A Formal Approach for Testing Security Rules

Wissam Mallouli, Jean-Marie Orset, Ana Cavalli
GET/INT Evry, SAMOVAR, 9 rue Charles Fourier
91011 Evry Cedex, France
{wissam.mallouli, jean-marie.orset, ana.cavalli}@int-evry.fr

Nora Cuppens, Frederic Cuppens
GET/ENST Bretagne, 2 rue de la chataigneraie
35512 Cesson Sevigne Cedex, France
{nora.cuppens, frederic.cuppens}@enst-bretagne.fr

ABSTRACT

Nowadays, security policies are the key point of every modern infrastructure. The specification and the testing of such policies are the fundamental steps in the development of a secure system since any error in a set of rules is likely to harm the global security. To address both challenges, we propose a framework to specify security policies and test their implementation on a system. Our framework makes it possible to generate in an automatic manner, test sequences, in order to validate the conformance of a security policy. System behavior is specified using a formal description technique based on extended finite state machine (EFSM) [13]. The integration of security rules within the system specification is performed by specific algorithms. Then, the automatic test generation is performed using a dedicated tool, called SIRIUS, developed in our laboratory. Finally, we briefly present a weblog system as a case study to demonstrate the reliability of our framework.

Categories and Subject Descriptors

I.6.4 [**Simulation and Modeling**]: Model Validation and Analysis; I.6.5 [**Simulation and Modeling**]: Model Development—*Modeling methodologies*

General Terms

Algorithms, Design, Security, Languages, Verification

Keywords

Security Policy, OrBAC, EFSM, Verification and Testing, Test Generation, SDL

1. INTRODUCTION

Security is a critical issue in dynamic and open distributed environments such as World Wide Web or wireless networks.

To ensure that a certain level of security is always maintained, the system behavior must be restrained by a security policy. A security policy is a set of rules that regulates the nature and the context of actions that can be performed within a system, according to specific roles. As an example, such policy can tackle the interactions between a network infrastructure and Internet or manage accounts and rights toward an operating system or a database. Generally, a security policy is written by the mean of a natural language specification, containing statements such as "this file must be accessible only to authorized users" or "all ports are closed except for 21 (ftp), 22 (ssh) and 80 (www)".

The main problem is that it is quite difficult to verify whether a system implementation conforms to its policy. However, if one cannot ensure this conformance, the global security cannot be guaranteed anymore. Most current works only concentrate on defining meta-languages in order to express security policies and provide unambiguous rules. [3] and [9] are typical examples of such generic policy description models. Indeed, they do not depend on the functional specification of the system. They suggest concepts to describe the security policy independently of the system implementation. Once the security policy is formally specified, it is essential to prove that the target system implements this policy by (1) injecting this policy in the system considered or (2) specifying formally the target system and generating proofs that this system implements the security policy or (3) by considering several strategies of formal tests.

In this paper, we propose an approach that makes it possible to validate security rules. This approach manipulates three different inputs: a functional specification of the system based on a well-know mathematically-based formalism: Extended Finite State Machine (see section 3.2), a specification of the security policy (based on the OrBAC model [3]) that we wish to apply to this system and finally an implementation of the system. We want to obtain a new specification of the system that takes into account the security policy (we can call it: secure functional specification), and then to generate tests to check whether the implementation of the system conforms with the secure functional specification.

This paper distinguishes itself from classical conformance testing work (see for instance [4]) by several significant differences. In fact, we propose an approach to integrate security rules within the functional specification of a system. Thus, we describe how modalities such as prohibitions, authorizations and obligations can be integrated in an EFSM by restricting predicates or by adding transitions and states. Then, we propose a method to automatically derive test sequences from a set of rules as well as an approach to restrict the number of test objectives required to perform verification. Besides, we do not address all issues. Checking the consistency of the security policy is out of the scope of this paper. We assume that this issue has been checked. There are several techniques to achieve this goal (see for instance [7]).

The remainder of this paper is organized as follows. In section 2, we discuss the related work tackling with the description and the validation of security policies. Section 3 presents the basic notions used for the management of security rules. In section 4, we expose the approach to integrate these security rules within an existing specification in EFSM as well as the relative algorithms. In section 5, we present a case study: a weblog with security features, as well as the results through generated test objectives. Finally, section 6 presents the conclusion and introduces the future work.

2. RELATED WORK

Most work related to security policy can be mainly divided into two parts: the description of the policy itself and the verification of the rules. Until recently, in many systems, there was no real policy specification, only a description in terms of low-level mechanisms such as access control lists. Thereafter, the analysis of access-control leads to define a number of access control models, which could provide a formal representation of security policies and in some cases, allow the proof of properties about access control. With the great majority of models, security rules are defined with three main modalities (permission, prohibition and obligation) that express the possible constraints on the behavior of the system [8]. Among these models, we can mention for instance the Policy Description Language (PDL) [14], Ponder [9] and OrBAC (Organisationnal Based Access Control) [3].

Concerning the verification of the rules, most work in the field deals with testing of firewall rules. First proposals consisted in performing testing of rules by hand. This implies that test construction is performed by human experts who focus on detecting traces of known attacks. Most recently, research tended to concentrate on the verification of security rules in order to detect errors or misconfigurations such as redundancy, contradiction or collision [11, 12].

Some approaches propose to focus on validation by checking the conformance of a system with respect to a security policy. In [15], authors show how an organisation's network security policy can be formally specified in a high-level way, and how this specification can be used to automatically generate test cases to test a deployed system. By contrast to other firewall testing methodologies, such as penetration testing, this approach tests conformance to a specified policy. These test cases are organisation-specific - i.e. they depend on the security requirements and on the network topology of an organisation - and can uncover errors both in the firewall products themselves and in their configuration. However, this model is limited to the network management and specifically to network and transport layer of the TCP/IP stack. Moreover, it is still a theoretical approach and there exists no tool yet to automate the testing process and to evaluate its effectiveness on a real case-study.

In [10], the authors choose another approach to achieve testing of network security rules. They express the network behavior using labeled transition systems formulae. Then, for each element of their language and each type of rule, they propose a pattern of test called a tile. Then, they combine those tiles into "complete" test cases for the whole rule to perform validation.

Our approach distinguishes itself from these propositions by the assumptions on the policy and the method used to generate test sequences. First, we make no assumption about the description language of the policy. Instead, we propose a framework to specify rules in a formal form, so that we can apply them on our mathematical model. Then, we generate the whole set of test cases and this, in an automatic manner.

3. BASICS FOR SECURITY RULES INTEGRATION

3.1 Assumptions

In this section, we define the relation between a security policy and the specification of a system and give the assumptions we rely on. First, we consider two parts in our approach: the initial system and the security policy. Initial system refers to the functionalities with no security consideration. Later, a new context can evolve to meet security considerations. Within this new one, the initial system is not valid anymore since it cannot satisfy new requirements. It has to be completed with a security policy, to fit the new context.

In this paper, we propose to automatically integrate the security policy rules into the initial specification in the form of an EFSM, by the way of a specific methodology. We take as an assumption that the specification of the security policy is correct. This means that we do not search within the policy for conflicts or redundancies. On the other hand, we consider that the rules have to be provided by an expert of the organization, who guarantees their completeness and soundness. Finally, the last assumption concerns the semantics of the policy description language. We consider that it can always be translated in the form of an alphabet acceptable by the formalism. That means we consider that no information is lost while deriving the formal model from the set of rules.

Now that the assumptions are defined, we can introduce the basic concepts and describe our algorithm.

3.2 The EFSM formalism

In order to model the initial system as well as the security policy, we choose to use the Extended Finite State Machine (EFSM) formalism. This formal description is used not only to represent the control portion of a system but also to properly model the data portion, the variables associated as well as the constraints which affect them.

Definition 1. *An Extended Finite State Machine M is a 6-tuple $M = < S, s_0, I, O, \vec{x}, Tr >$ where S is a finite set of states, s_0 is the initial state, I is a finite set of input symbols (eventually with parameters), O is a finite set of output symbols (eventually with parameters), $\vec{x}$ is a vector denoting a finite set of variables, and Tr is a finite set of transitions. A transition tr is a 6-tuple $tr =< s_i, s_f, i, o, P, A >$ where s_i and s_f are the initial and final state of the transition, i and o are the input and the output, P is the predicate (a boolean expression), and A is an ordered set (sequence) of actions.*

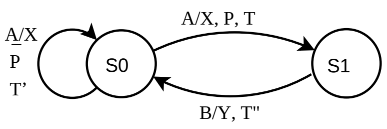

Figure 1: Example of a simple EFSM with two states.

We illustrate the notion of EFSM through a simple example described in Figure 1. The ESFM shown in Figure 1 is composed of two states S_0, S_1 and three transitions that are labeled with two inputs A and B, two outputs X and Y, one predicate P and three tasks T, T' and T''. The EFSM operates as follows: starting from state S_0, when the input A occurs, the predicate P is tested. If the condition holds, the machine performs the task T, triggers the output X and passes to state S_1. If P is not satisfied, the same output X is triggered but the action T' is performed and the state loops on itself. Once the machine is in state S_0, it can come back to state S_1 if receiving input B. If so, task T'' is performed and output Y is triggered.

3.3 The OrBAC syntax

Most security description languages use the same basic concepts (that are obligation, permission and prohibition) to describe access control rules. In this paper (framework description and case study), we choose to rely on the OrBAC [3] syntax to express the security policy (as an input).

OrBAC is an access and usage control model that allows an organization to express its security policy. For this purpose, OrBAC defines two abstraction layers. The first one is called abstract and describes a rule as a *role* having the permission, prohibition or obligation to perform an *activity* on a *view* in a given *context*. A *view* is a set of objects to which the same security rules apply. The second level is the concrete one. It is derived from the abstract level and grants permission, prohibition or obligation to a user to perform an *action* on an *object*. Thus, according to our syntax, a typical security rule has the following form:

Obligation (S, R, A, V, C)

This rule means that within the system S, the role R is obliged to perform the activity A targeting the objects of view V in the context C. The principle is similar for permission and prohibition.

We argue that starting from such simple syntax, it is easy to apply our approach to other languages such as Ponder or PDL. In that manner, our framework remains independent of any model for security description. The only restriction is that some specific complementary conditions have to be taken into account in the formal specification of the security rules:
- A rule context is divided into two parts: an *EFSM context* with conditions related to the position in the EFSM (eg. input=signal1) and a *variables context* with conditions related to variables values(eg. variable=value).
- If the roles and *variables context* are not already defined in the initial specification, precise definitions have to be added (type, default value, etc.).
- As an assumption, an activity within an obligation must be a new partial EFSM which starts with an obligation state *(OS)* and ends with one or many end obligation states *(EOS)*. All the new variables and signals have to be defined.
- An activity related to a permission or a prohibition must correspond to one transition at once (can be on several distinct transitions but not on a sequence). Otherwise, we cannot determine the predicate to be relaxed or restrained.
- The *EFSM context* is mandatory in an obligation.

4. INTEGRATION METHODOLOGY

In this section, we define algorithms to automatically integrate the security policy rules into the initial specification in the form of an EFSM, by the way of a specific methodology. The process is twofold. At first, the algorithm seeks for the rules to be applied on each transition of the specification and derives a simple automaton from this set of rules. Then, it integrates the automaton within the initial specification. At the end of the process, this integration will generate a new specification that takes into account the security requirements.

It is possible that the security policy defines some new concepts that cannot be directly apprehended by the initial specification. In particular, a rule can express an activity that does not exist in the specification (new role, different object, new action, etc.). In that case, the new activity must firstly be created in the specification. That means some new states might be created in order to make the EFSM accept the new elements. In the security policy, this is specified by an obligation.

The beginning of the algorithm is the same for the three kinds of rules (permission, prohibition and obligation). It parses the EFSM specification and for each transition, it identifies the rules that map the activity (which can be a state, an input, an output, a task or a logic combination of these features) and the *EFSM context* in the case of permissions and prohibitions, and only the *EFSM context* in the case of obligations (Indeed, we consider in our work that the activity introduced by an obligation is a new one). If no rule maps the transition, the default one will be applied. Once rules identified for each transition, we can proceed to their integration. Notice that several rules may apply to the same transition. In this case, the algorithm is recursively applied to each relevant rule.

4.1 Permissions integration

The permissions are the easiest modality to integrate. Indeed, by definition, a permission does not define what is *possible* to do but instead, what is *permitted* to do. Thus, permissions relate to activities which already exist in the initial system. Considering the EFSM, a permission corresponds to one (or many) transitions. If the transition related to a permission contains no predicate, a predicate has to be added. On the other hand, if a predicate is already defined in the specification, it only needs to be further restrained (the condition is stronger).

Algorithm 1 Permissions integration

```
Require: The transition $Tr$ that maps the permissions.
    Each $permission_i$ applies to a $role_i$ and possibly to a
    $variables\ context_i$ (may be empty).
1: if (∃ associated predicate $P$) then
2:     $P := P \wedge (\vee_i(variables\ context_i \wedge role_i))$
3: else
4:     create predicate $P := \vee_i(variables\ context_i \wedge role_i)$
5: end if
```

The Figure 2 gives an example. In the left transition, the system can pass from S_1 to S_2 if P is true, performing the task T. If the permission involves a role R, allowed to perform task T in the context C, the transition will be modified by strengthening the predicate, as in the left transition.

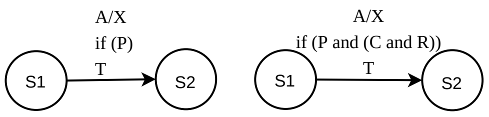

Figure 2: Permission (S, R, T, _ , C)

4.2 Prohibitions integration

Like permissions integration, prohibitions integration consists either of adding a new predicate or restraining an existing one (it becomes stronger).

Algorithm 2 Prohibitions integration

```
Require: The transition $Tr$ that maps the $prohibition_i$.
1: if (∃ associated predicate $P$) then
2:     $P := P \wedge_i (\neg variables\ context_i \vee \neg role_i)$
3: else
4:     create predicate $P := \wedge_i(\neg variables\ context_i \vee \neg role_i)$
5: end if
```

Here is an example in the Figure 3. In the left transition, the system can pass from S_1 to S_2 if P is true, performing the task T. If the rule specifies that a role R is prohibited to perform task T in the context C, the transition will be modified by the restriction of the predicate, as in the left transition.

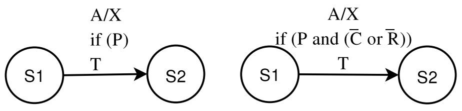

Figure 3: Prohibition (S, R, T, _ , C)

4.3 Obligation integration

In our model, we take as an assumption that an obligation implies the creation of a new activity (if an activity already exists in the initial specification, it only has to be allowed or denied). This new activity describes a new functional feature of the system. To make this possible, the new activity has to be initially expressed through a partial EFSM so that the algorithm can perform an automatic integration of the rule within the EFSM. In this manner, an obligatory activity is a partial EFSM which begins by a *starting obligation state* and ends with an *end obligation state*. Thanks to the *EFSM context* of the obligation, the algorithm identifies the transition which will be split into two (pre/post transitions), to insert the partial EFSM of the obligation. Then, the algorithm needs to know how the components of this transition will be distributed relatively to the obligation (pre/post transitions). This can be determined using the *cut point*, that corresponds to the *last component* of the initial transition (state, input, task or output, but not a predicate) which maps the *EFSM context*. Each component until this *cut point* (included) will be attributed to the pre-transition (through the obligation) while other ones will be attributed to the post-transition. Finally, a last transition has to be added to bypass the obligation in the case the initial predicate is not satisfied (see Algorithm 3).

Algorithm 3 Obligation integration

```
Require: The transition $tr =< S_1, S_2, A, X, P, t_1...t_n >$
    that maps the obligation with an activity specified by
    the mean of an EFSM with $OS$ as a first state and $EOS$
    as a last one
1: for all (transitions from $OS$) do
2:     if (∃ associated predicate $Q$) then
3:         $Q := Q \vee (variablescontext \wedge role)$
4:     end if
5: end for
6: determine the cut point $C_{ut}P_{oint}$
7: delete the transition tr
8: create transitions $C_1$, $C_2$ and $C_3$ such that
9: if ($C_{ut}P_{oint} == S_1$) then
10:     $C_1 :=< S_1, OS, -, -, -, - >$
        if (¬ ∃ $EOS$ state in $M$) then $C_2 :=< EOS, S_2, A, X,$
        $P, t_1...t_n >$ end if
        $C_3 :=< OS, S_2, A, X, \neg variables\ context \vee \neg role,$
        $t_1...t_n >$
11: else
12:     if ($C_{ut}P_{oint} == A$) then
13:         $C_1 :=< S_1, OS, A, -, P, - >$
            if (¬ ∃ $EOS$ state in $M$) then $C_2 :=< EOS, S_2, -,$
            $X, -, t_1...t_n >$ end if
            $C_3 :=< OS, S_2, -, X, \neg variables\ context \vee \neg role,$
            $t_1...t_n >$
14:     else
15:         if ($C_{ut}P_{oint} == t_i$ where $i \in \{1, ..., n\}$) then
16:             $C_1 :=< S_1, OS, A, -, P, t_1...t_i >$
                if (¬ ∃ $EOS$ state in $M$) then $C_2 :=< EOS, S_2,$
                $-, X, -, t_{i+1}...t_n >$ end if
                $C_3 :=< OS, S_2, -, X, \neg variables\ context \vee \neg role,$
                $t_{i+1}...t_n >$
17:         else
18:             if ($C_{ut}P_{oint} == X$) then
19:                 $C_1 :=< S_1, OS, A, X, P, t_1...t_n >$
                    if (¬ ∃ $EOS$ state in $M$) then $C_2 :=< EOS,$
                    $S_2, -, -, -, - >$ end if
                    $C_3 :=< OS, S_2, -, -, \neg variables\ context \vee$
                    $\neg role, - >$
20:             end if
21:         end if
22:     end if
23: end if
24: minimize the resulting EFSM by deleting silent transi-
    tions (without input nor output nor action)
```

The Figure 4 shows an example of the process. In this case, the initial transition is "$A/X, if\ P, T$". The new activity is a partial EFSM with two states (OS and EOS) and one transition characterised by the input B, the task T' and the output Y. According to the EFSM context, the *cut point* is the Input A. By the following, the transition $C1$ (pre-transition) is defined by the input A and the predicate P. The transition $C2$ (post-transition) is defined by the task T and the output X. Obligation integration is shown in the Algorithm 3.

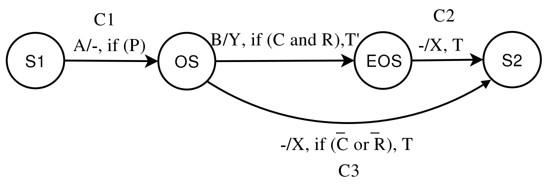

Figure 4: Obligation (S, R, new activity, _ , (Input = A) and C).

4.4 Integration result

The security rules integration allows us to obtain a new specification of the system that takes into account the security policy. This formal specification is described in the EFSM formalism that is a well adapted one to model communicating systems. Using SDL (Specification Description Language) [5] based on this formalism we can easily derive test sequences to check whether the implementation of the system conforms with the secure functional specification. The classical test generation methodology is presented in the ISO9646 standard [2].

5. CASE-STUDY: A WEBLOG

5.1 Weblog description

To prove the effectiveness of our framework, we choose to carry out a case-study on a weblog (also called blog): a blog is a website used to post stories or news (such as in a journal or diary) to make them available for reading by any other party. Here, we consider at first, a simple weblog with various features, such as those commonly used on the World Wide Web. First of all, the service is open. That means anyone cannot only read but also post a content in the form of news or stories. Then, other readers might add comments relating to any content. Possibly, a blogger[1] might take the decision to delete a posted content depending of its freshness or its relevance. The weblog can thus be seen as a mutable list of stories, which constitute themselves a content associated with a mutable, possibly empty, list of comments.

As one can notice, the initial model is voluntarily open and as a consequence, presents obvious security flaws. Indeed, no authentication is performed so that any user can delete numerous posted contents, which leads to a kind of denial of service attack. To tackle this problem, we specify a security rule whose goal is to protect the information within the organization, by preventing illegitimate users to delete any content.

For this purpose, the security rules will define a hierarchy of users, by defining three different roles. The first one is the administrator (*admin*). It has the responsibility to maintain the global organization of the website. It is the only one authorized to delete a posted content or to suspend the activity of the website (in the case of maintenance for example). Beside the administrators, the policy will define another role: the bloggers. Bloggers are users which can post stories and also commentaries, relating to their own content or not. Moreover, they are allowed to perform the *delete* action but only on their own content. Finally, the normal users (called visitors or anonymous) can only read stories and commentaries; that means *delete* and *write* actions are prohibited for them.

To specify the Web application, we used *ObjectGEODE* [17] and *SIRIUS* [6] tools. They are based on a language specifically dedicated to the formal specification of interactive systems: SDL which is a specification and description language standardized as ITU[2] (International Telecommunication Union) Recommendation Z.100.

5.2 The security policy specification and integration

Considering the syntax of the OrBAC language, the global system on which relate all modalities in the security policy is the *Website*. The objects will naturally correspond to each of the components defined initially, that are the *blog*, the *posts* and the *comments*. In the same manner, the first actions will be chosen among the existing in the initial system: *read*, *write* and *delete*. As for the context, it defines the set of conditions expressed by a rule, which have to hold to allow an activity. Based on these considerations, we specified 24 rules constituting the policy that manages the security of our weblog. Here are 3 different examples of theses rules:

1. Obligation(Website, anonymous, Authentication, _ , input = AddPostReq)
2. Permission(Website, admin, 'Reading Blog', blog,_)
3. Permission(Website, anonymous, 'Reading Blog', blog, _)

After the specification of all rules, we step to the second phase of the process, which is their integration within the extended finite state machine. This integration process respects the methodolgy described in section 4 and leads to a secure functional specification.

5.3 Test generation

5.3.1 Fixing the test objectives

In the Weblog case study, our aim was to test the security rules. The first idea was to define for each rule one or several test objectives. However, we noticed that one generated test sequence can verify more than one security rule. Then, we tried to minimize the number of the test objectives to test the entire modified transitions. We specified at the end 17 test objectives that represents more than 94% of the specification transitions.

5.3.2 Generation with SIRIUS

SIRIUS is based on Hit-or-Jump [6], an algorithm especially used for components testing to perform test sequences generation through the specification. This research is guided by objectives which are illustrated by predicates on transitions (and written in SDL). Research in the partial reachability graphs is performed in depth, width or both at the same time, and is restricted by a limited depth. In order to initialize the generation of test sequences, several parameters are necessary. Four main files must be developed. The first is the service specification (component to be tested), the second allows the initialisation of some variables if necessary,

[1]Contraction for weblog user.

[2]http://www.itu.int

the third one mentions the stop conditions (i.e. test objectives) and finally the last one allows the expert to guide the system at the beginning of the simulation, this file is called preamble. This last one is very important; it allows reducing in a consequent way the length of the test sequence and the duration of its generation.

5.4 Discussions

The results are obtained after a BFS (breadth-first search) exploration of the reachability graph. This choice is due to the specificity of the weblog service which has to take into account, after each transition, all the possible inputs injected by the user to analyze them and generate the right output. The test objectives are reachable via short sequences according to the specification size of the system and do not need a DFS or BDFS exploration that tries to search in depth of the reachability graph. The generated test sequences are usable since they correspond to the test strategy mentioned in section 5.3.1, they can be produced in TTCN [16] and MSC [1] standard notations facilitating their portability.

6. CONCLUSION AND FUTURE WORK

In this paper, we presented a framework for testing a security policy in a formal manner. We proposed an algorithm to automate the specification of security rules as an EFSM. Then, we presented a scheme to derive test objectives from the rules formal specification in order to test the conformance of a security policy with respect to its implementation. We described the process of our framework through a representative case-study. We showed that our approach allows the specification of various modalities such as obligation, permission and prohibition and makes it possible to obtain relevant test objectives. It is important to notice that our algorithm allows us to verify that the security policy has no gap (missing rules) when no rule is found to be applied on a functional specification transition.

As a future work, we are currently investigating several approaches to enhance this framework. At first, we consider generalizing the algorithm so that it can accept policies specified in other languages (Ponder, PDL, etc.). Then, we will extend our integration scheme to be able to take into account more complex modalities (such as temporary/definitive delegations), temporal rules (which denote actions limited in time) and interoperability of rules (that is testing if a rule can be deployed on several systems). Finally, we are working on a test optimization approach which would allow us to test only new transitions created/modified by the integration process.

7. REFERENCES

[1] *IUT-T Rec. Z. 120 Message Sequence Charts, (MSC).* Geneva, 1996.

[2] I. 9646-1. *Information Technology - Open Systems Interconnection - Conformance testing methodology and framework Part 1: General Concepts.*

[3] A. Abou El Kalam, R. E. Baida, P. Balbiani, S. Benferhat, F. Cuppens, Y. Deswarte, A. Miège, C. Saurel, and G. Trouessin. Organization Based Access Control. In *4th IEEE International Workshop on Policies for Distributed Systems and Networks (Policy'03)*, June 2003.

[4] A. V. Aho, A. T. Dahbura, D. Lee, and M. U. Uyar. An optimization technique for protocol conformance test generation based on uio sequences and rural chinese postman tours. In *IEEE Computer Society Press.* pages 427–438, 1995.

[5] A. Cavalli and D. Hogrefe. Testing and validation of SDL systems : Tutorial. In *SDL'95 forum*, 1995.

[6] A. Cavalli, D. Lee, C. Rinderknecht, and F. Zaïdi. Hit-or-Jump: An Algorithm for Embedded Testing with Applications to IN Services. In *Formal Methods for Protocol Engineering And Distributed Systems*, pages 41–56, Beijing, China, october 1999.

[7] F. Cuppens, N. Cuppens-Boulahia, and M. B. Ghorbel. High-level conflict management strategies in advanced access control models. In *Workshop on Information and Computer Security (ICS)*, Timisoara, Roumania, September 2006.

[8] N. Damiannnou, A. Bandara, M. Sloman, and E. Lupu. *Handbook of Network and System Administration*, chapter A Survey of Policy Specification Approaches. Elsevier, 2007 (to appear).

[9] N. Damianou, N. Dulay, E. Lupu, and M. Sloman. The ponder policy specification language. In *POLICY '01: Proceedings of the International Workshop on Policies for Distributed Systems and Networks*, pages 18–38, London, UK, 2001. Springer-Verlag.

[10] V. Darmaillacq, J.-C. Fernandez, R. Groz, L. Mounier, and J.-L. Richier. Test generation for network security rules. In *TestCom*, pages 341–356, 2006.

[11] J. García-Alfaro, F. Cuppens, and N. Cuppens-Boulahia. Analysis of policy anomalies on distributed network security setups. In *ESORICS*, pages 496–511, 2006.

[12] J. García-Alfaro, F. Cuppens, and N. Cuppens-Boulahia. Towards filtering and alerting rule rewriting on single-component policies. In *SAFECOMP*, pages 182–194, 2006.

[13] D. Lee and M. Yannakakis. Principles and methods of testing finite state machines - A survey. In *Proceedings of the IEEE*, volume 84, pages 1090–1126, 1996.

[14] J. Lobo, R. Bhatia, and S. A. Naqvi. A policy description language. In *AAAI/IAAI*, pages 291–298, 1999.

[15] D. Senn, D. A. Basin, and G. Caronni. Firewall conformance testing. In *TestCom*, pages 226–241, 2005.

[16] E. TTCN-3. *TTCN-3 – Core Language.*

[17] Verilog. *ObjectGEODE Simulator, Reference manual*, 1997.