book(b, d)	doProhibitedGiveBack (b)	doProhibitedGiveBack is a test method in the helper code. It fails if an exception (particularly security-related 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.
	reserved(b,d)	isBookReserved(b)	isBookReserved is a query method for verifying whether the status of the book is reserved
f_m	day(WD)	ContextManager.current-Context = ContextMan-ager.workingday;	It sets the concurrent context to working day.
	day(HD)	ContextManager.current-Context = ContextMan-ager.holiday;	It sets the concurrent context to holiday day.
l_s	day		The access control context day is a system setting
$\mathrm{f_h}$(PACKAGE)	package com.library.test.software.modeltest;		Helper code for the package statement of Java test code
$\mathrm{f_h}$(CODE)	private final String Book1Title = “Software security”;...		Declarations and methods to be included in the test code

2) *f_o is the object function that maps constants in the PrT net to objects in the SUT. Given an constant* X *in the PrT net,* $\mathrm{f_o}$*(X) is an object in the SUT,*
3) *f_c is the method mapping function that maps transitions in the PrT net to methods (operations) in the SUT,*
4) *f_a is the accessor function that maps places (predicates) in the PrT net to accessors in the SUT,*
5) *f_m is the mutator function that maps places (predicates) in the PrT net to mutators in the SUT,*
6) *l_s is the list of predicates in the PrT net that are implemented as system settings in the SUT. These predicates are called setting predicates,*
7) h *is the helper code function that defines user-provided code to be included in the test code.*

Table 3 presents portion of a MIM in LMS. Object function f_o maps objects in the RBAC test model to objects in the SUT. In LMS, for example, book *B1* in the test model of the *student* role is corresponding to *Book1Title*, which 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*. The methods for testing individual activities depend on how the SUT is implemented, e.g., what types of security exceptions will be reported. In our 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 *UndefinedSecuritPolicyException*. 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*.

Accessor function f_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_m maps the system setting predicates in l_s to operations in SUT so that the SUT can be configured to a specific testing state. For example, predicate *day* in LMS is a system setting. As an access control precondition, it must be set correctly before 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 = ContextManager.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.

- *h(package)*: package statement.
- *h(import)*: import statements.
- *h(setup)*: Junit setup code.
- *h(teardown)*: Junit teardown code.
- *h(local)*: named constants, variables, and methods to be put in the generated test code. These definitions can be used by f_o, f_c, f_a and f_m.

To generate test code from a test tree produced by a test generator (e.g., Algorithm 1), we create a JUnit test class (i.e., test suite) for the sub-tree of each initial state. Such a JUnit test class consists of test methods—each test method is corresponding to a firing sequence (test case) in the sub-tree. Algorithm 2 below describes how JUnit test classes are generated for a given test tree. For each JUnit test class, it first imports user-provided package and import statements (line 2), create the signature of the test class (line 3), declares an instance variable whose type is the given class ID (lines 4-6), and imports user-defined local code (line 5), setup code (lines 6-7), and teardown code (line 14). If the setup code is not provided in h, we generate a setup method for the initial state (lines 9-13). Each token $p(a_1, \ldots, a_k)$ in the initial state is converted into Java code for achieving the test configuration (lines 10-12). To do so, we first transform model-level objects a_i to implementation-level objects $f_o(a_i)$ and then call the mutator function f_m (line 14). This is similar for dealing with system settings in test sequences (lines 20-22). Then Algorithm 2 retrieves all tests for the initial state (lines 15-16) and generates a test method for each test $m_0[t_1\theta_1 > m_1, \ldots, [t_n\theta_n > m_n$ (lines 17-29). For each transition firing $t_i\theta_i$, the test method configures the system settings for the test operation (lines 20-22), issues the test operation (line 23), and verifies the oracle values (lines 24-26). Note that each model-level object a_i or b_i is mapped to the implementation-level object $f_o(b_i)$ or $f_o(b_i)$. The test method does not include explicit calls to setup and teardown because these calls are executed automatically by JUnit.

Consider the aforementioned sample test in Section 5

m_0, ReserveBook(b/B1, d/WD), m_1, BorrowBook(b/B1, d/WD), m_2, PGiveBackBook(b/B1, d/HD), m_3

or simply

m_0, ReserveBook(B1, WD), m_1, BorrowBook(B1, WD), m_2, PGiveBackBook(B1, HD), m_3.

The generated JUnit test method is as follows:

```
1   public void test12() throws exception {
2.  ContextManager.currentContext = ContextManager.
    workingday;
3.  doPermittedReserve(Book1Title);
4.  assertTrue(isBookReserved(Book1Title));
5.  ContextManager.currentContext = ContextManager.
    workingday;
6.  doPermittedBorrow(Book1Title);
7.  assertTrue(isBookBorrowed(Book1Title));
8.  ContextManager.currentContext = ContextManager.
    holiday;
9.  doProhibitedGiveBack(Book1Title);
10. assertTrue(isBookBorrowed(Book1Title));
11. }.
```

Lines 2-4 are generated for test operation *ReserveBook*(*B1*, *WD*)—setting up the testing context (line 2), issuing the test operation (line 3), and verifying the oracle value (line 4). According to the object mapping in Table 3, *B1* is corresponding to *Book1Title.* The precondition of *ReserveBook* (*Book1Title*, *WD*) is *day*(*WD*), and *day* is a system setting predicate. Before issuing the test operation, we have to set the context to a working day—this is done by calling the mutator function for *day*(*WD*), i.e., *ContextManager.currentContext = ContextManager.workingday;* specified in Table 3 (refer to lines 20-22 of Algorithm 2). According to the method mapping function, the test operation *reserveBook* (*Book1Title*, *WD*) is implemented by *doPermittedReserve* (*Book1Title*) (refer to line 23 of Algorithm 2). The postcondition of *reserveBook*(*Book1Title*, *WD*) is *reserved*(*Book1Title*, *WD*). According to the accessor function for *reserved*(*b*,*d*), *reserved*(*Book1Title*, *WD*) is verified by *isBookBorrowed* (*Book1Title*) in the implementation (refer to lines 24-26 of Algorithm 2). Similarly, lines 5-7 are generated for *BorrowBook*(*B1*, *WD*), and lines 8-10 are generated for *PGiveBackBook*(*B1*, *HD*).

Algorithm 2. Generation of test code in Java/JUnit

```
Input: transition tree root, MIM = $<ID, f_o, f_c, f_a, f_m, l_s, h>$.
Output: Java/JUnit test code.
Declare: initialStates is a set of initial markings;
         initState is an initial marking;
         leafNodes is a set of leaf nodes;
         testSequences is a set of test sequences;
         testSequence refers to one test sequence;
1.  for each initState ∈ initialStates, do
2.      create header according to h(package) and h(import);
3.      create test class signature according to ID (the class under test) and the index of initState;
4.      declare an instance variable whose type is ID;
5.      import h(local) into this test class;
6.      if h(setup) is defined
7.          import h(setup) to this test class
8.      else // generation of setup according to initState
9.          create the signature of the setup method;
10.         for each p ∈ P and token <$a_1, \ldots, a_k$> in p, do
11.             create $f_m$ (p($f_o(a_1), \ldots, f_o(a_k)$)) in the setup body;
12.         end for
13.         create the closing part of the setup method;
14.     import h(teardown) to this test class
15.     leafNodes ← all leaf nodes corresponding to initState;
16.     testSequences ← all tests according to leafNodes;
17.     for each $m_0$ [$t_1\theta_1$>$m_1$,. . ., [$t_n\theta_n$>$m_n$ ∈ testSequences, do
18.         create the signature of the JUnit test method;
19.         for (i = 1 to n) do
20.             for each input place p of $t_i$ such that p ∈ $l_s$ and <$a_1, \ldots, a_k$> ∈ $m_i$ (p), do
21.                 create system setting code $f_m$(p($f_o(a_1), \ldots, f_o(a_k)$));
22.         end for
23.         create component call code, $f_c$(c($f_o(b_1), \ldots, f_o(b_k)$));
24.         for each p($a_1, \ldots, a_k$) such that <$a_1, \ldots, a_k$> ∈ $m_i$ (p) do
25.             create assertion $f_a$ (p($f_o(a_1), \ldots, f_o(a_k)$));
26.         end for
27.     end for
28.     create the closing part of the test method;
29.   end for
30. end for
```

7 Empirical Studies

In this section, we describe our case studies for evaluating the fault detection capabilities of our approach. The case studies focus on testing of role permission assignments, rather than user-role assignments. We first introduce how the experiments were set up and then present the results of our experiments. Finally, we discuss scalability issue as well as threats to validity in our studies.

7.1 Experiment Setup

Our case studies are based on three Java programs, library management system, Virtual Meeting System (VMS), and

TABLE 4
Subjects of the Empirical Studies

Subject	*LOC*	*#Classes /Methods*	*#R*	*#O*	*#A*	*#Rules*
LMS	3,204	62/335	5	4	12	33
ASMS	10,703	122/797	6	6	23	107
VMS	6,077	134/581	9	9	18	106

LOC: lines of source code; #Classes/methods: number of classes and methods in the Java source code; #R: number of primitive roles tested; #O: number of objects; #A: number of activities; #Rules: total number of RBAC rules for primitive roles.

TABLE 5
Results of the Empirical Studies

	#T	LOC	GLOC	%GLOC	#M	# K	Score
LMS	207	3,185	1,789	56.2%	243	233	95.9%
ASMS	501	5,291	4,331	81.9%	914	914	100%
VMS	225	2,538	1,728	68.1%	755	755	100%
Total	933	11,014	7,848	71%	1,912	1,902	99.5%

#T: number of test cases generated; LOC: lines of executable JUnit test code; GLOC: lines of JUnit test code generated by MISTA;%GLOC: percentage of JUnit test code generated by MISTA; #M: number of access control mutants; #K: number of mutants killed by the generated test cases; Score: mutation score = #K/#M.

Auction Sale Management System (ASMS). Table 4 presents the main parameters of these programs.

LMS offers services to manage books in a public library. The books can be borrowed and returned by the users of the library on working days. LMS distinguishes three types of users: public users who can borrow five books for three weeks, students who can borrow 10 books for three weeks and teachers who can borrow 10 books for two months. LMS is managed by an administrator who can create, modify, and remove user accounts. Books in the library are managed by a secretary who orders books or adds them when they are delivered. The secretary can also fix the damaged books in certain days dedicated to maintenance. When a book is damaged, it must be fixed. While it is being fixed, this book cannot be borrowed but a user can reserve it. The director of the library has the same accesses than the secretary and can consult the accounts of the employees. The administrator and the secretary can consult all user accounts. All users can consult the list of books in the library.

VMS offers simplified web conference services. It is used in an advanced software engineering course at University of Rennes 1, in France. The virtual meeting server allows the organization of work meetings on a distributed platform. When connected to the server, a user can enter or exit a meeting, ask to speak, eventually speak, or plan new meetings. Each meeting has a manager. The manager is the person who has planned the meeting and has set its main parameters (such as its name, its agenda, etc.). Each meeting may also have a moderator, appointed by the meeting manager. The moderator gives the floor to a participant who has asked to speak.

ASMS allows users to buy or sell items online. A seller can start an auction by submitting a description of the item he wants to sell and a minimum price (with a start date and an ending date for the auction). Then usual bidding process can apply and people can bid on this auction. One of the specificities of this system is that a buyer must have enough money in his account before bidding.

The protocol of our experiment is as follows. First, we construct and analyze the test models. The test models of LMS and ASMS are created based on the contracts of related activities (Section 4.3), whereas the test models of VMS are constructed based on functional test models (Section 4.2). Second, we create the MIM specification for each test model as described in Section 6. 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 of the mutants of the RBAC rules.

The mutants were created automatically by the MutaX tool (https://sites.google.com/site/servalteam/tools/mutax) using five types of mutation operators: 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: (a) mutants related to non-implemented activities because the tests could not be performed, (b) mutants with inconsistent access control rules. These mutants are typically created by the adding rule operator, and (c) mutants that have the same behavior as the original version.

Interested reader may contact the corresponding author to obtain a copy of the source code, mutants, and test models of the case studies. The implementation of our approach is included in the current release of MISTA (the URL for download is given in Section 1), which can reproduce the test code from the test models used in the empirical studies.

7.2 Results

The results of our experiments are summarized in Table 5. For LMS, there were 207 test cases in 3,185 lines of code. 56.2 percent of the test code was generated. The tests killed 233 out of 243 mutants, with an overall detection rate of 95.9 percent. 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 added 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*).

For ASMS, the test models resulted in 501 tests. 81.9 percent of the 5,291 lines of test code was generated. The tests killed all of the 914 mutants. For VMS, the test models produced 225 tests. 68.1 percent of the test code was generated. The tests killed all of the 755 mutants.

The mutation scores in our case studies are almost perfect for two main reasons. First, our approach is able to deal with undefined access control conditions. 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 coverage can cover all access control rules, objects, activities, and contexts due to automated test generation and execution. In comparison, transition coverage has a low fault detection capability. As part of our initial experiment, the application of transition coverage to the *student* role in LMS only killed about 50 percent of the mutants because many access contexts were not exercised.

While the subject programs in our case studies have a reasonable size for the purposes of quantitative evaluation (e.g., ASMS has more than 10,000 LOC), they have a small number of roles and shallow role hierarchies. In the literature on RBAC specification and analysis, the number of roles and depth of role hierarchies are important factors for measuring the complexity and scalability of RBAC systems [22], [24]. For a complex real-world RBAC system with a large number of roles and a deep role hierarchy, our approach relies on the "divide and conquer" strategy and builds a number of test models to deal with subsets of roles and access control rules (rather than a single model for all roles and rules). Building test models (e.g., contracts and functional models) is essentially a manual process. It is also different from system modeling for design and verification. The former focuses on what needs to be tested with carefully selected test data, whereas the latter often deals with system-wide behaviors and input spaces. Thus, the complexity and scalability of test generation for individual test models in our approach is not directly related to the total number of roles and the depth of role hierarchy in the SUT. Instead, it depends on the number of access control rules, number of objects, number of activities, number of access contexts, and test data involved in a given test model. In theory, the complexity of Algorithm 1 for reachability coverage is exponential to the sizes of these factors because it aims to cover every possible state transition. These factors determine the number of states and state transitions in the model.

To evaluate the scalability of the implementation of our approach as a whole, including test generation (Algorithm 1) and code generation (Algorithm 2), we have conducted a performance testing experiment using a test model with eight different initial states. These initial states account for different complexity levels of the state space. Table 6 shows the results of our performance evaluation experiment, where *Vi* ($1 \leq i \leq 8$) denotes the model with the ith initial state. The number of states ranges from 90 to 19,000 (i.e., row *A*). The most complex case is *V8* (the last column), which has 19,000 states, 79,623 state transitions in the test tree, 61,091 test cases, 658,309 test activities (i.e., transition firings) in all test cases, and 1.3 millions of lines of test code (the file size is 49.5 MB). The Eclipse IDE for developing Java programs failed to open this test code file because it is too big. However, it only took less than 5 seconds to generate the test tree (i.e., Algorithm 1) and a total of about 8 seconds to generate the test code (including Algorithm 1 and Algorithm 2) on a MacBook Pro (Intel Core i7 2.6 GHz, 8 GB memory). For modified models with larger state space, MISTA is unable to generate test code because it runs out of memory. This experiment has demonstrated that the scalability of the current implementation is more constrained by space than time. Nevertheless, for all the test models in Table 6 (even for the simplest *V1* with 90 states and 200 state transitions), manual test generation and management are almost infeasible. We believe our work is a major improvement to the existing manual practices. It is thus of practical utility.

TABLE 6
Results of Performance Evaluation

	V1	V2	V3	V4	V5	V6	V7	V8
A	90	300	600	1.3K	4K	7K	10K	19K
B	200	800	1.7K	4.1K	15K	28K	41K	80K
C	100	500	1.1K	2.8K	11K	21K	31K	61K
D	600	3.1K	7.9K	22K	97K	201K	311K	658K
E	1.8K	8K	19K	50K	203K	408K	614K	1.3M
F	0.07	0.22	0.36	0.58	1.18	1.87	2.63	4.65
G	0.13	0.30	0.42	0.94	1.46	2.43	3.64	8.14

A: number of states; B: number of state transitions (i.e., edges) in the test tree; C: number of test cases (sequences); D: number of test activities (transition firings) in all test cases; E: lines of test code; F: test generation time in seconds (Algorithm 1); G: test code generation time in seconds (Algorithm 1 + Algorithm 2). A, B, C, D, and E are rounded to the nearest ten, hundred, or thousand.

7.3 Threats to Validity

The main result of our study is that our approach is highly effective in detecting access control defects. The key aspects that have led to this result include formalization of function nets and contracts, generation of access control tests with the reachability graph 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 general software applications where access control is an important security mechanism.

First, in LMS and ASMS, we formalize the contracts of activities involved in the RBAC rules and transform them together with the RBAC rules into a PrT net. Because preconditions and postconditions are used for test generation, they do not have to capture the precise semantics of activities. For example, they may only represent the ordering constraints of the activities under test. In the case studies, we were able to complete the formalization and transformation. For a real-world application, this may not be an easy task. It can depend on the application domain and complexity. In VMS, however, the role-permission test models are constructed from the functional test models. This method appears to be more practical.

Second, the underlying assumption of using the reachability tree coverage for test generation is that the test model has a finite number of states. Due to the small sizes of the subject programs in the case studies, we were able to generate tests to cover all rules and combinations of objects, resources, and contexts because the test models have a small number of states. Although RBAC test models are often related to partial behaviors of a SUT, generation of the above RBAC tests may not be feasible for complex real-world software where access control is involved in large state space.

Third, we were able to generate executable test code by completing the MIM specifications of the test models in the case studies. In the MIM specifications, calls to individual activities and verification of test oracles are programmed. Policy violations are assumed to be handled consistently—an exception is thrown when a prohibited or undefined activity is requested. For real-world software, the individual activity tests and the test oracles may not be completely programmable. If test execution requires human intervention, the number of tests that can be executed with limited budget and time would be decreased. This in turn can affect the effectiveness. Although our approach is applicable to a variety of languages supported by MISTA, the subject programs in the case studies were limited to Java applications.

Finally, the evaluation of fault detection capability is based on the mutation analysis of RBAC rules. The mutants of RBAC rules were created and have been used in several studies by the group at University of Luxembourg before the proposed approach was initiated. The tasks of modeling, test generation, and test execution for the case studies were accomplished independently by the first author's group at a different University. This assures the objectivity of mutation-based evaluation. While we believe the mutants created by the five types of mutation operators have represented a vast majority of access control defects, they do not necessarily cover every possible fault in real-world software.

8 Conclusion

We have presented the tool-supported, model-based approach to automated conformance testing of RBAC policies. It provides structured processes for building role-permission test models from functional nets and contract specifications. It also automatically generates executable access control tests from the test models. The empirical studies using three Java programs have demonstrated that our approach is highly effective in detecting access control defects and that 56-82 percent of the executable test code is generated automatically.

The contribution of this paper is twofold. First, we present methods for constructing operational RBAC test models by integrating declarative RBAC rules with functional test models represented by PrT nets or contracts (preconditions and postconditions) of the associated access control activities. Because RBAC rules are non-functional constraints on associated activities or system functions, our methods show that the systematic testing of interrelated RBAC rules can be built upon functional requirements. Second, we present an approach to automated generation of executable test code for exercising the RBAC rules. Once the MID (test model and MIM) specification is completed, test generation and test execution would need no human intervention. This automation has facilitated our empirical studies that aimed at evaluating the fault detection capability of our testing approach through mutation analysis. To the best of our knowledge, neither of these aspects has been addressed in the literature.

This paper has focused on the testing of role-permission assignments and user-role assignments in RBAC, where users, roles, and permission rules are predefined. Our future work will extend the current approach to the testing of Administrative RBAC (ARBAC) policies, which specify how administrators may change user-role assignments and role-permission assignments [24]. Understanding and testing the effects of an ARBAC policy are critical to system security. The RBAC rules in our approach can be adapted to specify ARBAC permissions to perform administrative operations on user-role and role-permission assignments. Another direction of future work is to extend the current approach for automated testing of obligation policies, which are critical to assuring information security and system accountability [19]. Obligation policies allow expressing actions that users should take to fulfill the responsibilities, in addition to usage control requirements, the mandatory actions related with some granted accesses. Obligation policies raise several challenges in automated test generation and execution. First, how can we generate test actions when obligation rules cannot be enforced by a computer system? Second, since obligation is usually related to a time window, how can we generate time-sensitive obligation tests? Third, how can we measure the test adequacy of obligation policy, particularly with respect to timing conditions of obligation fulfillment and violation? To address these issues, we will first need to enhance PrT nets for building testable models of obligation policies.

Acknowledgments

This work was supported in part by the US National Science Foundation (NSF) under grants CNS 1004843, CNS 1123220, and CNS 1359590. Dianxiang Xu is the corresponding author.

References

[1] M. Blackburn, R. Busser, A. Nauman, and R. Chandramouli, "Model-based approach to security test automation," in *Proc. Quality Week*, Jun. 2001. Q1

[2] Y. Deng, J. C. Wang, J. Tsai, and K. Beznosov, "An approach for modeling and analysis of security system architectures," *IEEE Trans. Knowl. Data Eng.*, vol. 15, no. 5, pp. 1099–1119, Sep. 2003.

[3] D. F. Ferraiolo and D. R. Kuhn, "Role-based access controls," in *Proc. 15th NIST-NCSC Nat. Comp. Security Conf.* Baltimore, MD, USA, Oct. 1992, pp. 554–563.

[4] H. J. Genrich, "Predicate/transition nets," in *Proc. Adv. Petri Nets 1986, Part I Petri Nets: Central Models their Properties*, 1987, pp. 207–247.

[5] H. Hu and G. Ahn, "Enabling verification and conformance testing for access control model," in *Proc. 13th ACM Symp. Access Control Models Technol.*, 2008, pp. 195–204.

[6] H. Huang and H. Kirchner, "Formal specification and verification of modular security policy based on colored Petri nets," *IEEE Trans. Dependable Secure Comput.*, vol. 8, no. 6, pp. 852–865, Nov./Dec. 2011.

[7] Y. Jia and M. Harman, "An analysis and survey of the development of mutation testing," *IEEE Trans. Softw. Eng.*, vol. 37, no. 5, pp. 649–678, Jun. 2010.

[8] J. Julliand, P.A. Masson, and R. Tissot, "Generating security tests in addition to functional tests," in *Proc. 3rd Int. Workshop Autom. Softw. Test*, 2008, pp. 41–44.

[9] J. Jürjens, "Model-based security testing using UMLsec," *Electron. Notes Theoretical Comp. Sci.*, vol. 220, no. 1, pp. 93–104, Dec. 2008.

[10] K. Knorr, "Dynamic access control through Petri net workflows," in *Proc. 16th Annu. Conf. Comput. Security Appl.*, 2000, pp. 159–167.

[11] K. Li, L. Mounier, and R. Groz, "Test generation from security policies specified in Or-BAC," in *Proc. 31st Annu. Int. Comput. Softw. Appl. Conf.*, 2007, pp. 255–260.

[12] W. Mallouli, J. M. Orset, A. Cavalli, N. Cuppens, and F. A. Cuppens, "A formal approach for testing security rules," in *Proc. 12th ACM Symp. Access Control Models Technol.*, 2007, pp. 127–132.

[13] E. Martin and T. Xie, "Automated test generation for access control policies via change-impact analysis," in *Proc. 3rd Int. Workshop Softw. Eng. Secure Syst.*, May 2007, p. 5.

[14] A. Masood, R. Bhatti, A. Ghafoor, and A. Mathur, "Scalable and effective test generation for role-based access control systems," *IEEE Trans. Softw. Eng.*, vol. 35, no. 5, pp. 654–668, Sep./Oct. 2009.

[15] A. Masood, A. Ghafoor, and A. Mathur, "Conformance testing of temporal role-based access control systems," *IEEE Trans. Dependable Secure Comp.*, vol. 7, no. 2, pp. 144–158, Apr.–Jun. 2010.

[16] B. Meyer, *Object-Oriented Software Construction*. 2nd ed. New York, NY, USA: Prentice-Hall, 1997.

[17] K. H. Mortensen, "Automatic code generation method based on coloured Petri net models applied on an access control system," in *Proc. 21st Int. Conf. Appl. Theory Petri Nets*, 2000, pp. 367–386.

[18] A. C. O'Connor and R. J. Loomis. (2010, Dec.). 2010 economic analysis of role-based access control *RTI International* [Online]. Available: http://csrc.nist.gov/groups/SNS/rbac/documents/20101219_RBAC2_Final_Report.pdf Dec. 2010

[19] J. Park and R. Sandhu, "The UCON ABC usage control model," *ACM Trans. Inform. Syst. Security.*, vol. 7, no. 1, pp. 128–174, 2004.

[20] A. Pretschner, Y. Le Traon, and T. Mouelhi, "Model-based tests for access control policies," in *Proc. 1st Int. Conf. Softw. Testing, Verification, Validation*, Lillehamer, Norway, Apr. 2008, pp. 338–347.

[21] A. Pretschner, W. Prenninger, S. Wagner, C. Kühnel, M. Baumgartner, B. Sostawa, R. Zölch, and T. Stauner, "On evaluation of model-based testing and its automation," in *Proc. 27th Int. Conf. Softw. Eng.*, 2005, pp. 392–401.

[22] R. Sandhu, D. F. Ferraiolo, and D. R. Kuhn, "The NIST model for role-based access control: Towards a unified standard," in *Proc. ACM Workshop Role-Based Access Control*, 2000, pp. 47–63.

[23] B. Shafiq, A. Masood, J. Joshi, and A. Ghafoor, "A role-based access control policy verification framework for real-time systems," in *Proc. 10th IEEE Int. Workshop Object-Orient. Real-Time Dependable Syst.*, 2005, pp. 13–20.

[24] S. D. Stoller, P. Yang, C. R. Ramakrishnan, and M. I. Gofman, "Efficient policy analysis for administrative role based access control," in *Proc. 14th ACM Conf. Comput. Commun. Security*, 2007, pp. 445–455.

[25] M. Utting and B. Legeard, *Practical Model-Based Testing: A Tools Approach*. San Mateo, CA, USA: Morgan-Kaufmann, 2006.

[26] D. Xu, "A tool for automated test code generation from high-level Petri nets," in *Proc. 32nd Int. Conf. Appl. Theory Petri Nets Petri Nets 2011, LNCS 6709*, Newcastle upon Tyne, U.K., Jun. 2011, pp. 308–317.

[27] D. Xu and W. Chu, "A methodology for building effective test models with function nets," in *Proc. IEEE 36th Comput. Softw. Appl. Conf.*, Izmir, Turkey, Jul. 2012, pp. 334–339.

[28] D. Xu and K. E. Nygard, "Threat-driven modeling and verification of secure software using aspect-oriented Petri nets," *IEEE Trans. Softw. Eng.*, vol. 32, no. 4, pp. 265–278, Apr. 2006.

[29] D. Xu, L. Thomas, M. Kent, T. Mouelhi, and Y. Le Traon, "A model-based approach to automated testing of access control policies," in *Proc. 17th ACM Symp. Access Control Models Technol.*, Jun. 2012, pp. 209–218.

[30] D. Xu, M. Tu, M. Sanford, L. Thomas, D. Woodraska, and W. Xu, "Automated security test generation with formal threat models," *IEEE Trans. Dependable Secure Comput.*, vol. 9, no. 4, pp. 525–539, Jul./Aug. 2012.

[31] D. Xu, J. Yin, Y. Deng, and J. Ding, "A formal architectural model for logical agent mobility," *IEEE Trans. Softw. Eng.*, vol. 29, no. 1, pp. 31–45, Jan. 2003.

[32] J. Zander, I. Schieferdecker, and P. J. Mosterman, Eds., *Model-Based Testing for Embedded Systems*. Boca Raton, FL, USA: CRC Press, 2011.

Dianxiang Xu (SM'01) received the BS, MS, and the PhD degrees in computer science from Nanjing University, China. He is a professor at the Department of Computer Science, Boise State University. Prior to joining BSU in 2013, he was at Dakota State University, North Dakota State University, Texas A&M University, and Florida International University. His research interests include software security and safety, access control, software engineering, and software-defined networking. He is a senior member of the IEEE.

Michael Kent received the BS degree in computer science from Dakota State University, South Dakota, in 2013. He is currently a software engineer at SDN Communications, South Dakota.

Lijo Thomas received the MS degree in Information Assurance from Dakota State University, South Dakota, USA in 2011. He is currently a security consultant at Cognizant Technology Solutions.

Tejeddine Mouelhi received the PhD degree from Telecom Bretagne, France, in 2010. He is currently a senior security researcher in itrust consulting. From 2010 until 2014, he was a scientific collaborator at the University of Luxembourg. He has published 25 peer-reviewed conference and journal papers.

Yves Le Traon received the engineering degree and the PhD degree in computer science from the Institut National Polytechnique in Grenoble, France, in 1997. He is a professor of the Interdisciplinary Centre for Security, Reliability and Trust (SnT) and the head of the CSC Research Unit at University of Luxembourg. From 1998 to 2004, he was an associate professor at the University of Rennes, France. His research interests include software testing, model-driven engineering, model-based testing, evolutionary algorithms, software security, security policies and Android security. He has published more than 140 peer-reviewed papers in international conferences and journals.

Query to the Author

Q1. Please provide page range in Ref. [1].