Notice of Violation of IEEE Publication Principles

“A Scenario-Based Test Case Generation Framework for Security Policies”
by Chen Yan and Wu Dan
International Conference on Computational Intelligence and Security, 2009. CIS '09
December 2009, pp. 489-493

After careful and considered review of the content and authorship of this paper by a duly constituted expert committee, this paper has been found to be in violation of IEEE's Publication Principles.

This paper contains significant portions of original text from the paper cited below. The original text was copied without attribution (including appropriate references to the original author(s) and/or paper title) and without permission.

Due to the nature of this violation, reasonable effort should be made to remove all past references to this paper, and future references should be made to the following article:

"Model-based Testing Using Scenarios and Event-B Refinements"
by Qaisar A. Malik, Johan Lilius and Linas Laibinis
Methods, Models and Tools for Fault-Tolerance, Lecture Notes in Computer Science series, Springer, March 2009, pp. 177-195

A Scenario-based Test Case Generation Framework for Security Policies

Chen Yan
Scholl of Mechatronical Engineering
Beijing Institute of Technology
Beijing, China
chenyanbit@gmail.com

Wu Dan
Scholl of Mechatronical Engineering
Beijing Institute of Technology
Beijing, China
dwu.bit@gmail.com

***Abstract*—Security policy system is critical to the security sensitive implementation systems. To increase confidence in the correctness of the security policies, policy developers can conduct policy testing to ensure security policies are correctly implemented in these systems. In this paper we present a novel framework for security policy testing. The framework takes the security policy as a support for the secure operation of the functional systems and seizes the dynamic characteristic of the security policies which change with the refinement of functional systems. The testing framework is model based and includes three parts. They are security policy integration; refinement based testing scenario acquisition and risk-based testing organization. The framework covers all elements for the model-based testing and provides a comprehensive solution for the security policy testing.**

Keywords-security policy; testing; scenario

I. INTRODUCTION

Security policies provide a new and promising solution for network security management. But there are still many problems in the practical application of security policies, one of which is the validation of security policies. To solve this key problem researchers are trying to bring modern software engineering techniques into the validation of security polices. Many classical validation techniques have been applied to security policy validation [1-2]. Among these works testing remains indispensable. Ammar Masood et al. [3] constructed an FSM model of the role based access control policy and then generated test cases from the model using W-method. Paul Ammann et al. [4] presented an automatic test case generation method based on model checking. The model checker generates the test cases that satisfies a given coverage criteria and validate whether software faithfully implements a policy. However, in these works the relationship between the security policies and the functional system is separated. Although some woks integrate the security policies into the functional system [5-7], the model they built is static and lacks the concept of refinement. Problems also exist in the coverage criteria. With existing techniques [8-9], it is still difficult to address the issue of good criteria for systematic generation of high-quality policy test suites. Although the works above have solved the problem of test cases generation for security policies to some extent, there still lacks the theory support towards the organization of testing process. In our work, we address these problems with respect to scenarios.

First we build the security policy integrated model. Then from the usage view of the system the use scenarios conformed to the model are built. The test cases are automatically generated from these scenarios, and on the risk factors the test cases are organized in the process of testing.

The rest of the paper is organized as follows. In Section II, the overview of the scenario-based test case generation framework is presented. Implementation details of the framework are described in Section III. In Section IV, the approach is demonstrated on a simple example: the security policy integrated workflow system. Section V concludes the paper.

II. SCENARIO-BASED TEST CASE GENERATION FRAMEWORK OVERVIEW

In this section, we present our scenario-based test case generation framework, which is depicted in Fig.1.The Framework is designed for facilitating the security policy testing process. In the modeling stage, the left hand-side of the Fig.1, functional and security requirements are integrated as the base of modeling. Based on the integrated requirements the process of modeling is stepwise. An abstract model is first constructed and given by a number of refinement steps a sufficiently detailed model is obtained. The final system, the system under test (SUT), is an implementation of this detailed model. In the test cases generation stage, the right hand-side of the Fig.1, scenarios conformed to the model with the same level of abstraction are constructed in parallel. On the basis of the sufficiently refined scenarios the test cases are generated and selected. At last, according to the risk factor the test cases are organized and implemented on the system.

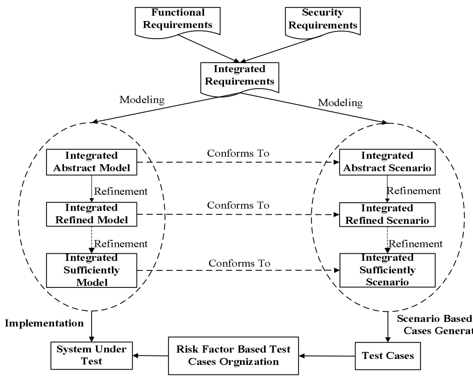


Figure 1. Scenario-Based test case generation framework.

III. Integrated Approach

A. Preliminaries

In this work the model of the security integrated system is given by IOSTS (Input/Output Symbolic Transition System).

Definition 1: (IOSTS) An IOSTS is a tuple $(V, P, \Theta, L, l_0, \Sigma, T)$ where *V* is a finite set of typed variables; *P* is a finite set of parameters; Θ is the initial condition, a predicate with variables in *V*; *L* is a nonempty, finite set of locations and $l_0 \in L$ is the initial location; Σ is a nonempty, finite alphabet, which is the disjoint union of a set $\Sigma^?$ of input actions, a set $\Sigma^!$ of output actions, and a set Σ^τ of internal actions; For each action $a \in \Sigma$, its signature $sig(a) = < p_1, ... p_k > \in P^k (k \in \mathbb{N})$ is a tuple of distinct parameters. *T* is a set of transitions. Each transition has:

- A location $l \in L$, called the origin of the transition; a location $l' \in L$ called the destination of the transition.
- An action $a \in \Sigma$ called the action of the transition.
- A predicate *G* with variables in $V \bigcup sig(a)$, called the guard.
- An assignment *A*, of the form $(x := A^x)_{x \in V}$ such that, for each $x \in V$, the right-hand side A^x of the assignment $x := A^x$ is an expression on $V \bigcup sig(a)$.

Definition 2: (IOLTS semantics of an IOSTS) The semantics of an IOSTS S = $(V, P, \Theta, L, l_0, \Sigma, T)$ is an IOLTS $[\![S]\!] = (S, S_0, \Lambda, \rightarrow)$, defined as follows:

- The set of states is $S = L \times \mathcal{V}$, $\mathcal{V}$ denote the set of valuations of the variables *V*.
- The set of initial states is $S_0 = \{< l_0, \nu > | \Theta(\nu) = true\}$, $\nu \in \mathcal{V}$.
- the set of action $\Lambda = \{< a, \pi > | a \in \Sigma, \pi \in \prod_{sig(a)}\}$, ($\prod$ denote the set of valuations of the parameters *P*, $\pi \in \Pi$),
- $\rightarrow$ is the smallest relation in $S \times \Lambda \times S$ defined by the following rule:

$$\frac{< l, \nu >, < l', \nu' > \in S \ <a, \pi> \in \Lambda \ t: <l, a, G, A, l'> \quad G(\nu, \pi) = \text{true} \ \nu' = A(\nu, \pi)}{< l, \nu > \xrightarrow{<a,\pi>} < l', \nu' >} \tag{1}$$

The rule says that the valued action $<a, \pi>$ takes the system from a state $< l, \nu >$ to a state $< l', \nu' >$ if there exists a transition *t*: $<l, a, G, A, l'>$ whose guard *G* evaluates to true when the variables evaluate according to ν and the parameters carried by the action *a* evaluate according to π. Then, the assignment *A* maps the pair (ν, π) to ν'.

The actions in S represent the events in the system. Let s= $(S, S_0, \Lambda, \rightarrow)$ be an input-output labeled transition system with s_0 in *S*. The behavior function, denoted by *beh*(*s*), is defined as $beh(s) = \{\Lambda \in \Lambda^* \mid s_0 \xrightarrow{\Lambda}\}$.

Definition 3: (Testing Scenario) denoted by *ts*, is collection of test sequences present in the behavior of IOLTS s. It is defined as: $ts \subseteq beh(s)$.

B. Security policy integration

In this section, we define algorithms to automatically integrate the security policies into the initial specification of the functional system in the form of an IOSTS by the way of a specific methodology. In this paper, we choose OrBAC [10] syntax to express security policies. The requirements are categorized into three types: permission, prohibition and obligation. Corresponding to these types of requirements, the integration algorithms are formed.

1) Permission policy integration: By definition, permissions relate to the existing states in the initial functional model. Considering the IOSTS, permissions correspond to one (or many) transitions and relate to the guard G. If the transition related to permission contains no guard, a guard has to be added. On the other hand, if a guard is already exists in the specification, it only needs to add the security constraints to the guard. The algorithm is as follows shown by Fig.2.

2) Prohibitions policy integration: Similar to the permissions integration, prohibitions integration consists either of adding a new guard or restraining an existing one.The corresponding algorithm is shown by Fig.3.

3) Obligations policy integration: In the obligation requirements integration, a new obligation activity is created. This new activity describes a new functional feature of the system. To make this possible, the new activity has to be initially expressed through IOSTS so that the algorithm can perform an automatic integration. The obligatory activity is added into the original functional model starting from the obligation start location to the obligation end location. The algorithm in Fig.4 identifies the transition which will be split into two (pre/post transitions).

```
Algorithm 1 Permission requirements integration
Condition: Corresponding to the location $l_i$, each permission applies to a role$_i$ ,action $a_i$ possibly to a $V_i$.
1 : if (∃ guard $G_i$) then
2:    $G_i := G_i \wedge (\vee_i((V_i \bigcup sig(a_i)) \wedge role_i))$
3: else
4:    create guard $G_i := \vee_i((V_i \bigcup sig(a_i)) \wedge role_i)$
5 : end if
```

Figure 2. Permission policy integration.

```
Algorithm 2 Prohibition requirements integration
Condition: Corresponding to the location $l_i$, each prohibition applies to a role$_i$ ,action $a_i$ possibly to a $V_i$.
1: if (∃ guard $G_i$) then
2:    $G_i := G_i \wedge (\vee_i(\neg(V_i \bigcup sig(a_i)) \wedge \neg role_i))$
3: else
4:    create guard $G_i := \vee_i(\neg(V_i \bigcup sig(a_i)) \vee \neg role_i)$
5: end if
```

Figure 3. Prohibition policy integration.

```
Algorithm 3 Obligation requirements integration
Condition: The initial transition $tr = <l, l', a, G, A>$ , The obligation start from SOL to EOL.
1: delete the transition tr
2: creat transition pre_tr and post_tr
3: if (∃guard $G$) then
4:   $G := G \wedge (\vee_i((V_i \cup sig(a_i)) \wedge role_i))$
5:   pre_tr = <$l$,SOL,$a$,$G$,$A$>
6:   post_tr = <EOL,$l'$,$a_o$,$G_o$,$A_o$>
7: else
8:   create guard $G := \vee_i((V_i \cup sig(a_i)) \wedge role_i)$
9:   pre_tr = <$l$,SOL,$a$,$G$,$A$>
10:  post_tr = <EOL,$l'$,$a_o$,$G_o$,$A_o$>
11: end if
```

Figure 4. Obligation policy integration.

C. *Testing scenario acquisition*

In this paper the testing scenario conforms to the integrated model is automatically refined with the integrated model refinement. We use the Event-B formalism for the description of the integrated model and Communicating Sequential Process (CSP) for testing scenario expressions.

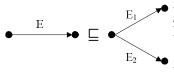
1) Integrated model refinement: The refinement process can be seen as a way to reduce non-determinism of the abstract specification and gradually introduce the implementable system.

In this paper, we are interested in how refinement affects the external behavior of a system under construction. Such external behavior can be represented by the actions of the IOLTS transitions, which can be reflected as a trace of observable events. These events can be used to form corresponding testing scenarios to produce test cases. Here we use the Event-B formalism for the description of the integrated model and present two different types of refinement called decomposition refinement and interposition refinement. The approach we proposed accepts more user defined refinement.

Definition 4: An Event-B machine denotes an Input-Output labelled transition system where

- S is set of Event-B states, where the state of an Event-B machine is a particular assignment of values to the Event-B variables;
- Λ is a set of actions which can be represented by events;
- The transition relation is constructed from single transitions of the form: $s \xrightarrow{ev} s'$, where s and s' are Event-B states and ev is the name of an event.
- s_0 is the state of an Event-B machine after initialization.

In decomposition refinement, one event operation is replaced by several operations, which amply describe the system reactions in different circumstances. Let us consider an abstract machine AM_D and a refinement machine AM_DR given below. It can be observed that an abstract event E is decomposed by the refined events E_1 and E_2. Any execution of E_1 and E_2 will correspond to some execution of abstract event E. It is also shown graphically in Fig.5.

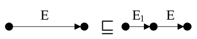


```
⊑  MACHINE AM_D
   ...
   EVENTS
     E = WHEN g(v)
         THEN S(v) END
   END

⊑  REFINEMENT AM_DR
   REFINES AM_D
   ...
   EVENTS
     E1 ref E = WHEN g(v) ∧ g1(v) THEN S1(v) END
     E2 ref E = WHEN g(v) ∧ g2(v) THEN S2(v) END
   END
```

Figure 5. Decomposition refinement.

In interposition refinement, new implementation details are introduced into the system in the form of new events that were invisible in the previous specification. Let us consider an abstract machine AM_I and a refinement machine AM_IR as shown in Fig.6.



```
⊑  MACHINE AM_I
   ...
   EVENTS
     E = WHEN g(v)
         THEN S(v) END
   END

⊑  REFINEMENT AM_IR
   REFINES AM_I
   ...
   EVENTS
     E = WHEN g(v) THEN S(v) END
     E1 = WHEN g1(v) THEN S1(v) END
   END
```

Figure 6. Interposition refinement.

2) Testing scenario refinement: In preliminaries, we have defined testing scenarios, which denote the behavior of system under test. As the possible valid execution paths that the system under test must follow, testing scenarios represent the expected functionalities of the system. In this paper, we present each such scenario by Communicating Sequential Process (CSP) expression. Each level of testing scenarios is corresponding with the integrated model. Therefore, knowing the way model M_i was refined by M_{i+1}, we can automatically refine scenario S_i into S_{i+1}. To check whether a scenario S_i is a valid scenario of its model M_i, i.e., model M_i satisfies ($\models$) scenario S_i, we use ProB model checker. ProB supports execution of Event-B specifications, guided by CSP expressions. The satisfiability check is performed at each refinement level as shown in the Fig.1.Here we will discuss how the refinement steps of the integrated model affect the scenarios.

Let us assume we are given an abstract model M_0 with three events: A, B and C, and a scenario S_0 represents the execution order of the three events. The CSP expression for scenario S_0 is given by

$$S_0 = A \rightarrow B \rightarrow C \rightarrow SKIP \tag{2}$$

When the model M_0 is refined by M_1, the scenario S_0 is refined accordingly by the method described below.

Corresponding to decomposition refinement, let us suppose an event B is refined using deposition refinement. As a result, it is decomposed into two events namely B_1 and B_2. As a CSP expression we can represent the new refined scenario S_1 as

$$\begin{gathered} S_1 = A \rightarrow ((B_1 \rightarrow CONT) \sqcap (B_2 \rightarrow CONT)) \\ CONT = C \rightarrow SKIP \end{gathered} \tag{3}$$

Where $\sqcap$ is an internal choice operator in CSP. The internal choice operator is used instead of external one because all of the events in the system have guards which make sure that only one event is enabled for execution at a time.

Corresponding to interposition refinement, a new event D is introduced in the system. The new scenario S_1 is represented as a CSP expression in the following:

$$S_1 = A \rightarrow B \rightarrow D \rightarrow C \rightarrow SKIP \quad (4)$$

In next section, we outline how scenarios are unfolded into test cases.

D. Test cases generation

In scenario based testing, we need to check that SUT has executed the corresponding action correctly for each event in the scenario. Here the semantics of the SUT is modeled by an IOLTS and the conformability is checked by the ProB model checker, which has the functionality to animate B specifications guided by the provided CSP expression. After each transition, present in the scenario, information about the changed system state is stored. The transition is led by the actions and represented by events in the scenario. So the transition trace is represented by a sequence of pairs $< e, s' >$, where e is an event and s' is the state after execution of event *e*. This information is stored during test case generation. For SUT these stored post-states become expected outputs of the system and act as a verdict for the testing.

For a finite number of events e_1, e_2.....e_n, present both in the model M and the SUT, a test case *t* of length n, defined in terms of test sequence, consists of an initial state s_0 and a sequence of pairs.

$$t = s_{0,} \{< e_1, s_1' >, < e_2, s_2' >, ..., < e_n, s_n' >\} \quad (5)$$

For a testing scenario, as formally defined in preliminaries, is a finite set of related test cases, i.e., a scenario *ts* is given as

$$ts = \{t_1, t_2, ..., t_n\} \quad (6)$$

E. Testing Process organization

After the generation of numerous test cases, the testing process should be organized in reason. As the security support for the functional systems, the protection has the same stress with the functions. This information can be useful in prioritize the security policy testing process, e.g., highly risked scenarios should be tested more to ensure the system reliability. In this paper, we define the risk factor of the testing scenario, which support the tests organization. Let η_{ts} is the risk weight of the scenario *ts*; p_n is the probability of risk occurrence associated with transition n; ε_n is the effects of the occurrence of p_n. The risk factor of the test scenario *ts* can be defined by:

$$R_{ts} = \eta_{ts}\left(\sum_n p_n \times \varepsilon_n\right) \quad (7)$$

Probability of occurrence depends on many factors, and one of the factors being the usage. The risk factor of a scenario is often determined by the impact to the safe operation of the system's function.

IV. CASE STUDY

A. Security integrated workflow system

In this section, we take the new company register workflow management system as our example. The initial system includes three states: application preparation, application verification and results publication. For security policies, they are described by Or-BAC. The initial functional model is described in Fig.7 (a). As one can notice, the initial model is voluntarily open and as a consequence, presents obvious security flaws. So we take security policies into consideration to protect the information within the initial system. The integrated model is described in Fig.7 (b).

B. Testing scenario acquisition and test case generation

For the sake of brevity, we just refine the integrated model one time and show the corresponding scenario refinement.

The initial security integrated model performs three basic tasks which is Application Preparation, Application Verification and Results Publication. To detail the verification process the system is repeatedly refined. Here in the refined model the application verification is refined into two new states in order to differentiate two kinds of the verification process, which is local verification and government verification. Our testing methodology can be applied to test this new scenario that result from the refinement. Fig.8 describes the refinement.

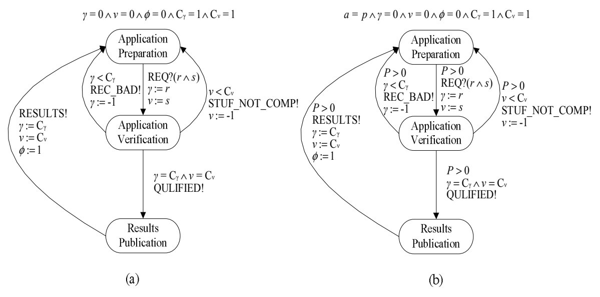

$\gamma = 0 \wedge v = 0 \wedge \phi = 0 \wedge C_\gamma = 1 \wedge C_v = 1$
Application Preparation
$\gamma < C_\gamma$ REC_BAD! $\gamma := -1$
REQ?$(r \wedge s)$ $\gamma := r$ $v := s$
$v < C_v$ STUF_NOT_COMP! $v := -1$
RESULTS! $\gamma := C_\gamma$ $v := C_v$ $\phi := 1$
Application Verification
$\gamma = C_\gamma \wedge v = C_v$ QULIFIED!
Results Publication
(a)
$a = p \wedge \gamma = 0 \wedge v = 0 \wedge \phi = 0 \wedge C_\gamma = 1 \wedge C_v = 1$
Application Preparation
$P > 0$ $\gamma < C_\gamma$ REC_BAD! $\gamma := -1$
$P > 0$ REQ?$(r \wedge s)$ $\gamma := r$ $v := s$
$P > 0$ $v < C_v$ STUF_NOT_COMP! $v := -1$
$P > 0$ RESULTS! $\gamma := C_\gamma$ $v := C_v$ $\phi := 1$
Application Verification
$P > 0$ $\gamma = C_\gamma \wedge v = C_v$ QULIFIED!
Results Publication
(b)

Figure 7. IOSTS of the integrated model.

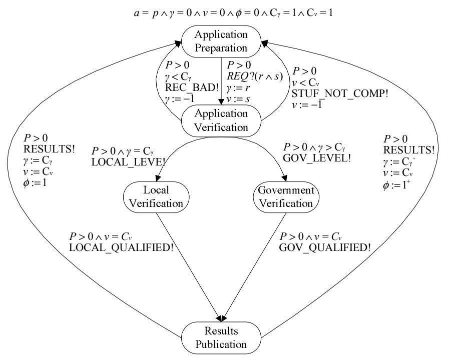

$a = p \wedge \gamma = 0 \wedge v = 0 \wedge \phi = 0 \wedge C_\gamma = 1 \wedge C_v = 1$
Application Preparation
$P > 0$ $\gamma < C_\gamma$ REC_BAD! $\gamma := -1$
$P > 0$ $REQ?(r \wedge s)$ $\gamma := r$ $v := s$
$P > 0$ $v < C_v$ STUF_NOT_COMP! $v := -1$
Application Verification
$P > 0$ RESULTS! $\gamma := C_\gamma$ $v := C_v$ $\phi := 1$
$P > 0 \wedge \gamma = C_\gamma$ LOCAL_LEVE!
$P > 0 \wedge \gamma > C_\gamma$ GOV_LEVEL!
$P > 0$ RESULTS! $\gamma := C_\gamma^+$ $v := C_v$ $\phi := 1^+$
Local Verification
Government Verification
$P > 0 \wedge v = C_v$ LOCAL_QUALIFIED!
$P > 0 \wedge v = C_v$ GOV_QUALIFIED!
Results Publication

Figure 8. Refinement of the integrated model.

The initial Event-B machine and the refined machine are specified as follows:

```
MACHINE WorkFlowSystem
SETS Record; Stuff ; Standard; Authority
VARIABLES γ; υ; Cγ; Cv; P
INVARIANT γ ⊆ Record ;υ ⊆ Stuff ;
              Cγ ⊆ Standard ;Cv ⊆ Standard;
            P ⊆ Authority
INITIALIZATION γ := ∅;υ := ∅;
EVENTS
  submit = WHEN P > 0 THEN γ := r ∪ v := v END;
  verification_pass = WHEN γ ≥ Cγ ∧ v ≥ Cυ ∧ P > 0 THEN SKIP END;
  verification_failed = WHEN γ < Cγ ∧ v < Cυ ∧ P > 0 THEN γ := − 1 ∧ v := − 1 END;
  result = WHEN γ ≥ Cγ ∧ v ≥ Cυ ∧ P > 0 THEN γ := Cγ ∧ v := Cυ ∧ Φ := 1 END;
END


REFINEMENT WorkFlowSystem1 REFINES WorkFlowSystem
...
EVENTS
...
  LocalVerification ref verification_pass = WHEN γ = Cγ ∧ P > 0 THEN SKIP END;
  GovernmentVerification ref verification_pass = WHEN γ > Cγ ∧ P > 0 THEN SKIP END;
...
END
```

Figure 9. The initial Event-B machine and the refined machine.

The corresponding testing scenario for the specification above is expressed as the following CSP expression.

```
Initial Testing Scenario:
   WorkFlowSystem = Application Preparation → Application Verification → Results Publication → SKIP
Refined Testing Scenario:
   WorkFlowSystem1 = Application Preparation → Application Verification
   Application Verification = Local Verification → Result Publication → SKIP
   Application Verification = Government Verification → Result Publication → SKIP
```

Figure 10. CSP expression of the testing scenario.

These CSP expressions are finally unfolded into test cases by the methodology described in previous section. These test cases are applied on the implementation to test the security policies.

C. Testing Process organization

To better test the security integrated workflow system, the security policy testing process is prioritized according to the risk factor. In this case study, we compared two scenarios: Government Verification Scenario and Local Verification Scenario. They have the same times of transitions n but different risk weight. According to the formula we present in previous section, the risk factor of government verification scenario $R_{gvs} = \eta_{gvs}(\sum_{n=5} p_n \times \varepsilon_n)$ is higher than the local verification scenario $R_{lvs} = \eta_{lvs}(\sum_{n=5} p_n \times \varepsilon_n)$. The testing process is organized accordingly.

V. Conclusion

The framework for testing security policies is presented in this paper. On the basis of security policy integration algorithms, the security integrated system is formed and the corresponding testing scenarios are constructed at the same time. Then the method of test cases generation based on testing scenario is proposed. At last the testing process is organized according to the risk factor of the testing scenarios. We describe the process of our framework through a representative case study and show the validity of this method.

In the future work we will improve the framework in three aspects. First, we will extend the security policy integration algorithms to be able to take into account more complex constraints in security policies (such as temporal constraints). Second, more refinement types will be included into our method. Finally, we plan to build an execution environment where the test cases can be executed on the system under test in a controlled manner corresponding to the risk factor of the testing scenarios.

Acknowledgment

This paper is supported by the National High Technology Research and Development Program (863Program) of China (No.2009AA01Z433).

References

[1] T. Ahmed and A. R. Tripathi, “Static verification of security requirements in role based CSCW systems,” Proc. ACM Symp. Access Control Models Technol (SACMAT 2002), ACM Press, June. 2003, pp. 196-203, doi:10.1145/775412.775438.

[2] E. C. Lupu and M. Sloman, “Conflicts in policy-based distributed systems management,” IEEE Trans. Softw. Eng, vol. 25, Nov.-Dec. 1999, pp. 852-869, doi:10.1109/32.824414.

[3] A. Masood, A. Ghafoor, and A. Mathur, “Scalable and effective test generation for access control systems that employ RBAC policies,” Technical Report SERC-TR-285, Purdue University Electrical and Computer Engineering Department, 2005.

[4] P. Ammann and W. Ding and D. Xu, “Using a Model Checker to Test Safety Properties,” Proceedings Seventh IEEE International Conference on Engineering of Complex Computer Systems, IEEE Computer Society, June. 2001, pp. 212-221, doi:10.1109/ICECCS. 2001. 930180.

[5] Y. L. Traon, T. Mouelhi, and B. Baudry, “Testing security policies: Going beyond function testing,” Proc. Int. Symp. Softw. Reliab. Eng, IEEE Computer Society, Nov. 2007, pp. 93-102, doi: 10.1109/ ISSRE.2007.29.

[6] A. Pretschner, T. Mouelhi, and Y. L. Traon, “Model-based tests for access control policies,” Proceedings First IEEE International Conference on Software Testing, Verification and Validation (ICST 2008), IEEE Press, Apr. 2008, pp. 338-347, doi:10.1109/ICST. 2008.44.

[7] W. Mallouli, J.-M. Orset, A. Cavalli, N. Cuppens, and F. Cuppens, “A Formal Approach for Testing Security Rules,” Proc. ACM Symp. Access Control Models Technol (SACMAT 2007), ACM Press, June. 2007, pp.127-132, doi:10.1145/1266840.1266860.

[8] E. Martin, T. Xie, “Automated test generation for access control policies via change-impact analysis,” Proceedings. Third International Workshop on Software Engineering for Secure Systems, IEEE Press, May. 2007, pp. 5-11.

[9] Y. Falcone, J. Fernandez, L. Mounier, and J. Richier, “A compositional testing framework driven by partial specifications,” Lect. Notes Comput. Sci, Springer Press, vol. 4581, pp. 107–122.

[10] A. A. E. Kalam, R. E. Baida, P. Balbiani et al. “Organization Based Access Control,” IEEE 4th International Workshop on Policies for Distributed Systems and Networks (Policy 2003), IEEE Comput. Soc, June. 2003, pp. 120-31, doi:10.1109/POLICY.2003.1206966.