Property Verification for Generic Access Control Models[1]

Vincent C. Hu[1], D. Richard Kuhn[1], Tao Xie[2]
[1]National Institute of Standards and Technology, [2]North Carolina State University
vhu@nist.gov, kuhn@nist.gov, xie@csc.ncsu.edu

Abstract

To formally and precisely capture the security properties that access control should adhere to, access control models are usually written to bridge the rather wide gap in abstraction between policies and mechanisms. In this paper, we propose a new general approach for property verification for access control models. The approach defines a standardized structure for access control models, providing for both property verification and automated generation of test cases. The approach expresses access control 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 these properties for the access control models and generates test cases via combinatorial covering array for the system implementations of the models.

1. Introduction

Access control systems are among the most critical of network security components. It is common that a system's privacy and security are compromised due to the misconfiguration of access control policies rather than the failure of cryptographic primitives or protocols. This problem becomes increasingly severe as software 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 specifications and their intended function is crucial because correct implementation and enforcement of policies by applications is based on the premise that the policy specifications are correct. As a result, policy specifications must undergo rigorous verification and validation to ensure that the policy specifications truly encapsulate the desires of the policy authors.

To formally and precisely capture the security properties that access control should adhere to, access control models are usually written, to bridge the rather wide gap in abstraction between policies and mechanisms. Users see access control models as unambiguous and precise expression of requirements; vendors and system developers see access control models as design and implementation requirements.

In this paper, we propose a new general approach to property verification for access control models by combining model checking and combinatorial testing. Section 2 presents our approach of model checking and combinatorial covering array, section 3 demonstrates an example for the approach, section 4 compares our work with related work, and section 5 is the conclusion.

2. Approach

Our approach expresses access control models in the specification language of a model checker (Section 2.1), expresses generic access control properties in temporal logic formulae, and verifies these properties with the model checker (Section 2.2). *Test cases*, consisting of *input data* and *expected results*, are generated from a combinatorial covering array following verification (Section 2.3) as illustrated in Figure 1. One goal of the techniques in our approach is to reduce overall software assurance costs by integrating verification with test generation.

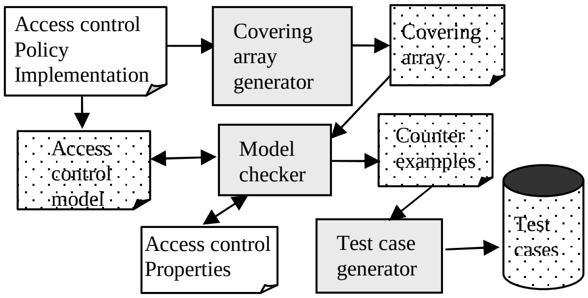

Figure 1. Property verification

[1] This work is supported in part by NSF grant CNS-0716579.

2.1. Model Specification

As the test cases for the verification are translated from the counterexamples generated by the model checker's property checking to the entries of covering array (Figure 1), we specify access control in a Finite State Machine (FSM) that describes the transition of authorization states. In general, any expression in the propositional calculus can be used to define the transition relation of authorization states; however, the flexibility of the expression is accompanied by the risk of a logical contradiction, which makes specifications vacuously true or makes the system unimplementable. Fundamentally, there are three basic types of FSM expressions **Synchronous, Asynchronous,** and **Direct specification** for specifying access control models in terms of the sequence of the state transitions.

From the separation of duty and safety point of view, the state transitions of access control properties can be categorized by three types of constraints, namely **static**, **dynamic**, and **historical** constraints. These constraints can be expressed directly by the three basic types of FSM models.

2.2. Property Specification and Verification

In this section, we illustrate how the properties of constraints (with their models) can be specified and verified by our approach. We use SMV[1]-like pseudo code for the specification of access control models, and Computational Tree Logic (CTL) for the specification of the access control properties for the reason that the constraint models/properties can be successfully specified, and the combinatorial testing functions can be easily integrated with the checking results generated by the SMV.

2.2.1. Static Constraints. Static constraints regulate the access permission by static system states or conditions such as rules, attributes, and system environments (times and locations for access). Popular access control models with these types of properties include Role-Based Access Control (RBAC) [2] and Multi-Level Access Control [3]. 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_i*….& *constraint_n*), 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 & …. constraint_n) → AX (access_state = 1)

AG (constraint_a & constraint_b & …. constraint_m) → AX (access_state = 1) ……

AG ! ((constraint_1 & ….constraint_n) | (constraint_a & …. constraint_m) |…) → 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-complied ones should be denied.

2.2.2. Dynamic Constraints. Dynamic constraints regulate the access permission by dynamic system states or conditions such as specified events or system counters. An access control 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 request. The specification of the dynamic constraints is verified by verifying the following properties expressed in temporal logic formula:

```
AG  (state == entering) & (act == rd) & (object == obj) → AX (access = grant)
AG (state == idle | state == critical | state == exiting) → AX (access = deny)
```

where temporal logic formula *AG (p → AX q)* indicates that “if condition *p* is true at time *t*, condition *q* is true at all times later than *t*.

2.2.3. Historical Constraints. Historical constraints regulate the access permission by historical access states or recorded and predefined series of events. The representative access control policies for this type of access control models are N-person [4], Chinese Wall [5], and Workflow [6] access control policies. This type of models can be best described by **synchronous** or **direct specification** expressions of an FSM. For example, the following **synchronous** FSM specification specifies a Chinese Wall access control model where there are two Conflict of Interest groups of objects:

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

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

```
AG ((state == 2 & act == rd & object == COI1) | (state == 3 & act == rd & object == COI2)) → AX (access = grant)
AG ! ((state == 2 & act == rd & object == COI1) | (state == 3 & act == rd & object == COI2)) → AX (access = deny)
```

2.3. Test Generation

In addition to supporting property verification, the model checking technique was adopted because it fits well with a variety of test generation techniques, such as fault-based mutation testing [7] and combinatorial testing [8]. Mutation testing allows us to test for the presence of hypothesized faults, or faults that they subsume, and combinatorial testing makes it possible to rule out complex interactions that may lead to failures. 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. 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), in addition to formal verification of properties. A case study of this technique for software is given in [9].

Even with highly automated tools such as model checkers, 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 errors 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.

To produce test cases that guarantee combinatorial coverage to an interaction level *t*, we produce a *t*-way covering array [10] 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* 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 ($p_1 = v_{11}$ & ... & $p_n = v_{n1}$) → *AX !(access_state = grant) ……*
AG ($p_1 = v_{12}$ & ... & $p_n = v_{n2}$) → *AX !(access_state = grant) ……*
AG ($p_1 = v_{11}$ & ... & $p_n = v_{n1}$) → *AX !(access_state = deny)……*

For a covering array with *n* rows, a total of 2*n* 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 is 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.

3. Demonstration

This section provides a step-by-step demonstration for the approach described in section 2. The result is a set of complete test cases, i.e., input values with the expected output for each set of inputs. We used a non-commercial research tool called Fireeye [11] developed by NIST and the University of Texas at Arlington (available on http://csrc/nist/gov/acts) as covering array generator and NuSMV [1] as model checker for the demonstration. From Fireeye, 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 SMV specifications that can be processed by the NuSMV model checker.

3.1. Sample access control policy

The rules of the demonstrated access control policy are a simplified multi-level security system [12], in which each subject (user) has a clearance level *u_l*, and each file has a classification level *f_l*. Levels are given as 0, 1, or 2, which could represent levels such as Confidential, Secret, and Top Secret. A user *u* can read a file f if *u_l* ≥ *f_l* (the "no read up" rule), or write to a file if *f_l* ≥ *u_l* (the "no write down" rule). Thus a pseudo code representation of the access control rules is:

```
if u_l >= f_l & act = rd then GRANT;
 else if f_l >= u_l & act = wr then GRANT;
 else DENY;
```

The **Static Constraints** property of this policy is easily modeled in SMV as a simple two-state finite state machine by **Direct Specification**. The START_ state merely initializes the system (line 8, Figure 2), with the rule above used to evaluate access as either GRANT or DENY (lines 9-13). For example, line 9 represents the first line of the pseudo code above: in the current state, if *u_l* ≥ *f_l* then the next state is GRANT. Each line of the case statement is examined sequentially, as in a conventional programming language. Line 12 implements the "else DENY" rule, since the predicate "1" is always true. SPEC clauses given at the end of the model are simple "reflections" that duplicate the access control rules as temporal logic statements.

```
MODULE main
VAR
--Input parameters
u_l:   0..2;                 -- user level
f_l:   0..2;                 -- file level
act:  {rd,wr};               -- action
--output parameter
access: {START_, GRANT,DENY};
ASSIGN
init(access) := START_;
--if access is allowed under rules, then next state is GRANT
--else next state is DENY
next(access) := case
u_l >= f_l & act = rd : GRANT;
f_l >= u_l & act = wr : GRANT;
1 : DENY;
esac;
next(u_l) := u_l;
next(f_l) := f_l;
next(act) := act;

-- reflection of the assigns for access
-- if user level is at or above file level then read is OK
SPEC AG ((u_l >= f_l & act = rd ) -> AX (access =
        GRANT));
-- if user level is at or below file level, then write is OK
SPEC AG ((f_l >= u_l & act = wr ) -> AX (access =
        GRANT));
-- if neither condition above is true, then DENY any
        action
SPEC AG (!( (u_l >= f_l & act = rd ) | (f_l >= u_l & act =
   wr ))  -> AX (access = DENY));
```

Figure 2. SMV model of access control rules

Specifications of the form "`AG ((`*predicate 1*`) -> AX (`*predicate 2*))" indicate essentially that for all paths (the "A" in "AG") for all states globally (the "G"), if *predicate 1* holds then ("`->`") for all paths, in the next state (the "X" in "AX") *predicate 2* will hold. If the statement is false, the model checker not only reports this, but also provides a "counterexample". We simply use the model checker to determine whether a particular input data set makes a SPEC claim true or false. That is, we will enter claims that particular results can be reached for a given set of input data values, and the model checker will tell us if the claim is true or false. This gives us the ability to match every set of input test data with the result that the system should produce for that input data.

3.2. NuSMV output

As shown in Figure 3, checking the properties in the `SPEC` statements shows that they match the access control rules as implemented in the FSM, as expected. In other words, the claims we made about the state machine in the `SPEC` clauses can be proven. This step is used to check that the `SPEC` claims are valid for the model defined previously. If NuSMV is unable to prove one of the `SPEC`s, then either the spec or the model is incorrect. This problem must be resolved before continuing with the test generation process. Once the model is correct and `SPEC` claims have been shown valid for the model, counterexamples can be produced that will be turned into test cases.

```
*** This is NuSMV 2.4.3 (compiled on Tue
      May 22 14:08:54 UTC 2007)
*** For more information on NuSMV see
      <http://nusmv.irst.itc.it>
*** or email to <nusmv-
      users@irst.itc.it>.
*** Please report bugs to
      <nusmv@irst.itc.it>.
*** This version of NuSMV is linked to
      the MiniSat SAT solver.
***See
      http://www.cs.chalmers.se/Cs/Resea
      rch/FormalMethods/MiniSat
*** Copyright (c) 2003-2005, Niklas Een,
      Niklas Sorensson

-- specification AG ((u_l >= f_l & act =
      rd) -> AX access = GRANT)  is true
-- specification AG ((f_l >= u_l & act =
      wr) -> AX access = GRANT)  is true
-- specification AG (!((u_l >= f_l & act
      = rd) | (f_l >= u_l & act = wr)) -
      > AX access = DENY)  is true
```

Figure 3. NuSMV output

3.3. Combinatorial testing of policy

Combinatorial testing is a methodology that tests all *t*-way combinations of input parameter values. 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 access control policies is that access control often relies on a small number of discrete values for most parameters. For example, a multi-level security 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 access control 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 remainder of this section gives a worked example of automated combinatorial test construction.

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 Figure 4, 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 1,024 for exhaustive coverage. Similar arrays can be generated to cover up to all 6-way combinations. Fireeye makes this possible with much greater efficiency than previous tools. For example, a commercial tool required 5,400 seconds to produce a less optimal test set than Fireeye generated in 4.2 seconds.

0	0	0	0	0	0	0	0	0	0
1	1	1	1	1	1	1	1	1	1
1	1	1	0	1	0	0	0	0	1
1	0	1	1	0	1	0	1	0	0
1	0	0	0	1	1	1	0	0	0
0	1	1	0	0	1	0	0	1	0
0	0	1	0	1	0	1	1	1	0
1	1	0	1	0	0	1	0	1	0
0	0	0	1	1	1	0	0	1	1
0	0	1	1	0	0	1	0	0	1
0	1	0	1	1	0	0	1	0	0
1	0	0	0	0	0	0	1	1	1
0	1	0	0	0	1	1	1	0	1

Figure 4. 3-way covering array for 10 parameters with 2 values each.

We compute covering arrays that give all *t*-way combinations, with degree of interaction coverage = 2 (2-way, or pairwise) for this example. The first step is to define the parameters and their values in a system definition file that will be used as input to Fireeye. Call this file "in.txt", with the following format:

```
[System]
[Parameter]
u_l: 0,1,2
f_l: 0,1,2
act: rd,wr
[Relation]
[Constraint]
[Misc]
```

Fireeye produces the output shown in Figure 5.

```
Number of parameters: 3
Maximum    number    of    values    per
parameter: 3
Number of configurations: 9
-------------------------------------
Configuration #1:
1 = u_l=0
2 = f_l=0
3 = act=rd
-------------------------------------
Configuration #2:
1 = u_l=0
2 = f_l=1
3 = act=wr
-------------------------------------
Configuration #3:
1 = u_l=0
2 = f_l=2
3 = act=rd
-------------------------------------
Configuration #4:
1 = u_l=1
2 = f_l=0
3 = act=wr
-------------------------------------
Configuration #5:
1 = u_l=1
2 = f_l=1
3 = act=rd
-------------------------------------
Configuration #6:
1 = u_l=1
2 = f_l=2
3 = act=wr
-------------------------------------
Configuration #7:
1 = u_l=2
2 = f_l=0
3 = act=rd
-------------------------------------
Configuration #8:
1 = u_l=2
2 = f_l=1
3 = act=wr
-------------------------------------
Configuration #9:
1 = u_l=2
2 = f_l=2
3 = act=wr
```

Figure 5. Fireeye output

Each test configuration defines a set of values for the input parameters *u_l*, *f_l*, and *act*. The complete test set ensures that all 2-way combinations of parameter values have been covered. If we had a larger number of parameters, we could produce test configurations that cover all 3-way, 4-way, etc. combinations. (With only three parameters, 3-way interaction would be equivalent to exhaustive testing.) Fireeye may output "don't care" for some parameter values. This means that any legitimate value for that parameter can be used and the full set of configurations will still cover all t-way combinations. Since "don't care" is not normally an acceptable input for programs being tested, a random value for that parameter is substituted before using the covering array to produce tests.

3.4. SPEC claims with combinatorial test values inserted

The next step is to assign values from the covering array to parameters used in the model. For each test, we claim that the expected result will not occur. The model checker determines combinations that would disprove these claims, outputting these as counterexamples. Each counterexample can then be converted to a test with known expected result. As can be seen below, for each of the 9 configurations in the covering array of Figure 5, we create a SPEC claim of the form:

```
SPEC AG(( <covering array values> ) -> AX !(access =
<result>));
```

This process is repeated for each possible result, in this case either "GRANT" or "DENY", so we have 9 claims for each of the two results as in Figure 6.

Excerpt:

```
. . .
-- reflection of the assign for access
--SPEC AG ((u_l >= f_l & act = rd ) -> AX (access = GRANT));
--SPEC AG ((f_l >= u_l & act = wr ) -> AX (access = GRANT));
--SPEC AG (!( (u_l >= f_l & act = rd ) | (f_l >= u_l & act = wr ) )
        -> AX (access = DENY));

* SPEC AG((u_l = 0 & f_l = 0 & act = rd) -> AX !(access = GRANT));
* SPEC AG((u_l = 0 & f_l = 1 & act = wr) -> AX !(access = GRANT));
  SPEC AG((u_l = 0 & f_l = 2 & act = rd) -> AX !(access = GRANT));
  SPEC AG((u_l = 1 & f_l = 0 & act = wr) -> AX !(access = GRANT));
* SPEC AG((u_l = 1 & f_l = 1 & act = rd) -> AX !(access = GRANT));
* SPEC AG((u_l = 1 & f_l = 2 & act = wr) -> AX !(access = GRANT));
* SPEC AG((u_l = 2 & f_l = 0 & act = rd) -> AX !(access = GRANT));
  SPEC AG((u_l = 2 & f_l = 1 & act = wr) -> AX !(access = GRANT));
* SPEC AG((u_l = 2 & f_l = 2 & act = rd) -> AX !(access = GRANT));
  SPEC AG((u_l = 0 & f_l = 0 & act = rd) -> AX !(access = DENY));
  SPEC AG((u_l = 0 & f_l = 1 & act = wr) -> AX !(access = DENY));
* SPEC AG((u_l = 0 & f_l = 2 & act = rd) -> AX !(access = DENY));
* SPEC AG((u_l = 1 & f_l = 0 & act = wr) -> AX !(access = DENY));
  SPEC AG((u_l = 1 & f_l = 1 & act = rd) -> AX !(access = DENY));
  SPEC AG((u_l = 1 & f_l = 2 & act = wr) -> AX !(access = DENY));
  SPEC AG((u_l = 2 & f_l = 0 & act = rd) -> AX !(access = DENY));
* SPEC AG((u_l = 2 & f_l = 1 & act = wr) -> AX !(access = DENY));
  SPEC AG((u_l = 2 & f_l = 2 & act = rd) -> AX !(access = DENY));
```

Figure 6. SPEC Claims

3.5. Counterexamples

NuSMV produces counterexamples where the input values would disprove the claims specified in Figure 2. Each of these counterexamples is thus a set of test data that would have the expected result of GRANT or DENY.

For each SPEC claim, if this set of values cannot in fact lead to the particular result R_j, the model checker indicates that this is true. For example, for the configuration below, the claim that access will not be granted is true, because the user's clearance level (u_l = 0) is below the file's level (f_l = 2):

```
-- specification AG (((u_l = 0 & f_l = 2) & act = rd) -
> AX !(access = GRANT)) is true
```

If the claim is false, the model checker indicates this and provides a trace of parameter input values and states that will prove it is false, as shown below:

Excerpt from NuSMV output:

```
-- specification AG (((u_l = 0 & f_l = 0) & act = rd) -
> AX !(access = GRANT)) is false
-- as demonstrated by the following execution
sequence
Trace Description: CTL Counterexample
Trace Type: Counterexample
-> State: 1.1 <-
  u_l = 0
  f_l = 0
  act = rd
  access = START_
-> Input: 1.2 <-
-> State: 1.2 <-
  access = GRANT
```

It is then simple to map these values into complete test cases in the syntax needed for the system under test. The model checker finds that 6 of the input parameter configurations produce a result of GRANT and 3 produce a DENY result as SPEC Claims marked with a * in Figure 5, so at the completion of this step we have successfully matched each input parameter configuration with the result that should be produced by the SUT.

3.6. Test cases

We now strip out the parameter names and values, giving tests that can be applied to the system under test. The tests produced are shown below:

```
u_l = 0 & f_l = 0 & act = rd -> access = GRANT
u_l = 0 & f_l = 1 & act = wr -> access = GRANT
u_l = 1 & f_l = 1 & act = rd -> access = GRANT
u_l = 1 & f_l = 2 & act = wr -> access = GRANT
u_l = 2 & f_l = 0 & act = rd -> access = GRANT
u_l = 2 & f_l = 2 & act = rd -> access = GRANT
u_l = 0 & f_l = 2 & act = rd -> access = DENY
u_l = 1 & f_l = 0 & act = wr -> access = DENY
u_l = 2 & f_l = 1 & act = wr -> access = DENY
```

These test definitions can now be post-processed using simple scripts to produce a test harness that will execute the SUT with each input and check the results.

4. Related Work

There exist several verification techniques for applying model checking on access control *policies* but few *general* verification techniques for applying model checking on access control *models* and generating test cases as our proposed approach. Zhang et al. [13]

present a model-checking algorithm that evaluates if an access control 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 access control system and goals of agents must be described in the access control 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 access control model nor for the descriptions of the general properties of access constraints. Kikuchi et al. [14] 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 access control policies. Schaad et al. [15] 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 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 access control models.

Different from these existing approaches, our proposed approach is targeted at access control 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 access control implementation.

5. Conclusion

To verify properties for access control models, we propose a new general approach that expresses access control models in the specification language of a model checker and generic access control 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 access control 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.

6. References

[1] NuSMV: http://nusmv.irst.itc.it/

[2] D. Ferraiolo and R. Kuhn. Role based access control. In *Proc. 15th NIST-NCSC National Computer Security Conference*, pp. 554–563, 1992.

[3] D. E. Bell and L. J. LaPadula. Secure computer systems: Mathematical foundations, 1973. MITRE Corporation.

[4] National Computer Security Center. Integrity in Automated information System. Technical Report 79-91, Library No. S237,254, Sept. 1991.

[5] D. F. C. Brewer and M. J. Nash. The Chinese wall security policy. In *Proc. IEEE Symposium on Security and Privacy*, pp. 206–214, 1989.

[6] Workflow Management Coalition. Workflow Management Coalition Terminology & Glossary. http://www.wfmc.org/ Documentation number WFMC-TC-1011, February 1999.

[7] P. Ammann and P.E. Black. Abstracting Formal Specifications to Generate Software Tests via Model Checking. In *Proc. Digital Avionics Systems Conference*, pp. 10.A.6-1 - 10.A.6-10, 1999.

[8] D.R. Kuhn, D.R. Wallace, and A.J. Gallo, Jr. Software Fault Interactions and Implications for Software Testing. *IEEE Trans. on Software Engineering*, Vol. 30, , June 2004.

[9] D. R. Kuhn and V. Okun. Pseudo-exhaustive Testing For Software, In *Proc. 30th NASA/IEEE Software Engineering Workshop*, April 25-27, 2006.

[10] Y. Lei, R. et al. Efficient Test Generation for Multi-Way Combinatorial Testing, *Software Testing, Verification, and Reliability*. Wiley InterScience, , October 2007.

[11] http://csrc.nist.gov/groups/SNS/acts/index.html

[12] Pfleeger C. P. Security In Computing Second Edition, by *Prentice-Hall PTR*, 1997.

[13] N. Zhang, M. D. Ryan, and D. Guelev. Evaluating Access Control Policies Through Model Checking. In *Proc. Information Security Conference*, pp. 446-460, 2005.

[14] 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*, pp. 1-9, 2007.

[15] 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*, pp. 139-149, 2006.