International Journal of Software Engineering
and Knowledge Engineering
Vol. 21, No. 1 (2011) 103–127

DOI: 10.1142/S021819401100513X

World Scientific
www.worldscientific.com

MODEL CHECKING FOR VERIFICATION OF MANDATORY ACCESS CONTROL MODELS AND PROPERTIES

VINCENT C. HU* and D. RICHARD KUHN†

National Institute of Standards and Technology
Gaithersburg, MD 20899-8930, USA
**vhu@nist.gov*
†*kuhn@nist.gov*

TAO XIE‡ and JEEHYUN HWANG§

Department of Computer Science
North Carolina State University
Raleigh, NC 27695-8206, USA
‡*xie@csc.ncsu.edu*
§*jhwang4@csc.ncsu.edu*

Received 30 May 2009
Revised 15 July 2009
Accepted 27 July 2009

Mandatory access control (MAC) mechanisms control which users or processes have access to which resources in a system. MAC policies are increasingly specified to facilitate managing and maintaining access control. However, the correct specification of the policies is a very challenging problem. To formally and precisely capture the security properties that MAC should adhere to, MAC models are usually written to bridge the rather wide gap in abstraction between policies and mechanisms. In this paper, we propose a general approach for property verification for MAC models. The approach defines a standardized structure for MAC models, providing for both property verification and automated generation of test cases. The approach expresses MAC models in the specification language of a model checker and expresses generic access control properties in the property language. Then the approach uses the model checker to verify the integrity, coverage, and confinement of these properties for the MAC models and finally generates test cases via combinatorial covering array for the system implementations of the models.

Keywords: Access control; policy; model; testing.

1. Introduction

Mandatory access control (MAC) [1] is concerned with determining the allowed activities of legitimate users, mediating every attempt by a user to access a resource in a system. A given information technology (IT) infrastructure can implement MAC systems in many places and at different levels. Operating systems use MAC to protect files and directories. Database management systems (DBMS) apply MAC to

regulate access to tables and views. Most commercially available application systems implement MAC, often independent of the operating systems and/or DBMSs on which they are installed.

The objectives of a MAC system are often described in terms of protecting system resources against inappropriate or undesired user access. From a business perspective, these objectives could just as well be described in terms of optimal sharing of information. After all, the main objective of IT is to make information available to users and applications. A greater degree of sharing may get in the way of resource protection; in reality, a well-managed and effective MAC system actually facilitates sharing. A sufficiently fine-grained MAC mechanism can enable selective sharing of information where in the absence of MAC, sharing may be considered too risky altogether [2].

When planning a MAC system, three abstractions of controls should be considered: MAC policies, models, and mechanisms. MAC *policies* are high-level requirements that specify how access is managed and who, under what circumstances, may access what information. While MAC policies can be application-specific and thus taken into consideration by the application vendor, policies are just as likely to pertain to user actions within the context of an organizational unit or across organizational boundaries. For instance, policies may pertain to resource usage within or across organizational units or may be based on need-to-know, competence, authority, obligation, or conflict-of-interest factors. Such policies may span multiple computing platforms and applications.

At a high level, MAC policies are enforced through a *mechanism* that translates a user's access request, often in terms of a structure that a system provides. There are a wide variety of structures; for example, a simple table lookup can be performed to grant or deny access. Although no well-accepted standard yet exists for determining their policy support, some MAC mechanisms are direct implementations of formal MAC policy concepts [2].

Rather than attempting to evaluate and analyze MAC systems exclusively at the mechanism level, security models are usually written to describe security properties of a MAC system. A model is a formal presentation of a security policy enforced by the MAC system, and is useful for proving theoretical limitations of a system. MAC models are of general interest to both users and vendors. They bridge the rather wide gap in abstraction between policies and mechanisms. MAC mechanisms can be designed to adhere to the properties of the model. Users see a MAC model as an unambiguous and precise expression of requirements. Vendors and system developers see MAC models as design and implementation requirements. On one extreme, a MAC model may be rigid in its implementation of a single policy. On the other extreme, a MAC model allows for the expression and enforcement of a wide variety of policies and policy classes [2, 3].

It is common that a system's privacy and security are compromised due to the faulty MAC model and mechanism of MAC policies instead of the failure of cryptographic primitives or protocols. Such faults can result in serious

vulnerabilities, especially when different MAC models and rules are combined. This problem becomes increasingly severe as systems become more and more complex, and are deployed to manage a large amount of sensitive information and resources that are organized into sophisticated structures. Identifying discrepancies between policy, model, and implementation is crucial because correct implementation and enforcement of policies by applications is based on the premise that the policy specifications are correct, therefore the policy specification must undergo rigorous verification and validation through systematic verification and testing to ensure that they truly encapsulate the desired MAC properties from the policy authors.

To the best of our knowledge, no techniques exist for verifying whether the properties of a MAC policy are correctly expressed in a model as well as whether the policy is satisfied in the implementation. In practice, the same MAC policies may express multiple different MAC models or express a single model in addition to extra access control (AC) constraints outside of the model. Verifying the conformance of MAC models and policies is a non-trivial and critical task. One important aspect of such verification is to formally check the inconsistency and incompleteness [4–7] of the model and properties because a MAC model and its implementation do not necessarily explicitly expressed the policy, which can also be implicitly embedded by mixing with direct access constraints or other MAC models.

In our approach, users first specify MAC models in the specification language of a model checker, and properties in temporal logic formula. To ensure the correctness of the MAC model against the properties, the system automatically verifies these properties by exploiting the verification process of the model checker. In this process, the confidence of the model's correctness depends on the quality of the specified properties. Next, the system assesses the quality of given properties based on mutation analysis and checks the entities (i.e., rules) of the model are sufficiently covered and confined by the properties. The assessment result help guide high quality property specification by targeting uncovering or unconfined entities. Finally, the system automatically generates test cases (both test inputs and expected outputs) from the domain variables in the MAC model and specified properties using a combinatorial testing technique [9]. The system feeds these test inputs into real MAC implementation of the given model to verify whether the actual test outputs are the same as the expected outputs.

The rest of this paper is organized as follows. Section 2 presents a formal model of access control model checking. Section 3 describes three fundamental (static, dynamic, and historical) MAC models and their properties, which are expressed in finite state machine specification and in temporal logic formula, respectively. Section 4 describes rule coverage and property confinement checking techniques. Section 5 illustrates a combinatorial test suite generation technique. Section 6 describes our test scheme for model checking, property assessment, and testing. Section 7 illustrates the case study to demonstrate our test scheme. Section 8 discusses related work. Section 9 concludes the paper.

2. Access Control Model Checking

There are two levels of verification steps. First, the correct specification of a MAC model needs to be verified. To achieve this goal, the scheme of our approach includes a *black-box* model checking method that allows the users to specify AC properties and then verifies the MAC model against these properties. Since the confidence of the model's correctness depends on the quality of the specified properties, our scheme also includes a *white-box* property assessment method that applies mutation analysis [8] on entities in the model and properties to assess the sufficiency of the *covering* and *confinement* of the properties for the model. Second, the correct implementation of the policy needs to be tested. Our scheme includes a test generation method that generates test cases (both test inputs and expected outputs) from the AC variables in the model and specified properties using a combinatorial testing technique [9]. The approach then runs these test cases on the MAC implementation to verify whether the actual test outputs are the same as the expected outputs. We next provide the formal definition of MAC model checking in terms of AC attributes:

Let S, O, and A denote respectively the set of all the subjects, objects, and actions in a MAC system. Each subject, object, or action is associated with a set of attributes that may be used for AC decisions. For example, a subject's attributes may include a user's role, rank, and security clearance. An object's attributes may include a file's type, a document's security class, and a printer's location.

Definition 1. A MAC *rule* r is a statement: "if c then d", where *constraint* c is a predicate expression on AC attributes (subjects, objects, or actions) and system states (global system events) for the permission decision d. An example rule is "***if*** (*a user is a member of X group with security level* 3 ***and*** *today is Friday* ***and*** *the user's action is read* ***and*** *the object is file Y*) ***then*** *grant*".

Definition 2. An AC *property* p is a proposition: "$b \rightarrow d$" where the result of the access permission d depends on **quantified** predicate b on AC attributes and system states. An example property is "***for all*** *users whose security level is* 2 ***and*** *the action is read* ***and*** *the object is file* $Y \rightarrow$ *deny*".

An access request q is a tuple (s, o, a), where $s \subset S$, $o \subset O$ and $a \subset A$. A request (s, o, a) means that subject s requests to take action a on object o. Note that each of s, a, or o may have multiple attributes. A MAC model is a sequence of rules, each of which is of the form (*sCond*, *oCond*, *aCond*, *decision*, *gCond*, *st*) in the logic expression of c in Definition 1. *sCond*, *oCond* and *aCond* are constraints over the attributes of a subject, object, and action, respectively. *gCond* is a general constraint that may potentially be over all the attributes of subjects, objects, actions, and other properties of a system (e.g., the current time and the load of a system), and s is the current state recorded from the previous access event of the MAC system. Given a request (s, o, a), if *sCond*, *oCond*, *aCond*, *gCond*, and *st* are all evaluated to be TRUE, then the request is either permitted or denied according to the decision d as described in the rule of Definition 1. Thus, each rule's applicability to a request

is of the form: *If* $(sCond \wedge oCond \wedge aCond \wedge gCond \wedge st)$*then* (d). We can specify more complex constraint structures in a rule. For example, we can specify rules that can be applicable to multiple-attributes requests.

A deterministic finite state transducer of a MAC model corresponding to a Finite State Machine (FSM) with a five-tuple $M = (\Sigma, ST, s_0, \delta, F)$, where:

$$\Sigma = \{sCond_1, \ldots, sCond_n, aCond_1, \ldots, aCond_n, oCond_1, \ldots, oCond_n, gCond_1, \ldots, gCond_n\}$$

is the input alphabet that represents the attribute constraints associated with subject s, access a, object o, and global event g.

$S = \{st_0, st_2, \ldots, st_n, Grant, Deny\}$ is a finite, non-empty set of recorded MAC system states and permissions. st_0 is the initial state.

δ is the state-transition function, where δ: $ST \times \Sigma \rightarrow ST$

$F = \{Grant, Deny\}$ is the set of final states.

For **static** MAC models [10] such as Multi-Level AC (MLS) [11], Role-Based AC (RBAC) [12], and Rule-Based AC policies (RuBAC), the FSM M_{static} does not require intern states st to reach the permission state, thus $F = ST = \{Grant, Deny\}$, i.e., M_{static} is just a straightforward FSM model without state transitions. For **dynamic** MAC models such as N-Person Control [13], and Limited_Number_of_Access policies, the input alphabets of FSM M_{dynamic} are $\Sigma_{\text{dynamic}} = \{gCond_1, \ldots, gCond_n\}$, where $gCond_i$ is the threshold indicator of the access limitation, such as the number of persons have to access at the same time in a N-Person control policy, or the maximum number of access allowed for Limited_Number_of_Access policy. For **historical** MAC models such as Chinese Wall [14] and Workflow policies [15], the input alphabets of the FSM $M_{\text{historical}}$ are $\Sigma_{\text{historical}} = \Sigma - \{gCond_1, \ldots, gCond_n\}$, where $sCond_i$, $aCond_i$, and $oCond_i$ contribute to a historical recording that is used as determining factors for the next permission decision. Note that it is possible for different types of MAC models to combine into one model such that $M_{\text{combine}} = \{M_{\text{static}} \cup M_{\text{dynamic}} \cup M_{\text{historical}}\}^2$.

An AC property p in Definition 2 as expressed by the proposition p: $ST \times \Sigma^2 \rightarrow ST$ of FSM, which can be collectively translated in terms of logical formula such that $p = (s_i * sCond_1 * \cdots * sCond_n * aCond_1 * \cdots * aCond_n * oCond_1 * \cdots * oCond_n * gCond_1 * \cdots * gCond_n) \rightarrow d$, where $p \in P$ is a set of properties, and $*$ is a Boolean operator in terms of logical formulas of temporal logic such as computational tree logic (CTL) [16, 17] and linear-time temporal logic (LTL) [18]. The purpose of model checking is to verify the set ST in M in which p is true according to an exhaustive state space search. In addition, by verifying the set of states in which the negation of p is true, we can obtain the set of counterexamples to make the assertion that p is true. The satisfaction of a MAC model M to the AC properties P by model checking is composed of two requirements:

(1) Safety, where M satisfies P in description of safety. That is, there is no violation of rules to the logic specified in P, and it is assured that M will eventually be in a desired state after it takes actions in compliance with a user access request. Thus,

Axiom 1. AC **safety** verification is to verify M_{acp} **satisfies** P_{acp} of MAC policy *acp*.

(2) Liveness, where M will not have unexpected complexities. That is, there is neither a deadlock in which the system waits forever for system events, nor a livelock in which the model repeatedly executes the same operations forever. Thus,

Axiom 2. Liveness check on M calculates the complexity to prove that the model is practical, i.e., liveness of M_{acp} is that P_{acp} will be satisfied within finite states (a permission decision of AC request will be eventually made) for policy *acp*.

Figure 1 shows the relations between M and P in a model checking framework.

The AC rules define the system behaviors that function as the transition relation δ in M. Then when the AC property is represented by temporal logic formula p, we can represent the assertion that model M satisfies p by $M| = Ab \rightarrow AXd$ from Definition 2 using temporal logic quantifier A to represent "always", and logic quantifier X to represent "is true next state". The purpose of safety verification (Axiom 1) and liveness verification (Axiom 2) using model checking is to determine whether these assertions are true, and to identify a state in which the assertions are not true as a counterexample for the assertions. Since the behavior of the MAC system can be represented by FSM M, and the properties that M must satisfy can be represented by temporal logic formulas, we can define the correctness of policies more precisely as that the model can be led from every possible state that is reachable from initial states to the defined final state while complying with the properties [19].

3. Generic Access Control Properties

This section demonstrates the three fundamental (static, dynamic, and historical) MAC models and properties from the separation of duty and safety point of view [10, 20]. We also illustrate how a model and its properties can be specified in a model checking environment.

3.1. *Static models*

Static policies regulate the access permission by static system states or conditions such as rules, attributes, and system environments (times and locations for access). Popular MAC models with these types of properties include RBAC, MLS,

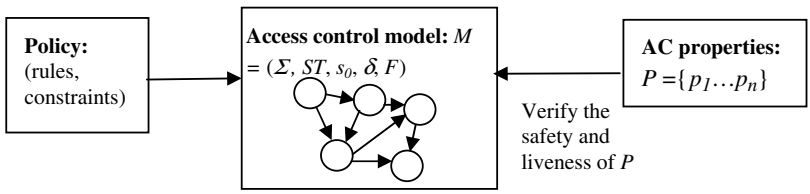

Fig. 1. Mandatory Access Control model and property.

and RuBAC. These types of models can be specified by **asynchronous** or **direct** specification expressions of an FSM. The transition relation of authorization states is directly specified as a propositional formula in terms of the current and next values of the state variables. Any current state/next state pair is in the transition relation if and only if it satisfies the formula, as demonstrated in the following direct specification of an FSM:

```
{
  VARIABLES
    access_state : boolean; /* 1 as grant, 0 as deny*/
    ..........
  INITIAL
    access_state := 0;
  TRANS /* transit to next access state */
    next (access_state) :=
      ((constraint_1 & constraint_2 & ..... constraint_n) |
      (constraint_a & constraint_b & ..... constraint_m) ........);
}
```

where the system state of access authorization is initialized as the *deny* state and moved to the *grant* state for any access request that complies with the constraints of the rule corresponding with each constraint predicate (i.e., *constraint_1*. . . .& *constraint_n*) in a rule, and stay in the *deny* state otherwise. The properties of the static constraints can be verified using the properties expressed in the following temporal logic formulae:

$AG\,(constraint\_1\ \&\ constraint\_2\ \&\ \ldots.\ constraint\_n) \rightarrow AX\,(access\_state = 1)$
$AG\,(constraint\_a\ \&\ constraint\_b\ \&\ \ldots.\ constraint\_m) \rightarrow AX\,(access\_state = 1)\ \ldots\ldots$
$AG\,!\,((constraint\_1\ \&\ \ldots.constraint\_n)\,|\,(constraint\_a\ \&\ \ldots.\ constraint\_m)\,|\ldots) \rightarrow AX\,(access\_state = 0)$

which simply means that all access requests that comply with specified constraints for the rules should be granted, and all non-compliant ones should be denied. Specifications of the form "$AG\,(b) \rightarrow AX\,(d)$" (Definition 2) indicate essentially that for all paths (the "A" in "AG") for all states globally (the "G"), if b holds then ("$\rightarrow$") for all paths, in the next state (the "X" in "AX") d will hold.

3.2. *Dynamic models*

Dynamic policies regulate the access permission by dynamic system states or conditions such as specified events or system counters or N-person AC policy. A MAC model with these types of properties specifies that accesses are permitted only by a certain subject to a certain object with certain limitations (e.g., object x can be accessed only no more than i times simultaneously by user group y). For example, if a user's role is a *cashier*, he or she cannot be an *accountant* at the same time when handling a customer's checks. This type of model can be specified with

asynchronous or **direct specification** expressions of an FSM, which uses a variable semaphore to express the dynamic properties of the authorization decision process. Another example of dynamic constraint states is enforcing a limited number of concurrent accesses to an object. The authorization process for a user thus has four states: *idle*, *entering*, *critical*, and *exiting*. A user is normally in the *idle* state. The user is moved to the *entering* state when the user wants to access the critical object. If the limited number of access times is not reached, the user is moved to the *critical* state, and the number of the current access is increased by 1. When the user finishes accessing the critical object, the user is moved to the *exiting* state, and the number of the current access is decreased by 1. Then the user is moved from the *exiting* state to the *idle* state. The authorization process can be modeled as the following asynchronous FSM specification:

```
{
  VARIABLES
    count, access_limit : INTEGER;
    request_1 : process_request (count);
    request_2 : process_request (count);
    . . . . . .
    request_n: process_request (count);
    /*max number of user requests allowed by the system*/
    access_limit := k; /*max number of concurrent access*/
    count := 0; act {rd, wrt}; object {obj};
    process_request (access_limit) {
      VARIABLES
        permission : {start, grant, deny};
        state : {idle, entering, critical, exiting};
      INITIAL_STATE (permission) := start;
      INITIAL_STATE (state) := idle;
      NEXT_STATE (state) := CASE {
        state == idle : {idle, entering};
        state == entering & ! (count > access_limit): critical;
        state == critical : {critical, exiting};
        state == exiting : idle;
        OTHERWISE: state};
      NEXT_STATE (count) := CASE {
        state == entering : count +1;
        state == exiting : count −1;
        OTHERWISE: DO_NOTHING };
      NEXT_STATE (permission) := CASE {
        (state == entering) & (act == rd) & (object == obj): grant;
        OTHERWISE: deny;
        }
    }
}
```

The state variables of the preceding example are used as the asynchronous states for the concurrent access of the limited number of access requests. The specification of the dynamic constraints is verified through the following properties expressed in temporal logic formula:

$AG\ (state == entering)\ \&\ (act == rd)\ \&\ (object == obj) \rightarrow AX\ (access = grant)$

$AG\ (state == idle \mid state == critical \mid state == exiting) \rightarrow AX\ (access = deny)$

where temporal logic formula $AG\ (b) \rightarrow AX\ (d)$ (Definition 2) indicates that "if condition p is true at time t, condition d is true at all times later than t.

3.3. *Historical models*

Historical policies regulate the access permission by historical access states or recorded and predefined series of events. The representative MAC policies for this type of AC model are Chinese Wall and Workflow AC policies. This type of model can be best described by **synchronous** or **direct specification** expressions of an FSM. For example, the following synchronous FSM specification specifies a Chinese Wall AC model where there are two Conflict of Interest groups COI_1, COI_2 of objects:

```
{
  VARIABLES
    access {grant, deny};
    act {rd, wrt};
    o_state {none, COI1, COI2};
    u_state {1, 2, 3};
  INITIAL_STATE(u_state) := 1;
  INITIAL_STATE(o_state) := none;
  NEXT_STATE(state) := CASE {
    u_state == 1 & act == rd & o_state == COI1: 2;
    u_state == 1 & act == rd & o_state == COI2: 3;
    u_state == 2 & act == rd & o_state == COI1: 2;
    u_state == 2 & act == rd & o_state == COI2: 2;
    u_state == 3 & act == rd & o_state == COI1: 3;
    u_state == 3 & act == rd & o_state == COI2: 3;
    OTHERWISE: 1; };
  NEXT_STATE(access) := CASE {
    u_state == 2 & act == rd & o_state == COI1: grant;
    u_state == 3 & act == rd & o_state == COI2: grant;
    OTHERWISE: deny; };
    NEXT_STATE (act) := act;
    NEXT_STATE (o_state) := object;
}
```

The properties of the dynamic constraints can be verified by verifying the following temporal logic formula:

$AG\,((u\_state == 2\ \&\ act == rd\ \&\ o\_state == COI_1)\,|\,(u\_state == 3\ \&\ act == rd\ \&\ o\_state == COI_2)) \rightarrow AX\,(access = grant)$
$AG\,!\,((u\_state == 2\ \&\ act == rd\ \&\ o\_state == COI_1)\,|\,(u\_\ state == 3\ \&\ act == rd\ \&\ o\_state == COI_2)) \rightarrow AX\,(access = deny)$

where temporal logic $AG\ (b) \rightarrow AX\ (d)$ indicates that the access event d is invocated by historical events in b.

4. Coverage and Confinement Checking

Although the integrity of logic in MAC model can be checked by the safety and liveness verification (Section 2), the MAC models are still not fault-proof because the temporal logic in the properties might not be thorough in covering all possible values of all rules or all conditions in rules. For example, an extra permit rule may be added to a list of rules specified for a MAC model, and the constraint of this rule may not be included in any of the properties; therefore, the unauthorized access allowed by this extra rule cannot be exposed by only the safety and liveness verification, thus leading to a fault due to insufficient properties (i.e., **coverage** fault). Further, even if the properties cover all the rules in the model, it is possible that the properties do not completely **confine** to intended properties: the complement of a specified predicate does not guarantee results to the complement of the permission of a property, thus risking exceptional permissions despite the constraints enforced by the property. The rules in the model, properties, and confined properties may each describe its own space of permission conditions, and may not be congruent in one space as the initial relation illustrated examples in Fig. 2. The safety and liveness check can assure only the logic integrity of some rules against some properties. The complete satisfaction of a model to its properties requires fixing of coverage and confinement faults if any spotted by additional Coverage and Confinement Check (CCC), the second line of defense against such semantic faults.

CCC requires mutant versions of the model [21], and extra modified properties for additional model checking. As illustrated in Fig. 2, the goal of CCC is to ensure that the rules in the model are completely covered by the properties, and to confirm that no exceptional access permissions are granted unless intentionally allowed. The first step of CCC is to discover the rules, which are seeped through the specification of the properties by applying model checking on mutated versions of rules. The second step is to detect unexpected access permission that might not be the intention of the policy author by applying model checking on modified properties extracted from the original properties. The preceding steps are described in Secs. 4.1 and 4.2 after the following formal definitions.

Axiom 3. A MAC rule r is **covered** by an AC property p when the access decision d of p depends on r of the MAC model, verified through safety and liveness checking

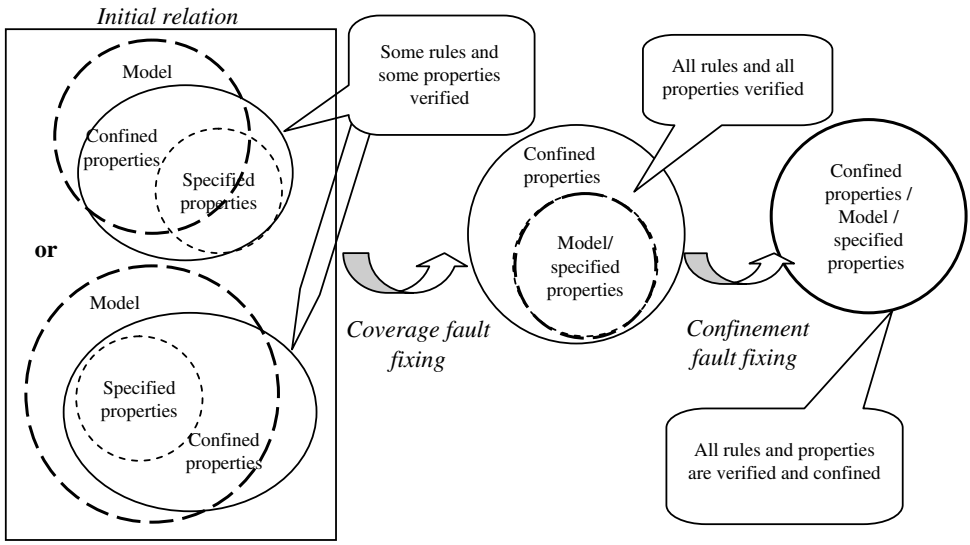

Fig. 2. Model, confined properties, and specified properties.

without counterexamples of r against p. Function $C_M(\ldots r_i \ldots p_i \ldots) = \text{TRUE} \mid \text{FALSE}$ decides if rule r_i in the model M is covered by property p_i, where r_i is a member of rule set R, and p_i is a member of property set P. For example, the rule: "*users with security level* 3 *can read file* Y" is covered by the property: "*For all users with security greater then* 2 *can read file* Y".

Definition 3. The **negation** of a MAC rule r, $\sim r$ = "if c then $\neg d$" from Definition 1. For example, the negation of the example rule in Definition 1 is "***if*** (*a user is a member of X group with security level* 3 ***and*** *today is Friday* ***and*** *the user's action is read* ***and*** *the object is file* Y) ***then*** *deny*".

Definition 4. The **complement** of an AC property $p = b \rightarrow d$, $p' = \neg b \rightarrow \neg d$, is the complement expression of p, i.e., the negation of b causes the negation of d. For example, $p = (x \wedge y \wedge z) \rightarrow$ *grant*, $p' = \neg(x \wedge y \wedge z) \quad \rightarrow$ *deny*. For example, the complement of the example property in Definition 2 is "***for all*** *users whose security level is not* 2 ***or*** *the action is not read* ***or*** *the object is not file* $Y \rightarrow$ *grant*".

4.1. *Rule coverage checking*

The key notion of rule coverage checking is to synthesize a version of the given model in such a way that the permission of its rules is mutated such that rule$r$is changed to $\sim r$. If property set P is satisfied by both mutated and original models of M through model checking, then some of the rules and their mutants would never apply to P; in other words, P does not cover all the rules in model M. Formally:

Theorem 1. If $(C_M(r, p) \wedge C_M(\sim r, p))$ then "r is not applied to properties p".

Proof. $C_M\ (r, p)$ = TRUE says that p depends on rule r to reach the access decision (Axiom 1). $C_M\ (\sim r, p)$ = TRUE says that p depends on rule $\sim r$ to reach the access decision d, since r = "if c then d" and $\sim r$ = "if c then $\neg d$"(Definitions 1 and 3), which leads to $C_M\ (r, \sim r, p)$ = TRUE, i.e., p depends on both r and $\sim r$ for d. The only condition for this result to hold is when r is a "**don't care**" variable in the Boolean predicate of p; in other words, r is not covered by p.

As an example in Fig. 3, the safety and liveness checking verify that p conforms to the model without counterexamples; however, by applying the CCC by mutating the rule $u == j : grant$ to $u == j : deny$ for the coverage checking, the result shows that the property satisfies the mutated rules as well (without counterexamples), indicating that the variable u in the rule r was never applied to the property p.

This result shows that the rule $u == j : grant$ is not verified with the property $AG\ (q == i) \rightarrow access = grant$. One way for addressing this insufficiency is adding a new property that describes proper control of u. Note that it is necessary to check every r in M against the set of all properties P to achieve thorough verification. □

4.2. *Property confinement checking*

Property confinement checking ensures that there is no exceptional permission allowed in addition to the specified properties; this checking requires a confined property p' (Definition 4) modified from the original property p to be added for the next run of model checking. Confinement check should discover the discrepancy of the specified properties and the properties the MAC policy author intend. The rationale is that if model M does not satisfy p', then there are exceptional access permissions that leak through p, formally:

Theorem 2. If $(C_M\ (r, p) \wedge C_M\ (r, p'))$ then there is no exceptional permission allowed from p in model M against rule r.

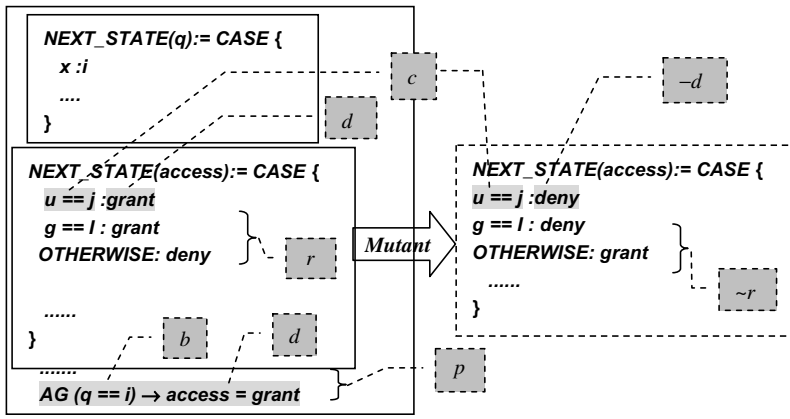

Fig. 3. Example of uncovered rules in a MAC model.

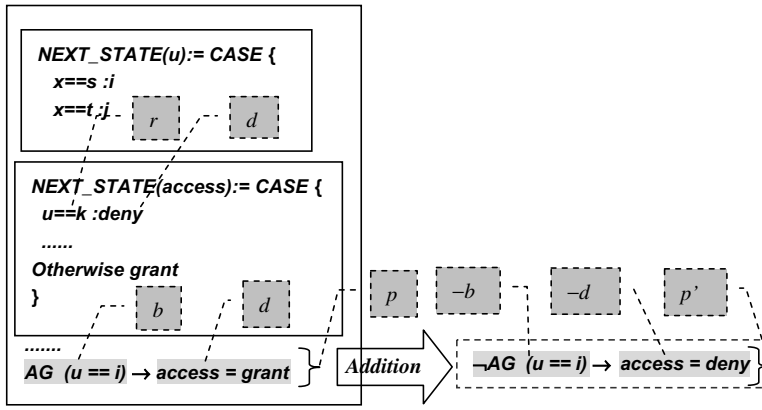

Fig. 4. Unconfined rule in a property.

Proof. $C_M(r,\ p)$ says that p is covered by model M with rule r, and $C_M\ (r,\ p')$ says that p' is covered by rule r (Axiom 3), since $C_M\ (r,p) \wedge C_M(r,p')$ equaling to $C_M\ (r,\ p,\ p')$ implies that r is covered by both p and p', such that p: $r \rightarrow d$ and p': $\neg r \rightarrow \neg d$, which means any rule that is a negation of r will cause permission d changes to $\neg d$.

Figure 4 shows a transition to an unspecified state for a certain range of data values that allow exceptional permission not covered by a specified property because the value of *access* when u value is different than i (such as $u = j$) also grants access permission by the rule *otherwise : grant*. This fault can be caught by a counterexample *AG (u == j) → access = grant* when checking the model M against the additional confinement property ¬ *AG (u == i) → access = deny* derived from original property *AG (u == i) → access = grant*. The additional model checking for confinement verification informs the MAC policy authors which property is not confined so that the MAC policy author can add new rules to enforce the safety of the model. As in this case, changing the rule *otherwise : grant* to *otherwise : deny* and adding all granted rules in the state will correct the problem. □

Note that it is possible the MAC policy author intentionally allowed the exception for a specified property, and it is necessary to check every p against the set of rules $R = \{r_1, \ldots, r_n\}$ to achieve thorough verification.

5. Test Suite Generation

As testing must always be conducted once a policy is implemented to assure correct implementation, automated generation of test cases can reduce total costs, thus making formal specification easier to integrate into the development process in addition to supporting property verification. Model checking is ideal for this integration because it can solve the oracle problem for testing (determining expected

results for a particular set of test input data). A case study of this technique for software is given by [22]. Even with highly automated tools, real-world development budgets rarely allow the development and exploration of formal models, because the cost must be balanced against the cost of releasing code with faults that would not be caught in testing. But testing typically consumes 50% or more of a development budget. Generating test cases from formal specifications makes it cost-effective to allocate a portion of the testing budget to produce a formal specification, which can then be used to confirm desired properties and generate test cases.

Combinatorial testing is a methodology that tests all t-way [23] combinations of input parameter values. For n variables with v values, t-way combinations, combinatorial testing requires a number of tests proportional to $v^t \log n$, which is enormous to be practical if t is a large enough number; however, the most common form is pairwise testing, in which all pairs of input values are covered in at least one test. Higher strength versions of this method cover 3-way, 4-way, or more interactions at least once. The advantage of combinatorial testing for verifying MAC policies is that AC often relies on a small number of discrete values for most parameters. For example, an MLS policy (i.e., standard military classification policy) may have levels unclassified, confidential, secret, top secret, plus a small number of categories, all applied to a collection of resources such as files and programs. While real-world MAC is likely to have far too many variables for exhaustive testing, it will probably be possible to test, for example, all 5-way combinations of variable values. Thus a failure that results from the interaction of five or fewer variables is likely to be caught. The number of tests required to provide 5-way coverage may be large, but if complete tests are fully automated, then this form of testing is practical even for large systems.

The first step in combinatorial testing of the policy is to find a set of tests that will cover all t-way combinations of parameter values for the desired combinatorial interaction strength t. This collection of tests is known as a *covering array*. The covering array specifies test data, where each row of the array can be regarded as a set of parameter values for an individual test. Collectively, the rows of the array cover all t-way combinations of parameter values. An example is given in Fig. 5, which shows a 3-way covering array for 10 variables with two values each. The interesting property of this array is that any three columns contain all eight possible values for three binary variables. For example, taking columns F, G, and H, we can see that all eight possible 3-way combinations (000,001,010,011,100,101,110,111) occur somewhere in the rows of the three columns. In fact, this is true for any three columns. Collectively, therefore, this set of tests will exercise all 3-way combinations of input values in only 13 tests, as compared with 1024 for exhaustive coverage. Similar arrays can be generated to cover up to all 6-way combinations. A non-commercial research tool called Automated Combinatorial Testing Suite (ACTS) [24] developed by NIST and the University of Texas at Arlington makes this possible with much greater efficiency than previous tools. For example, a commercial tool

	A	**B**	**C**	**D**	**E**	**F**	**G**	**H**	**I**	**J**
1	0	0	0	0	0	0	0	0	0	0
2	1	1	1	1	1	1	1	1	1	1
3	1	1	1	0	1	0	0	0	0	1
4	1	0	1	1	0	1	0	1	0	0
5	1	0	0	0	1	1	1	0	0	0
6	0	1	1	0	0	1	0	0	1	0
7	0	0	1	0	1	0	1	1	1	0
8	1	1	0	1	0	0	1	0	1	0
9	0	0	0	1	1	1	0	0	1	1
10	0	0	1	1	0	0	1	0	0	1
11	0	1	0	1	1	0	0	1	0	0
12	1	0	0	0	0	0	0	1	1	1
13	0	1	0	0	0	1	1	1	0	1

Fig. 5. 3-way covering array for 10 parameters with 2 values each.

required 5400 seconds to produce a less optimal test set than ACTS generated in 4.2 seconds.

To produce test cases that guarantee combinatorial coverage to an interaction level t, we produce a t-way covering array [22] for input parameters used in the policy. Informally, a covering array can be viewed as a table of input data where each column is an input parameter and values in each column are parameter values, so that each row represents a test. All possible t-way combinations of parameter values are guaranteed to be covered at least once. If $t = 2$, this procedure results in the familiar "pairwise" testing, but using new algorithms, we are able to produce covering arrays up to strength $t = 6$.

Two *specification claims* in forms of properties are generated for each covering array row, one for result *grant* and one for result *deny*. Values v_{ij} are taken from row $_i$, column $_j$ of the covering array, for all rows.

$$AG\ (p_1 = v_{11} \,\&\, \ldots \,\&\, p_n = v_{1n}) \rightarrow AX\ !(access\_state = grant)$$
$$AG\ (p_1 = v_{21} \,\&\, \ldots \,\&\, p_n = v_{2n}) \rightarrow AX\ !(access\_state = grant)$$
$$\ldots\ldots$$
$$AG\ (p_1 = v_{n1} \,\&\, \ldots \,\&\, p_n = v_{nn}) \rightarrow AX\ !(access\_state = grant)$$
$$AG\ (p_1 = v_{11} \,\&\, \ldots \,\&\, p_n = v_{1n}) \rightarrow AX\ !(access\_state = deny)$$
$$AG\ (p_1 = v_{21} \,\&\, \ldots \,\&\, p_n = v_{2n}) \rightarrow AX\ !(access\_state = deny)$$
$$\ldots\ldots$$
$$AG\ (p_1 = v_{n1} \,\&\, \ldots \,\&\, p_n = v_{nn}) \rightarrow AX\ !(access\_state = deny)$$

For a covering array with n rows, a total of $2n$ specification claims will thus be produced, one *grant* and one *deny* for each row of the covering array. In the claims, possible results *grant* or *deny* are negated. For each claim, if this set of values cannot in fact lead to the particular result, the model checker indicates that this is true. If the claim is false, the model checker indicates so and provides a *counterexample*

with a trace of parameter input values and states that will prove it to be false. The model checker thus filters the claims that we have produced so that a total of n test inputs are generated. In effect, each one is a test case, i.e., a set of input parameter values and expected result. It is then simple to map these values into test cases in the syntax needed for the system under test. When interaction testing is done today, t is nearly always 2 (i.e., pairwise testing) because higher strength interactions require exponentially more test cases. Thus, higher strength interaction testing requires fully automated generation of test input data and expected results, which is made possible through model checking.

This technique makes it possible to produce two complementary types of test cases. In addition to combinatorial test cases, fault-based testing can be automated. By inserting particular faults in the specification, then generating counterexamples using the model checker, we can produce test cases that will detect these faults or faults that are subsumed by them.

6. Test Scheme

A generic test scheme for MAC models and properties verification can be constructed. The scheme starts by expressing MAC models in the specification language of a model checker, and the AC properties in temporal logic formula. Then the system verifies these properties by exploiting the verification process of the model checker. Next, another run of model checking with mutated rules and modified properties guarantees that the rules are covered and confined by the properties. Finally, test cases consisting of input data and expected results are created by applying the covering array generated from the combinatorial array generation function to model checking with the sufficient properties. One goal of the techniques in this approach is to reduce overall software assurance costs by integrating verification with test generation.

The scheme in Fig. 6 contains four major functions implementing the previously described mechanisms. The function Model Checking checks the MAC model against the specified AC properties, including three such checks. The first is phase safety and liveness verification, which ensures that the specified properties are satisfied by the model. The second is phase verification, which rectifies the differences between the MAC rules and properties in terms of coverage and confinement through the Coverage and Confinement Mutation function. When the results report uncovered entities, the users further modify/add new properties or rules to amend the discrepancies. The last check, phase of model checking takes the covering array generated by the Covering Array generator and integrates the array variables into generic deny and grant properties for detecting counterexamples against properties resulted from the second phase. The counterexamples are then fed into the Test Case generator to produce the test cases (both test inputs and their expected outputs). These test cases running on the MAC implementation can comprehensively cover the behavior and verify whether the actual test outputs are the same as the expected outputs.

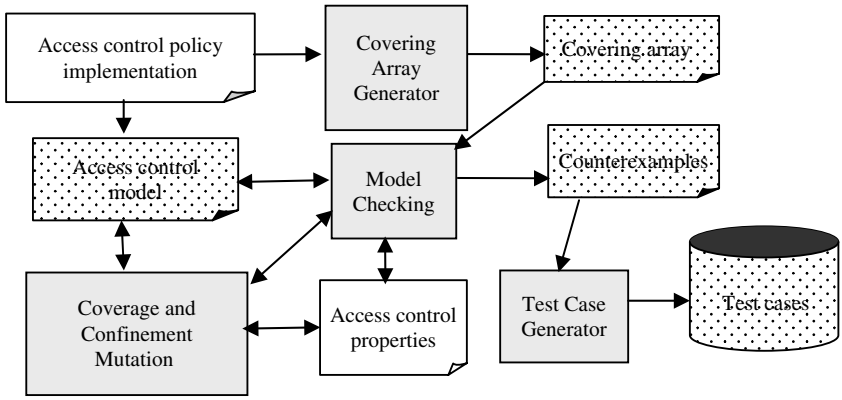

Fig. 6. Scheme for MAC model/AC properties testing.

7. Case Study

We developed a tool called Access Control Policy Testing System (ACPTS). The tool helps a user specify policy models and their properties. ACPTS integrates NuSMV [25] for symbolic model checking and ACTS for generating combinatorial tests. From ACTS, the covering array specifies test data, where each row of the array can be regarded as a set of parameter values for an individual test. Collectively, the rows of the covering array cover all t-way combinations of parameter values for incorporating into Symbolic Model Verifier (SMV) property specifications that can be processed by the NuSMV model checker.

In this study, we used a simple grading RBAC access control policy model composed by ACPTS. We also describe its property set for verification. The policy model and its property set are converted into NuSMV model and verified whether its property set is satisfied. We then perform covering array generation for combinatorial tests, mutant rule verification for detecting for detecting insufficient rule coverage by a specified property set, and mutant property verification to detect the discrepancy of the specified properties and the properties that the MAC policy author intend.

7.1. *Model specification in ACPTS*

A policy author can edit (i.e., add, delete, and modify) RBAC, Multi-Level security, and Workflow policy models [26] and their properties using the tool. The top-left window in Fig. 7 shows specified policy models as a tree structure. The top-right window provides a working area for the policy author to edit a selected model. In Fig. 7, the policy author specifies an RBAC policy with a set of roles (i.e., Faculty and Student), user-role relations (i.e., Jane is Faculty and Jim is Student), and roles' permissions (e.g., Faculty can write grades and Student cannot write grades).

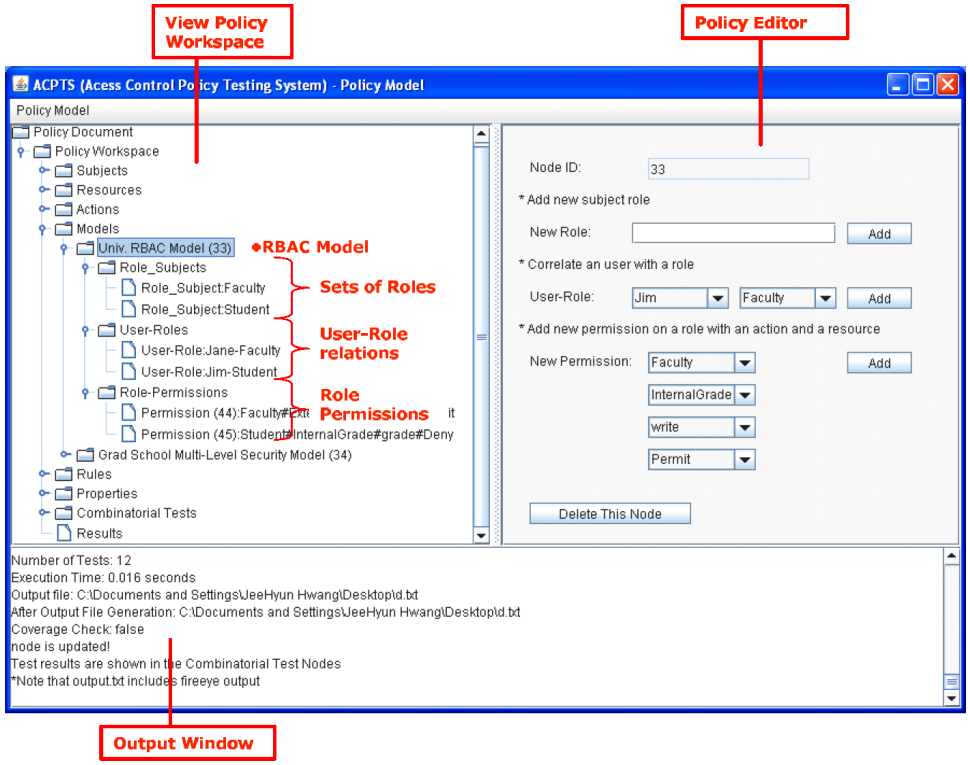

Fig. 7. An example RBAC policy model using ACPTS.

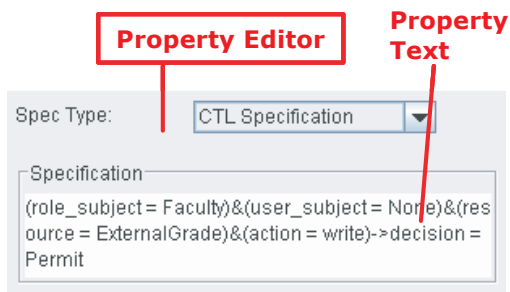

Fig. 8. An example property specified in ACPTS.

As shown in Fig. 8, the property describes the conditions for permitting a Faculty to write grades. Note that the policy author does not need to specify some of NuSMV-specific constraint symbols (i.e., AG and AX). However, such constraint symbols are added by ACPTS when a property is converted to the NuSMV format shown in Fig. 9.

```
MODULE main

VAR
        decision : {NA, Permit, Deny};
        role_subject      : {None, Faculty , Student};
        user_subject      : {None, Jim , Jane};
        action    : {write , view};
        resource          : {grades , records};
ASSIGN
        init (decision) := NA ;
        next (decision) := case
                role_subject = Faculty & resource = grades & action = write :Permit ;
                role_subject = Student & resource = grades & action = write :Deny ;
                1                              : decision;
                esac;
        next (role_subject) := role_subject;
        next (user_subject) := user_subject;
        next (action) := action;
        next (resource) := resource;

SPEC AG ( (decision = NA) & (role_subject = Faculty) & (user_subject = None) &
(resource = grades) & (action = write) -> AF decision = Permit)
```

MAC State Descriptions

MAC State Transitions

MAC Properties

Fig. 9. A NuSMV input describing an example RBAC model and its property.

For model and property verification, NuSMV takes the description of finite state systems of the MAC model and specified properties as input; it then verifies finite state systems against their properties. NuSMV produces verification reports on whether the given properties are satisfied; when a property is violated, a counterexample will be generated accordingly. Figure 9 shows a NuSMV input describing the example RBAC model.

7.2. *Covering array generation*

For covering array generation, ACTS takes the description of variables as input; it then generates *t-way* covering arrays for given variables. The ACPTS generates 2-way and 3-way covering array for combinatorial tests, and compare their size and rule coverage.

Figure 10 shows the generated 2-way and 3-way covering arrays for the given subjects (e.g., Faculty, and Student), resources (e.g., grades and records), and actions (e.g., write and view), and, 4 and 8 rows are generated, respectively. As an MAC policy model is often composed of three attributes (subject, action, object), a 3-way covering array can be considered as exhaustively includes all possible combinations of values in each attribute. We can reduce the number of rows in a covering array

	SUBJECTS	RESOURCES	ACTIONS
1	Faculty	grades	write
2	Faculty	records	view
3	Student	grades	view
4	Student	records	write

	SUBJECTS	RESOURCES	ACTIONS
1	Faculty	grades	write
2	Faculty	grades	view
3	Faculty	records	write
4	Faculty	records	view
5	Student	grades	write
6	Student	grades	view
7	Student	records	write
8	Student	records	view

Fig. 10. 2-way (left) and 3-way (right) covering array of given subjects, resources, and actions by ACTS.

by considering 2-way combinations of these attributes for detecting a fault related to 2-way interactions.

7.3. *Mutant rules*

We perform mutant rule verification to detect insufficient rule coverage by a specified property set. When ACPTS detects any missing rule coverage, a policy author can augment the existing properties with new properties to achieve high rule coverage.

In order to check whether a given property set in Fig. 9 is satisfied, we mutate the first and second rules one at a time to produce two mutant rules as shown in Fig. 11 where *r1 and r2* represent mutant rules of the first and second rules, respectively by negating their decisions (Definition 3) in Fig. 9. As a verification result, the property set is not satisfied and a counterexample is reported as follows.

-> State: 1.1 <-
decision = NA
role_subject = **Faculty**
action = **write**
resource = **grades**
. . .
-> State: 1.2 <-
decision = **Deny**

This counterexample indicates that the property set can cover at least one of the two mutated rules. The counterexample illustrates that the property (Faculty is permitted to write grades) is violated because a request that a Faculty is denied to write grades.

r1: role_subject = Faculty & resource = grades & action = write : Deny;
r2: role_subject = Student & resource = grades & action = write : Permit;

Fig. 11. Mutant rules.

To determine which rule is not covered by the property set, we mutate a rule (one at a time) in the original policy. When only the first rule is mutated, the counterexample is generated in the process of verification. This counterexample indicates that the first rule is covered by the property set. However, when the second rule is mutated, no counterexample is generated. This verification result indicates that the second rule is not covered by the property set. Therefore, the existing property set achieves insufficient rule coverage not covering the second rule coverage. We manually generate and augment the following property derived from the second rule.

SPEC AG ((role_subject = Student) & (resource = grades) & (action = write) -> AF decision = Deny)

With the addition of this property, the new property set is sufficient in achieving full rule coverage and NuSMV reports counterexamples in the verification of all the mutants.

7.4. *Mutant property*

We conduct property confinement checking to detect security problems caused by allowing exceptional permission. We generate and add a property's mutant property to the NuSMV model for the next run of model checking. Figure 12 shows a mutant property derived from the property set described in Fig. 9.

The model in Fig. 9 is verified against the mutated property, and a counterexample is reported as follows.

-> State: 1.1 <-
decision = **NA**
role_subject = **Student**
action = **view**
resource = **records**

This counterexample illustrates that the mutated property is violated because NuSMV found that non-applicable decision (denoted as "NA") is returned for a request that a Student view records. This checking detects the discrepancy of the specified properties by the counterexample, which is derived from *otherwise : decision* (which is specified as "1 : decision; " in Fig. 9). Therefore, we change *otherwise : decision* to *otherwise : Deny* (which is specified as "1 : Deny; " in Fig. 9) to remove such discrepancy. Our confinement checking technique helps detect such discrepancy

SPEC AG (! (role_subject = Facutly) & (resource = grades) & (action = write) -> AF decision = Deny)

Fig. 12. Mutant property.

and the policy author can increase their confidence for policy correctness by fixing the discrepancy or confirming the discrepancy to be intended.

8. Related Work

There exist several verification techniques for applying model checking on MAC *policies* but few *general* verification techniques for applying model checking on MAC *models* and generating test cases as our proposed approach. Zhang *et al.* [27] present a model-checking algorithm that evaluates if a MAC policy can satisfy a user's access request as well as prevent intruders from reaching their malicious goals. Instead of generic model language, policies of the MAC system and goals of agents must be described in the AC description and specification language introduced as RW in their earlier work. The language does not provide the flexibility for the specification of dynamic or historical types of MAC model nor for the descriptions of the general properties of access constraints. Kikuchi *et al.* [19] proposed the policy verification and validation framework based on model checking that exhaustively verifies a policy's validity by considering the relations between system characteristics and policies. Their approach defines the validity of policies and the information needed to verify them from the viewpoint of model checking as well as constructs the policy verification framework based on the definition. Besides rule-based system policies, there is no demonstration that shows the proposed framework is proper for generic MAC policies. Schaad *et al.* [28] presented a model-checking approach to analyze the delegation and revocation functionalities of workflow-based enterprise resource management (ERP) systems. Their approach is done in the context of a real-world banking workflow requiring static and dynamic separation of duty properties. The approach derived information about the workflow from Business Process Execution Language (BPEL) specifications and ERP business object repositories. This was captured in an SMV specification together with a definition of possible delegation and revocation scenarios. Their focus was on how to capture the workflow in an SMV model amended by an LTL-based specification of the Separation of Duty properties without much consideration of generic MAC models.

Commercial policy manager tools such as IBM security policy manager [36] and Cisco policy manager [37] do not generate policy models for property verification, property assessment, and test suite generation. The tools include PDP (Policy Decision Point) and security protocol support. Some of the tools have limited verification feature. For example, IBM security policy manager includes limited SOD (Separation of Duty) check on given policies.

Table 1 summarizes model specification, property verification, property assessment, test suite generation feature information for each of related approaches. Each row of the table corresponds to a related model checking or policy manager approach and each of columns corresponds to its features. More specifically, the second column in the table includes description of demonstrated policy models in the corresponding approach.

Table 1. Comparison of features on related model chcking and policy manager approaches.

Product	Model specification	Property verification	Property assessment	Test suite generation
ACPTS	Static, dynamic, and historic policy model	Yes	Yes	Yes
Model Checking tool (Zhang *et al.*)	Static policy model	Yes	No	No
Model Checking tool (Kikuchi *et al.*)	Static policy model	Yes	No	No
Model Checking tool (Schaad *et al.*)	Historic policy model	Yes	No	No
IBM Security Policy Manager V7.0	No	Yes (Separation of Duty)	No	No
Cisco Policy Manager	No	No	No	No

Different from these existing approaches, our proposed approach is targeted at MAC models and their generic properties, and is more general and applicable in a larger scope of models and properties. In addition to property verification, our approach provides efficient test generation, which generates test cases that guarantee combinatorial coverage for the input parameters used in the policy, thus a thorough verification of MAC implementation.

9. Conclusion

To verify properties for MAC models, we propose a new general approach that expresses MAC models in the specification language of a model checker and generic AC properties in its property language as temporal logic formula. Then the approach exploits the verification process of the model checker to verify the specified models against the specified properties. Our approach is able to support the verification of three common types of generic AC properties: static, dynamic, and historical constraints. In addition, the approach also supports automated generation of test cases to check the conformance of the models and their implementations.

In future work, we plan to develop a tool for assisting the users in specifying MAC models and properties in a more user friendly way. We also plan to investigate and expand the scope of models and properties supported by our approach. Through our research, we will gain understanding about testing and verifying MAC policies in policy development, which should lead to better policy quality and higher security assurance in general. Our research results related to fundamentally advancing knowledge and understanding will be disseminated in software engineering and security conferences, journals, and books in various forms (e.g., papers, tutorials, and book chapters). The groundwork for the proposed work has been widely published [29–35], and we will continue to widely disseminate the results produced by the proposed work.

The work of conformance verification of generic MAC properties brings benefits to society in two aspects. First, it should lead the practices for testing and verifying MAC policies in improving policy quality and security in general. Second, innovations in new testing and verification algorithms and tools tend to propagate quickly across application or task domains where MAC policies are used.

References

1. C. P. Pfleeger, *Security in Computing*, 2nd edn. (Prentice Hall PTR, 1997).
2. D. Ferraiolo, D. Kuhn and R. Chandramouli, Role-Based Access Control, Artech House, Computer Security Series, 2003.
3. V. Hu, D. Frincke and D. Ferraiolo, The policy machine for security policy management, in *Proc. ICCS Conference*, San Francisco, May 2001.
4. P. Bonatti, S. Vimercati and P. Samarati, A modular approach to composing access control policies, in *Proc. ACM Conference on Computer and Communication Security*, Athens, Greece, November 2000.
5. S. Jajodia, P. Samarati and V. S. Subrahmanian, A logical language for expressing authorizations, in *Proc. 1997 IEEE Symposium on Security and Privacy* (1997), pp. 31–42.
6. M. Kudo and S. Hada, XML document security based on provisional authorization, in *Proc. ACM Conference on Computer and Communication Security*, Athens, Greece, November 2000.
7. E. C. Lupu and M. Sloman, Conflict in policy-based distributed systems management, *IEEE Trans Software Engineering* **25**(6) (1999) 852–869.
8. E. Martin, T. Xie and V. C. Hu, Assessing quality of policy properties in verification of access control policies, North Carolina State University Department of Computer Science Technical report TR-2007-25, September 16, 2007.
9. D. R. Kuhn, R. Kacker and Y. Lei, 22 CROSSTALK, *The Journal of Defense Software Engineering*, June 2008.
10. V. C. Hu, R. D. Kuhn and T. Xie, Property Verification for Generic Access Control Models, in *Proc. 2008 IEEE/IFIP International Symposium on Trust, Security and Privacy for Pervasive Application (TSP2008)*, Shanghai, China, December 17–20, 2008.
11. D. E. Bell and L. J. LaPadula, *Secure Computer Systems: Mathematical Foundations* (MITRE Corporation, 1973).
12. D. Ferraiolo and R. Kuhn, Role based access control, in *Proc. 15th NIST-NCSC National Computer Security Conference*, 1992, pp. 554–563.
13. National Computer Security Center, Integrity in Automated information System, Technical Report 79-91, Library No. S237,254, September 1991.
14. D. F. C. Brewer and M. J. Nash, The Chinese wall security policy, in *Proc. IEEE Symposium on Security and Privacy* (1989), pp. 206–214.
15. Workflow Management Coalition, Workflow Management Coalition Terminology & Glossary, documentation number WFMC-TC-1011, February 1999. http://www.wfmc.org/.
16. M. Ben-Ari, Z. Manna and A. Pnueli, The temporal logic of branching time, *Acta Informatica* **20** (1983).
17. E. M. Clarke, E. A. Emerson and A. P. Sistla, Automatic verification of finite-state concurrent systems using temporal-logic specifications, *ACM Trans. Programming Languages and Systems* **8**(2) (1986).

18. A. Pnueli, A temporal logic for concurrent programs, *Theoretical Computer Science* **13** (1980).
19. S. Kikuchi, S. Tsuchiya, M. Adachi and T. Katsuyama, Policy verification and validation framework based on model checking approach, in *Proc. International Conference on Autonomic Computing* (2007), pp. 1–9.
20. T. Jaeger and E. T. Jonathon, Practical safety in flexible access control model, *ACM Transitions on Information and System Security* **4**(2) (2001) 158–190.
21. E. Martin and T. Xie, A fault model and mutation testing of access control policies, in *Proc. 16th International Conference on World Wide Web*, Banff, Alberta, Canada (2007), pp. 667–676.
22. D. R. Kuhn and V. Okun, Pseudo-exhaustive testing for software, in *Proc. 30th NASA/IEEE Software Engineering Workshop* (2006) April 25–27.
23. Y. Lei *et al.*, Efficient test generation for multi-way combinatorial testing, *Software Testing, Verification, and Reliability*, Wiley InterScience, October 2007.
24. http://csrc/nist/gov/acts.
25. NuSMV: NuSMV 2.2 Tutorial, a new symbolic model checker, http://nusmv.irst.itc.it/.
26. R. Sandhu, V. Bhamidipati and Q. Munawer, The ARBAC97 model for role-based administration of roles, *ACM Transactions on Information and Systems Security* **2**(1) (1999) 105–135.
27. N. Zhang, M. D. Ryan and D. Guelev, Evaluating access control policies through model checking, in *Proc. Information Security Conference* (2005), pp. 446–460.
28. A. Schaad, V. Lotz and K. Sohr, A model-checking approach to analysing organisational controls in a loan origination process, in *Proc. ACM Symposium on Access Control Models and Technologies* (2006), pp. 139–149.
29. V. C. Hu, E. Martin, J. Hwang and T. Xie, Conformance checking of access control policies specified in XACML, in *Proc. 1st IEEE International Workshop on Security in Software Engineering (IWSSE 2007)*, Beijing, China (July 2007), pp. 275–280.
30. E. Martin and T. Xie, A fault model and mutation testing of access control policies, in *Proc. 11th International Conference on World Wide Web (WWW 2007), Security, Privacy, Reliability, and Ethics Track*, Banff, Alberta, Canada (May 2007), pp. 667–676.
31. E. Martin and T. Xie, Automated test generation for access control policies via change-impact analysis, in *Proc. 3rd International Workshop on Software Engineering for Secure Systems (SESS 2007)*, Minneapolis, MN (May 2007), pp. 5–11.
32. E. Martin, T. Xie and T. Yu, Defining and measuring policy coverage in testing access control policies, in *Proc. 8th International Conference on Information and Communications Security (ICICS 2006)*, Raleigh, NC (December 2006) pp. 139–158.
33. E. Martin and T. Xie, Automated test generation for access control policies, in *Supplemental Proc. 17th IEEE International Conference on Software Reliability Engineering (ISSRE 2006)*, *Fast Abstracts*, Raleigh, NC (November 2006).
34. E. Martin and T. Xie, Inferring access-control policy properties via machine learning, in *Proc. 7th IEEE Workshop on Policies for Distributed Systems and Networks (POLICY 2006)*, London, Ontario Canada (June 2006), pp. 235–238.
35. E. Martin, T. Xie and V. C. Hu, Assessing quality of policy properties in verification of access control policies, North Carolina State University Department of Computer Science Technical report TR-2007-25 (September 16), 2007.
36. IBM Policy Manager V7.0: http://www.redbooks.ibm.com/redpapers/pdfs/redp4512.pdf.
37. Cisco Policy Manager: http://www.cisco.com/en/US/products/ps9530/index.html.