Test Generation for Network Security Rules

Vianney Darmaillacq[1], Jean-Claude Fernandez[2], Roland Groz[1], Laurent Mounier[2], and Jean-Luc Richier[1] ⋆ ⋆⋆

[1] Laboratoire LSR-IMAG
BP 72, 38402 St Martin d'Hères, France
[2] Laboratoire Vérimag
Centre Equation - 2, avenue de Vignate, 38610 Gières, France

Abstract. Checking that a security policy has been correctly deployed over a network is a key issue for system administrators. Although this is a kind of conformance testing, there are a number of significant differences with the framework of such standards as IS9646. We propose a method to derive tests from a policy expressed as a set of rules, with a single modality. For each element of our language and each type of rule, we propose a pattern of test, which we call a tile. We then combine those tiles into a test for the whole rule.

1 Introduction

Network and system administrators are in charge of implementing and controlling the security policies of their organizations. Enforcing a policy typically relies on the adequate configuration of Policy Enforcement Points (PEP) in dedicated equipment (such as firewalls, ciphering chips) or specific software (e.g. account managers, mail scanners). Since most networks and systems would undergo daily changes, maintaining the consistency of the rules actually implemented by the PEPs and their conformance to the specified security objectives and the rules expressed in the policy is not an easy task.

One way to guarantee a correct policy enforcement is to derive configurations from a description of the policy (top-down approach). In order to do that systematically, the description must be formal enough to provide unambiguous rules and translations of them into configurations of security devices. Since most policies consist of combination of rules with various scopes and potentially conflicting modalities (e.g. restricting access for generic subjects but authorizing it for specific categories), the descriptions usually include constructs or semantic rules to solve conflicts between policy statements. Ponder [5] and OrBAC [1] are typical such description languages with associated methods to allow deployment on networks.

In this paper, we investigate another approach (bottom-up). We consider that testing a given network configuration for compliance with a stated policy is a kind of conformance testing. Therefore, we aim at deriving tests from a formal

⋆ This work has been partly funded by the POTESTAT project of the national research programme on security (ACI Sécurité) and by the IMAG project Modeste.

⋆⋆ Email: {Vianney.Darmaillacq,Jean-Claude.Fernandez,Roland.Groz,Laurent.Mounier, Jean-Luc.Richier}@imag.fr specification of the security policy to check whether the implementation is correct. These tests could be used either after some initial deployment of a policy to check whether it can be and actually is well implemented, or typically on a more regular basis to see whether any update on the configurations might have breached security rules. Some sort of testing is actually performed by system administrators to check for known vulnerabilities: portscans and password crackers fall into that category. However, such tests are usually limited to just a single security mechanism. With existing techniques, it is difficult to address the issue of consistency of configurations on distributed devices. Although some work has been published on the analysis of the consistency of firewall rules (typically to minimize them or to detect conflicts), this is still limited to specific points of security policies. [11] is an application of the idea of generating conformance tests for a single firewall.

In our work, we address conformance with respect to a global specification of a security policy for a network of interconnected systems. Although at first view this might appear as just another instance of conformance testing of an implementation w.r.t. a given formal specification, there are a number of significant differences with the framework of such standards as IS9646 and its formal counterpart FMCT [8]. We investigated a few of these differences in [6]. Here are typical issues:

- Security policies are not just a specification of rules to be implemented in machines. They also address human behaviours (for users, administrators and managers), which are not necessarily reflected in electronic form. Therefore we concentrate on security rules which would actually be translated into testable machine implementations.
- Protocol conformance testing is done on a well defined protocol level (normally one at a time), with a given interface and associated PDU definitions. Whereas security policies are often implemented with a mixture of mechanisms at various levels of communication and O/S interfaces. We have to bridge the gap between specification at policy level and the actual level of PDUs and O/S events that are used by security control mechanisms and which can be accessed through points of control and observation.
- Contrary to protocol testing where the observable behaviour consists of PDUs that can be observed during the test, specific security actions might not yield clearly observable behaviours. Typically, access might be transparently granted, and the information on the control performed would be silently logged. It might be relevant, in testing security implementations to combine on-line observations by a tester with off-line analysis of system logs.
- Reliability of observations might also be an issue. We might have to consider that the information collected in a test might have been tampered with.
- Security rules use deontic modalities, such as authorization, denial and obligation. Testing such modalities is not straightforward. The fact that a category of subjects in the system might be authorized to do certain actions does not imply that any specific instance has been configured to act in this way. The whole test process has to be organized to avoid tests that could systematically be inconclusive.

In this paper, we do not address all issues, but we propose a method to derive tests from a policy expressed as a set of rules. We identify a language with a restricted form of rules that cover most of the rules that can be found in texts describing security policies, keeping in mind that we are focusing on rules that are translated into actual configurations and behaviours or network security devices. This restricted set of rules allows us to design a "tile-based" generation method. For each element of our language and each type of rule, we propose a pattern of test, which we call a tile. The simple form of rules makes it possible to compose a global test by simply combining the tiles associated to the elements.

This paper is organized as follows. In section 2, we present our approach, in particular the types of rules that we cover and their relation to proposed formal methods for the description of security policies. Our notation for rules is presented in section 3. In section 4, we define the test generation tiles and the algorithm to derive tests. The whole approach is illustrated in section 5 on a typical example taken from an e-mail security architecture. Finally, in section 6 we present our perspectives for development from this basis.

2 Approach

Our aim is to automatically generate test cases from network policy rules. We intend to adapt the approach used in protocol conformance testing. We first started by looking for a formal description of security policies that would make it possible to generate tests. In order to identify typical requirements and the corresponding tests, we worked from a case study.

2.1 A case study

We carried out during the summer of 2004 a case study to identify the security policies used in the IMAG network, which connects our laboratories. This study included the analysis of documents provided to the administrators and the users, and interviews. We collected information at different levels of management, from the laboratory to the access supplier. The analysis resulted in a rather broad and detailed description of a typical network security policy in a university environment. In this paper, we shall present a small subset of rules, centred on electronic mail. This set was selected to be rich enough to show the majority of the concepts to be studied: several levels of policies and inter-connected organizations, variety of the services and of the access methods.

In a network, security distinguishes inside and outside. The inside of the network is the set of machines under the responsibility of the organization. The outside consists of uncontrolled machines considered dangerous for security purposes. This can be refined by considering other zones, differing by degrees of administration, reliability and trust. The sub-zones of the internal zone correspond in general to architectural criteria, for example separations between physical sub-networks, whereas the outside sub-zones correspond to different levels of trust.

One distinguishes moreover among the internal hosts those which provide services accessible from the outside. Due to their visibility, these hosts are more often subject to attacks. Therefore state of the art in network administration recommends to define for these hosts a strongly controlled buffer zone, often called DMZ (demilitarized zone). Only the hosts in this zone can communicate with the outside world, and all the traffic between the DMZ and the rest of the internal network is controlled.

We give in figure 1 a sample set of rules for the electronic mail drawn from our case study. This sample illustrates: flows of information, separation in zones, possibility, obligation or prohibition of certain actions, at different levels of details. To simplify the problem, we suppose that certain global hypotheses are true and do not have to be clarified: correct routing, systems up to date w.r.t vulnerability patches ...

No	**Requirement**
1	Mail relays accepting messages from the exterior should be placed at the entry of the network, in the DMZ if possible.
2	There should be no user account on relays placed in the DMZ.
3	Mailbox servers containing user accounts should be in the private zone. There could be as many of these servers as necessary.
4	Relays in the DMZ are the only machines allowed to communicate with the exterior world using the SMTP protocol. Relay of inbound mails (to mailboxes) and of outbound mails (to exterior) is done using these relays.
5	At the entry of the site, a filtering default policy is applied, which forbid all traffic not explicitly authorized.
6	It is forbidden to hosts of the network to relay mails from an external host to an external host.
7	All messages coming from the exterior are redirected to mail relays placed in entry of the site (MX field of the DNS), probably in the DMZ.
8	It is forbidden to communicate with a host belonging to the blacklist updated daily provided by the MAPS (Mail Abuse Prevention System) partner.
9	Antivirus and spam filters shall be installed on hosts acting as relays or mailboxes.
10	A mail shall be checked by antivirus software before being opened.
11	All mails entering the network infected by a virus shall be disinfected.
12	All mail shall be modified if it contains a potentially dangerous attached file. The file is stored somewhere in the network for 15 days. Sender and recipients are notified and an address id provided to the recipient to get the file.
13	All mail analyzed as a possible spam shall be annotated.

Fig. 1. Security rules for e-mail

2.2 Description techniques for network security

Most of the rules in the case study (which actually covers much more than e-mail) express some constraints about the possible behaviour of the system. More specifically, they are of the form "*Mod P*", where *Mod* is a modality among obligation, permission or interdiction, and *P* is either a predicate on the system or a behaviour.

The deontic logic of von Wright [13] is a modal logic whose modal operator semantic is that of obligation. With only this operator authorization of a formula is defined as the negation of the obligation of the negation of the formula, while interdiction is defined as the negation of the obligation of the formula. However deontic logic raises a number of paradoxes that have hindered its wider use [10]. Nevertheless modalities are a key issues in formal models of security policies, in particular in formal description techniques such as PDL, Ponder and Or-BAC, which have been proposed to address network security policies.

PDL is a language created to model network management policies, including security requirements. It is based on the idea that a policy is a specification of the behaviour the network should have according to what happened in the network [9]. A rule states that an action is triggered by an event, provided a condition holds. PDL was used to monitor switches in a network, to guarantee quality of service [12].

Ponder is a language created to specify network security policies [5] and proposes a full choice of different modalities: conditional obligation, authorization, interdiction, delegation and refrain policies could be specified.

Or-BAC [1] is another access control model, loosely based on RBAC. While RBAC abstracts subjects into roles, Or-BAC abstracts moreover objects into views and actions into activities, and introduces the notion of organization. Or-BAC has been used to model a network security policy, and to generate firewall rules from it [4, 2].

All these FDTs include structuring and typing constructs, resolution of conflict mechanisms and various aspects which are important. In this paper, we go for a simpler course, since we want to derive tests from the rules. These rules could be extracted from description in the above formalisms. All we need is a description with just enough expressive power to represent the modalities used in PDL, Ponder or Or-BAC, at least as they can be tested through network events.

2.3 Approach for test generation

Compared to classical conformance testing for communication protocols, test generation in our case exhibits two major differences.

- The policy is not described by a comprehensive model such as a global LTS or state machine, but by a collection of rules. This is similar to deriving tests from requirements. Much work has been done in test generation from test purposes which are confronted to a formal model of the protocol. Here, we do not assume any formal model of the network, we derive tests from the rules. Our approach is rule based: it generates separate tests for each rule.
- The policy is described at a much higher level than the actual events that can be observed or controlled in the network; whereas the specification of a communication protocol would refer to PDUs or SDUs even though some of them might occur at non-observable interfaces. We need to establish a correspondence between the basic predicates appearing in a rule and sequences of events that can represent a test or instanciations of such predicates.

To each high-level predicate (such as *externRelay*(h) meaning that machine h can relay mails sent from outside the domain considered) we associate a test pattern which we call a *tile*. For instance, in the case of external relaying of mails, a tester would have to establish an SMTP connection to the machine from an external machine and try to send a mail. However, there are different types of predicates. Some may have to be tested dynamically through interaction with the system. Others might be checked without PDU exchanges, for instance if we have access to the configuration files of the system when the test is set up. In the case of *externRelay*(h), this could be checked in the configuration of the mail system on h. The choice of one method or another may depend on accessibility to the system, but on trust as well: typically, information on configurations might not be reliable.

In [7], we investigated a refinement approach to derive control at PDU level from security rules. In this paper, we will consider that the tiles are provided. A policy would be described by combining elementary predicates which would be well-known elements for security policies, so that the corresponding tiles would be readily available.

In this paper, we concentrate on the combination of tiles, based on the structure of the formula in the rule. We address formulas of a restricted form with just one modality, as this corresponds to the usual style of security policy rules; Ponder, PDL and OrBAC also propose a single modality on each rule. We first propose a formal description of the rules in the next section, then we describe the combination of tiles in the following one.

3 Rules Formalisation

We first give the syntax and semantics of a more general formalism and then we restrict it to a smaller subset used in this work to generate test cases.

Preliminaries. We recall some basic definitions that are used in this paper. A *labelled transition system* (LTS, for short) is a quadruplet (Q, A, T, q^0) where Q is a set of states, A a set of labels, T the transition relation ($T \subseteq Q \times A \times Q$) and q^0 the initial state ($q^0 \in Q$). We will use the following definitions and notations: $(p, a, q) \in T$ is noted $p \xrightarrow{a}_T q$ (or simply $p \xrightarrow{a} q$). An *execution sequence* ρ is a composition of transitions: $q^0 \xrightarrow{a_1} q_1 \xrightarrow{a_2} q_2 \cdots \xrightarrow{a_n} q_n$. We denote by σ^ρ (resp. α^ρ) the sequence of states (resp. observables actions) associated with ρ. The sequence of actions α^ρ is called a *trace*. We note by Σ_S, the set of finite execution sequences starting from the initial state q^0 of S. For any sequence λ of length n, λ_i or $\lambda(i)$ denotes the i-th element and $\lambda_{[i \cdots n]}$ denotes the suffix $\lambda_i \cdots \lambda_n$.

We also consider in the sequel Boolean Labelled Transition Systems in order to obtain a more compact representation of test cases. A Boolean Labelled Transition System (BLTS for short) is a tuple (X_b, Q, A, T, q^0) where X_b is a set of Boolean variables, Q is a set of states, A is a set of actions, q^0 is the initial state and $T \subseteq Q \times (Bexp \times A \times Bcmd) \times Q$ is the set of transitions, where:

- *Bexp* is a *guard*, i.e. a Boolean expression constructed using the following grammar `b ::= True | x | False | b ∧ b | ¬ b` (where x$\in X_b$);
- *Bcmd* is either an assignment `x := b` (where x$\in X_b$, b $\in$ *Bexp*) or the null command `skip`.

As usual, we note $p \xrightarrow{(b,a,c)} q$ for $(p,(b,a,c),q) \in T$. We can omit b (*resp.* c) when b is `True` (*resp.* c is `skip`).

The semantics of a BLTS is given by a LTS. We define a notion of configuration (p, γ), where p is a state of the BLTS and $\gamma : X_b \mapsto Bool$ a valuation, where $Bool = \{True, False\}$, is the set of Boolean values. Valuation γ are extended to $Bexp$ in the usual way (i.e., $\gamma : Bexp \mapsto Bool$). Given a BLTS $B = (X_b, Q, A, T, q^0)$ and the initial valuation γ_0, where $\gamma_0(x) = False$ for all $x \in X_b$, the underlying LTS $S_B = (Q_1, A, T_1, q_1^0)$ is defined as follows:

$Q_1 \subseteq Q \times Bool^{X_b}$,

$(p,\gamma) \xrightarrow{a} (p_1,\gamma_1)$ iff $(p,(b,a,c),q) \in T$ and $\gamma(b) = True$,

$$\gamma_1 = \begin{cases} \gamma[v/x] & \text{if } c \text{ is } \texttt{x:=e}, \gamma(\texttt{e}) = v \text{ and } \gamma(b) = true \\ \gamma & \text{if } c \text{ is } \texttt{skip} \end{cases}$$

$q_1^0 = (q^0, \gamma_0)$

3.1 Syntax of security rules

A security policy rule is expressed by a logical *formula* (φ), built upon *literals*. Each literal can be either a *condition literal* ($p_c \in P_c$), or an *event literal* ($p_e \in P_e$). A conjunction of condition literals is simply called a *condition* (C), whereas a conjunction of a single event literal and a condition is called a *(guarded) event* (E). Finally, we also use a modal operator $\mathcal{F}$, the dual one $\mathcal{G}$, and the usual Boolean connectors $\neg$ and $\Rightarrow$.

The abstract syntax of a formula is then given by the following grammar:

$$\begin{aligned} &\varphi ::= C \mid E \mid \neg\varphi \mid \varphi \Rightarrow \varphi \mid \mathcal{F}\varphi \mid \mathcal{G}\varphi \\ &C ::= p_c^1 \wedge \cdots \wedge p_c^n \quad \text{where } p_c^i \in P_c \\ &E ::= p_e[C] \qquad \text{where } p_e \in P_e \end{aligned}$$

3.2 Semantics

Formulas are interpreted over LTS. Intuitively, an LTS S satisfies a formula φ iff *all* its execution sequences ρ do, where condition literals are interpreted over *states*, event literals are interpreted over *labels* and the modal operator $\mathcal{F}\varphi$ means that *there exists* a suffix $\rho_{[i..|\rho|]}$ of ρ such that φ holds on $\rho_{[i..|\rho|]}$, where $|\rho|$ is the number of elements of ρ. We first introduce two interpretation functions for condition and event literals:

$f_c : P_c \to 2^Q$, associates to p_c the set of states on which p_c holds;

$f_e : P_e \to 2^A$, associates to p_e the set of labels on which p_e holds.

The satisfaction relation of a formula φ on an execution sequence ρ ($\rho \models \varphi$) is then (inductively) defined as follows:

- $\rho \models C$ for $C = p_c^1 \wedge \cdots \wedge p_c^n$ iff $\forall i.\ \sigma^\rho(1) \in f(p_c^i)$
- $\rho \models E$ for $E = p_e[C]$ iff $\alpha^\rho(1) \in g(p_e) \wedge \sigma^\rho(2) \models C$
- $\rho \models \neg\varphi$ iff $\rho \not\models \varphi$
- $\rho \models \varphi_1 \Rightarrow \varphi_2$ iff $((\rho \models \varphi_1) \Rightarrow (\rho \models \varphi_2))$
- $\rho \models \mathcal{F}\varphi$ iff $\exists i \in [1, |\rho|].\ \rho_{[i \cdots |\rho|]} \models \varphi$
- $\rho \models \mathcal{G}\varphi$ iff $\forall i \in [1, |\rho|].\ \rho_{[i \cdots |\rho|]} \models \varphi$

Finally, $S \models \varphi$ iff $\forall \rho \in \Sigma_S.\ \rho \models \varphi$.

3.3 Expression of security rules

The specification language defined above can be viewed, at first glance, as a rather classical linear temporal logic, with a single modality $\mathcal{F}$, and mixing state-based and event-based atomic predicates. However, our purpose is to use it here to express *security rules* to be satisfied by a network. In this particular context its semantics should be interpreted as follows:

- The network behaviour is expressed by the LTS S: each state of S represents the global state of the network at a given time (network configuration and topology, transiting PDUs, etc.), and each label of S represents an observable action performed at the network level (modification of the configuration/topology, PDU reception, PDU emission, etc.).
- A condition literal p_c expresses a (*static*) predicate on a network state, at the security policy level. For instance $\mathit{externRelay}(h1)$ holds on a state iff machine $h1$ is configured as an external mail relay in this state, or $\mathit{infected}(m1)$ holds on a state iff message $m1$ contains a virus.
- An event literal p_e expresses a (*dynamic*) predicate on a network transition, from a given state, at the security policy level. For instance $\mathit{enterNetwork}(m)$ holds iff the current transition corresponds to reception of mail m by the network, and $\mathit{chkVirus}(m)$ holds iff the current transition corresponds to a virus check on mail m.
- The $\mathcal{F}$ operator is used here to express an *obligation*, meaning that a given formula *should eventually* hold later, in a bounded future. For instance $\mathit{enterNetwork}(m) \Rightarrow \mathcal{F}\mathit{chkVirus}(m)$ means that whenever mail m enters the network then it should be checked.

As a matter of fact, it happens that all the formulas found in the case study could be expressed using only a restricted subset of this language. In particular formulas can be classified on three types, according to the following grammar:

$$\begin{aligned}
\varphi &::= \mathcal{G}\ \mathcal{C}\mathbf{-Rule} \mid \mathcal{G}\ \mathcal{F}\mathbf{-Rule} \mid \mathcal{G}\ \mathcal{G}\mathbf{-Rule} \\
\mathcal{C}\mathbf{-Rule} &::= C \Rightarrow C \mid E \Rightarrow C \\
\mathcal{F}\mathbf{-Rule} &::= E \Rightarrow \mathcal{F}E \\
\mathcal{G}\mathbf{-Rule} &::= C \Rightarrow \mathcal{G}\neg E \mid E \Rightarrow \mathcal{G}\neg E
\end{aligned}$$

A $\mathcal{C}$**−Rule** expresses a static conditional implication, an $\mathcal{F}$**−Rule** expresses a (triggered) obligation and a $\mathcal{G}$**−Rule** expresses that, when a given condition holds or when a given event happens, then a particular event is always *prohibited.*

4 Test Generation

In this section, we propose a "tile-based" approach to generate *abstract test cases* from a formula expressing a security rule. The test generation principle is the following: assuming that elementary test cases (i.e., *tiles*) t_i are provided for each (condition and event) literals appearing in a formula φ, the test case t associated to φ is obtained by combining test cases t_i with "test operators" (defined below), corresponding to the logical operators appearing in φ. This allows defining a structural correspondence between formulas and test cases.

4.1 Test cases and test execution

We can define the notion of test case as a BLTS extended with two special actions (to deal with *timers*), and three special states (called *verdicts*): action *timerset* means a timer initialization to a given value, and action *timeout* means the timer expiration; verdict states are "sink states", indicating the end of a successful (*pass*), unsuccessful (*fail*) or inconclusive (*inconc*) test execution. We denote by A^t the set $A \cup \{timeout, timerset\}$, and by V the set $\{pass, fail, inconc\}$. We denote by Σ_S^{pass}(*resp.* Σ_S^{fail}, Σ_S^{inconc}) the set of execution sequences, starting from the initial state and ending in the state *pass* (resp. *fail*, *inconc*). A *test case* t is then a BLTS $t = (X_p, Q, A^t, T, q^0, V)$, with $V \subseteq Q$.

A test case is supposed to be executed by a tester against a network whose behaviour can be modelled by an LTS $I = (Q^I, A^I, T^I)$ (we do not take care of the initial state of the network). Usually, in "black-box" conformance testing, this behaviour is observed/controlled by the tester only through a restricted interface. For the sake of simplicity we assume here that *any* output action (*resp.* input action) performed by the network can be observed (*resp.* controlled) by the tester. Thus, execution of a test t on an IUT I, noted $\text{Exec}(t, I)$, is simply expressed as a set of common execution sequences of S_t and I, defined by a composition operator $\otimes$: let $\rho_I = q_0^I \xrightarrow{a_1} q_1^I \xrightarrow{a_2} q_2^I \cdots \xrightarrow{a_n} q_n^I \cdots \in \Sigma_I$ and $\rho_{S_t} = q^{0,t} \xrightarrow{a_1} q_1^t \xrightarrow{a_2} q_2^t \cdots \xrightarrow{a_n} q_n^t \in \Sigma_{S_t}$, then

$\rho_{S_t} \otimes \rho_I = (q^{0,t}, q_0^I) \xrightarrow{a_1} (q_1^t, q_1^I) \cdots \xrightarrow{a_n} (q_n^t, q_n^I) \in \text{Exec}(t, I)$.

For $\rho \in \text{Exec}(t, I)$, we define the verdict function: $\text{VExec}(\rho) = pass$ (resp. *fail*, *inconc*) iff there is $\rho_{S_t} \in \Sigma_{S_t}^{\text{pass}}$ (resp $\Sigma_{S_t}^{\text{fail}}, \Sigma_{S_t}^{\text{inconc}}$) and $\rho_I \in \Sigma_I$ such that $\rho_{S_t} \otimes \rho_I = \rho$.

4.2 Test generation functions

Let φ be a formula, let p_e, p_c^i its literals, and t_{p_e}, $t_{p_c^i}$ their corresponding elementary test cases. Note that an elementary test case is reduced to a simple verdict state when it corresponds to a literal that can be immediately checked on the network behaviour (without requiring any interaction sequence with a tester). The test generation function $\text{gentest}(\varphi)$ is inductively defined on the syntax of the formula, where X_1 denotes either a condition C or an event E:

$$\begin{aligned}
\text{gentest}(X_1 \Rightarrow C_2) &= \text{gentest\_lX}(X_1) \rhd_{\text{lrC}} \text{gentest\_rC}(C_2)\\
\text{gentest}(E_1 \Rightarrow \mathcal{F}E_2) &= \text{gentest\_lX}(E_1) \rhd_{\text{lrF}} \text{gentest\_rF}(E_2)\\
\text{gentest}(X_1 \Rightarrow \mathcal{G}\neg E_2) &= \mathit{Inv}(\text{gentest\_lX}(X_1) \rhd_{\text{lrF}} \text{gentest\_rF}(E_2))\\
\text{gentest\_lX}(p_c^1 \wedge \cdots \wedge p_c^n) &= (((t_{p_c^1} \rhd_{\text{llX}} t_{p_c^2}) \rhd_{\text{llX}} \ldots) \rhd_{\text{llX}} t_{p_c^n})\\
\text{gentest\_lX}(p_e[C]) &= t_e \rhd_{\text{llX}} \text{gentest\_lX}(C)\\
\text{gentest\_rC}(p_c^1 \wedge \cdots \wedge p_c^n) &= (((t_{p_c^1} \rhd_{\text{rrC}} t_{p_c^2}) \rhd_{\text{rrC}} \ldots) \rhd_{\text{rrC}} t_{p_c^n})\\
\text{gentest\_rC}(p_e[C]) &= t_e \rhd_{\text{rrC}} \text{gentest\_rC}(C)\\
\text{gentest\_rF}(p_e[C]) &= t_e \rhd_{\text{rrF}} \text{gentest\_rC}(C)
\end{aligned}$$

Test operators $\rhd_{\text{rrF}}$, $\rhd_{\text{lrF}}$, $\rhd_{\text{rrC}}$, $\rhd_{\text{lrC}}$, $\rhd_{\text{llX}}$ and Inv are defined below.

4.3 Test operators

In the following we assume that $t_1 = (X_{b_1}, Q_1, A_1, T_1, q_1^0, \{\mathit{pass}_1, \mathit{fail}_1, \mathit{inconc}_1\})$ and $t_2 = (X_{b_2}, Q_2, A_2, T_2, q_2^0, \{\mathit{pass}_2, \mathit{fail}_2, \mathit{inconc}_2\})$ are two test cases. For each binary test operator $\rhd$ we define the test case $t = (X_b, Q, A, T, q^0, V)$ such that $t = t_1 \rhd t_2$ and $V = \{\mathit{pass}, \mathit{fail}, \mathit{inconc}\}$. For each operator, a graphical presentation is proposed on figure 2 and we give hereafter the formal definition of only three operators, the three others being given in annex.

Operator $\rhd_{\text{llX}}$ ($t = t_1 \rhd_{\text{llX}} t_2$). This operator is used to combine two test cases appearing on the left-hand side of an implication. Therefore, t *pass* iff t_1 *and* t_2 does, and t is *inconclusive* otherwise (the entire formula cannot be tested). More formally:

$$\begin{aligned}
X_b &= X_{b_1} \cup X_{b_2},\\
Q &= (Q_1 \setminus V_1) \cup (Q_2 \setminus V_2) \cup V,\\
A &= A_1 \cup A_2,\\
q^0 &= q_1^0,\\
T &= T_1 \setminus \{(p,a,q) \mid q \in V_1\} \cup T_2 \setminus \{(p,a,q) \mid q \in V_2\}\\
&\quad \cup \{(p,a,q_2^0) \mid (p,a,\mathit{pass}_1) \in T_1\} \;\cup\; \{(p,a,\mathit{pass}) \mid (p,a,\mathit{pass}_2) \in T_2\}\\
&\quad \cup \{(p,a,\mathit{inconc}) \mid (p,a,q) \in T_1 \wedge q \in \{\mathit{inconc}_1, \mathit{fail}_1\}\}\\
&\quad \cup \{(p,a,\mathit{inconc}) \mid (p,a,q) \in T_2 \wedge q \in \{\mathit{inconc}_2, \mathit{fail}_2\}\} \qquad (\alpha)
\end{aligned}$$

Operator $\rhd_{\text{lrC}}$ ($t = t_1 \rhd_{\text{lrC}} t_2$). This operator is used to combine a left-hand side (t_1) and a right-hand side (t_2) of an implication for a $\mathcal{C}-$**Rule**. Therefore t can be expressed by a sequential execution of t_1 and t_2, and it *pass* iff t_1 *and* t_2 does, it *fails* when t_1 pass and t_2 fails (implication), and it is *inconclusive* otherwise. Formal definition of test t is then simply obtained by replacing the line (α) of the previous definition (operator $\rhd_{\text{llX}}$) by the following one:
$\{(p,a,\mathit{inconc}) \mid (p,a,\mathit{inconc}_2) \in T_2\} \cup \{(p,a,\mathit{fail}) \mid (p,a,\mathit{fail}_2) \in T_2\}$

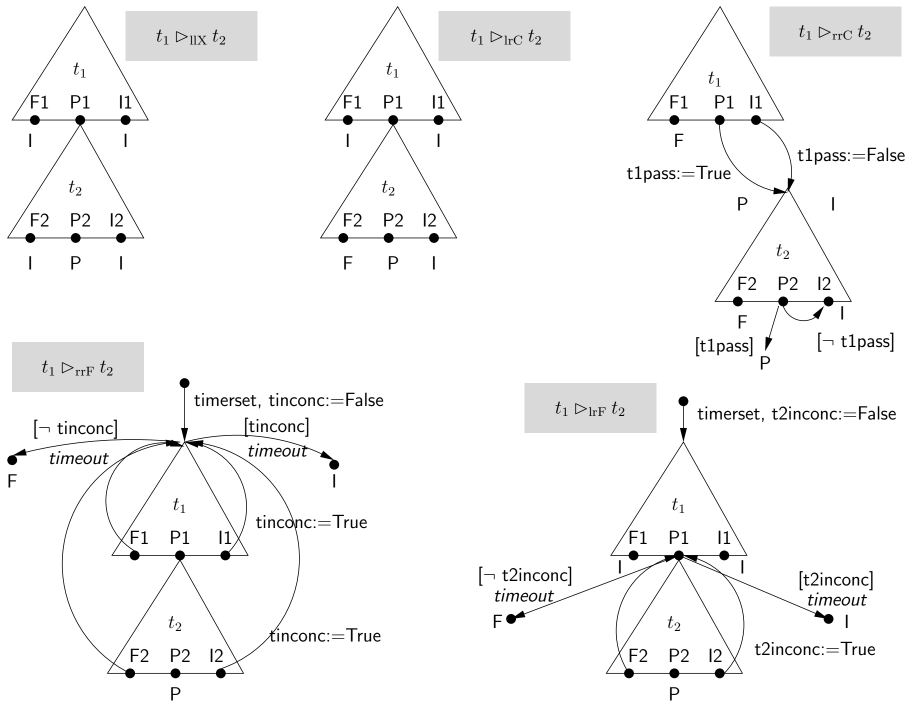

Fig. 2. Test operators

Operator $\rhd_{\mathbf{rrC}}$ ($t = t_1 \rhd_{\mathbf{rrC}} t_2$). This operator is used to combine two test cases appearing on the right-hand side of a an implication, for a $\mathcal{C}-$**Rule**. Therefore t *pass* when t_1 *and* t_2 does, t *fails* when t_1 *or* t_2 fails, and it is *inconclusive* otherwise. Thus, t can be obtained by executing t_1 first, followed by t_2 (when t_1 does not fail). A Boolean variable `t1pass` is used to store the verdict of t_1. The formal definition is given in annex.

Operator $\rhd_{\mathbf{lrF}}$ ($t = t_1 \rhd_{\mathbf{lrF}} t_2$). This operator is used to combine a left-hand side (t_1) and a right-hand side (t_2) of an implication for an $\mathcal{F}-$**Rule**. It is therefore similar to the $\rhd_{\mathrm{lrC}}$ operator, excepted that, due to the $\mathcal{F}$ temporal modality, t *pass* iff t_1 pass and t_2 pass *at some point later in the future* (remember that the right-hand side of an $\mathcal{F}-$**Rule** is necessarily an *event*). t is then obtained by executing t_1 first, and then repeatedly executing t_2 until either it passes or a given timeout is reached (to ensure that the test execution remains finite). A Boolean variable `t2inconc` is used to keep track of an occurrence of an inconclusive verdict of t_2 (during its repeated execution), hence leading to an inconclusive verdict of t. More formally:

$X_b = X_{b_1} \cup X_{b_2} \cup \{\texttt{t2inconc}\}$,
$Q \ = (Q_1 \setminus V_1) \cup (Q_2 \setminus V_2) \cup V \cup \{q_0\}$,
$A \ = A_1 \cup A_2$,

$$\begin{aligned} T \;=\;& T_1 \setminus \{(p,a,q) \mid q \in V_1\} \cup T_2 \setminus \{(p,a,q) \mid q \in V_2\} \\ &\cup \{(q_0, (\mathit{timerset}, \texttt{tinconc} := \texttt{False}), q_1^0)\} \\ &\cup \{(p,a,\mathit{inconc}) \mid (p,a,q) \in T_1 \wedge q \in \{\mathit{inconc}_1, \mathit{fail}_1\}\} \\ &\cup \{(p,a,q_2^0) \mid (p,a,\mathit{pass}_1) \in T_1\} \\ &\cup \{(p,a,q_2^0) \mid (p,a,\mathit{fail}_2) \in T_2\}\} \;\cup\; \{(p,a,\mathit{pass}) \mid (p,a,\mathit{pass}_2) \in T_2\} \\ &\cup \{(p,(a,\texttt{t2inconc} := \texttt{True}),q_2^0) \mid (p,a,\mathit{inconc}_2) \in T_2\} \\ &\cup \{(q_2^0, (\neg\texttt{t2inconc}, \mathit{timeout}), \mathit{fail})\} \cup \{(q_2^0, (\texttt{t2inconc}, \mathit{timeout}), \mathit{inconc})\} \end{aligned}$$

Operator $\rhd_{\mathbf{rrF}}$ ($t = t_1 \rhd_{\mathbf{lrF}} t_2$). This operator is used to combine two test cases appearing on the right-hand side of an implication for an $\mathcal{F}-$**Rule**, where t_1 tests the occurrence of an *event literal* p_e and t_2 a (static) *condition* C (possibly restricting p_e). Therefore, t *pass* iff both the expected event occurs at some point (t_1 pass) *and* condition C holds on the same time. t is then obtained by repeating t_1 followed by t_2 until both pass. Here again, a timeout ensures that execution of t always remains finite, and a Boolean variable `tinconc` is used to keep track of an inconclusive verdict of t_1 or t_2. The formal definition is given in annex.

Operator *Inv* ($t = \mathit{Inv}(t_1)$). This operator simply "reverts" the *pass* and *fail* verdicts produced by a test case. The formal definition is given in annex.

4.4 Soundness of the test generation function

It now remains to establish that an abstract test case produced by function gentest(φ) is always *sound*, i.e., it delivers a *fail* verdict when executed on a network behaviour I only if formula φ does not hold on I.

Two hypotheses are required in order to prove this soundness property:

H1. First, for any formula φ, we assume that each elementary test case t_i provided for the (event or condition) literals p_i appearing in φ is *strongly sound* in the following sense:
$\forall \rho \in \mathrm{Exec}(t_i, I) \cdot \mathrm{VExec}(\rho) = \mathit{Pass} \Rightarrow \rho \models p_i \wedge (\mathrm{VExec}(\rho) = \mathit{Fail} \Rightarrow \rho \not\models p_i)$.

H2. Second, we assume that the whole execution of a (provided or generated) test case t associated to a condition C is *stable* with respect to condition literals: the valuation of these literal does not change during the test execution. This simply means that the network configuration is supposed to remain stable when a condition is tested. Formally:
$\forall p_i \in P_c.\ \forall \rho \in \Sigma_I \cdot \rho_{S_t} \otimes \rho \in \mathrm{Exec}(t, I) \Rightarrow (\sigma^\rho \subseteq f_c(p_i) \vee \sigma^\rho \cap f_c(p_i) = \emptyset)$
where σ^ρ denotes here tacitly a set of states instead of a sequence.

We now formulate the soundness property:
Proposition: Let φ a formula, I an LTS and $t = \mathrm{gentest}(\varphi)$. Then:
$\rho \in \mathrm{Exec}(t, I) \wedge \mathrm{VExec}(\rho) = fail \Longrightarrow I \not\models \varphi$.

The proof of this proposition relies on the following lemma:
Lemma 1. Test cases generated by auxiliary function gentest_lX are *strongly sound*, and test cases generated by auxiliary functions gentest_rC and gentest_rF are *sound*.

Proof sketch of Lemma 1. Let t a test case generated by function gentest_lX. The proof that t is *strongly sound* is performed by recurrence on the number of elementary tests cases t_i appearing in t (assuming that each t_i itself is *strongly sound* according to hypothesis **H1**). A similar proof can be done for functions gentest_rC and gentest_rF.

Proof of Proposition. By structural induction on the formulas φ (we only detail here some representative induction steps).

- $\varphi = C_1 \Rightarrow C_2$. By definition of function gentest there exists test cases t_1 and t_2 such that $t_1 = \text{gentest\_lX}(C_1)$, $t_2 = \text{gentest\_rC}(C_2)$, and $t = t_1 \rhd_{\text{lrC}} t_2$. Let ρ be an execution sequence of $\text{Exec}(t, I)$ such that $\text{VExec}(\rho) = \mathit{fail}$. Then, by definition of operator $\rhd_{\text{lrC}}$, there exist ρ_1 and ρ_2 such that: $\rho = \rho_1.\rho_2$, $\rho_1 \in \text{Exec}(t_1, I)$, $\rho_2 \in \text{Exec}(t_2, I)$, $\text{VExec}(\rho_1) = \mathit{pass}_1$ and $\text{VExec}(\rho_2) = \mathit{fail}_2$.
  Therefore, by Lemma 1, $\rho_1 \models C_1$, hence $\sigma^\rho(1) \models C_1$ and similarly $\rho_2 \not\models C_2$ and $\sigma^\rho(|\rho_1|) \not\models C_2$. By hypothesis **H2** we obtain $\sigma^\rho(1) \not\models C2$ and then $\sigma^\rho(1) \not\models \varphi$.
- $\varphi = E_1 \Rightarrow \mathcal{F}E_2$. By definition of function gentest there exist test cases t_1 and t_2 such that $t_1 = \text{gentest\_lX}(E_1)$, $t_2 = \text{gentest\_rF}(E_2)$, and $t = t_1 \rhd_{\text{lrF}} t_2$. Let ρ be an execution sequence of $\text{Exec}(t, I)$ such that $\text{VExec}(\rho) = \mathit{fail}$. Then, by definition of operator $\rhd_{\text{lrF}}$, there exist ρ_1, ρ_2 such that: $\rho = q_0 \xrightarrow{a_0} q_0^1 \xrightarrow{\rho_1} \mathit{pass}_1 (\xrightarrow{\rho_2} \mathit{fail}_2)^* \xrightarrow{a_1} q_0^2 \xrightarrow{a_2} \mathit{fail}$, with $a_0 = (\mathit{timerset}, \texttt{t2inconc} := \texttt{False})$, $a_1 = \texttt{t2inconc} := \texttt{False}$, and $a_2 = (\mathit{timeout}, \neg\texttt{t2inconc})$. Moreover, $\rho_1 \in \text{Exec}(t_1, I)$ and $\rho_2 \in \text{Exec}(t_2, I)$. Therefore, by Lemma 1, $\sigma^\rho(1) \models E_1$ and, for all i in $[|\rho_1|, |\rho|]$, $\sigma^\rho(i) \not\models E_2$. We can then conclude that $\sigma^\rho(1) \not\models (E_1 \Rightarrow \mathcal{F}E_2)$.

5 Case study application

This section shows how the approach presented above can be applied to generate concrete tests for some examples from the case study of section 2.1.

5.1 $\mathcal{C}-$Rule

Consider the requirement "*External relays shall be in the DMZ*", which can be modelled by the $\mathcal{C}-$**Rule**:

$$\mathit{externRelay}(h) \Rightarrow \mathit{inDMZ}(h)$$

The goal of the test is to verify that each external relay is in the DMZ. As noted in section 4.2, an elementary test case is reduced to a simple verdict state when it corresponds to a literal that can be checked without requiring an interaction sequence. Such a case arises when the value can be checked by an analysis of the configuration of devices in the network and/or administrators' databases. For example, if the value of $\mathit{externRelay}(h)$ is true, that means that

h is defined as an external relay in the administrators' database and/or by the configuration of the network. This is known and trusted, not to be tested.

On the other side, the value is unsure if one has no knowledge about the fact that h is an external relay from the analysis of configurations, or if these data are untrusted. In this case the behaviour of the network should be tested to decide whether h acts as an external relay.

The following table shows the different formulas that may be built depending on which literals can be immediately asserted:

	externRelay(h)=true	... = false	... unsure
inDMZ(h)=true	t_{pass}	t_{inconc}	$t_{externRelay(h)} \rhd_{\mathrm{lrC}} t_{pass}$
inDMZ(h)=false	t_{fail}	t_{inconc}	$t_{externRelay(h)} \rhd_{\mathrm{lrC}} t_{fail}$
inDMZ(h) unsure	$t_{pass} \rhd_{\mathrm{lrC}} t_{inDMZ(h)}$	t_{inconc}	$t_{externRelay(h)} \rhd_{\mathrm{lrC}} t_{inDMZ(h)}$

If both values of *externRelay*(h) and *inDMZ*(h) are known and trusted, there is nothing to test. Also, no test is needed if *externRelay*(h) is *false*, as we cannot put the system in the desired state, and the verdict is *inconc*. If the value of *inDMZ*(h) (resp. *externRelay*(h)) is unsure, then it should be tested whether h behave like a host in the DMZ (resp. an external relay). These tests are then composed as described in section 4 into the formula $t_{externRelay(h)} \rhd_{\mathrm{lrC}} t_{inDMZ(h)}$, as illustrated in figure 3.

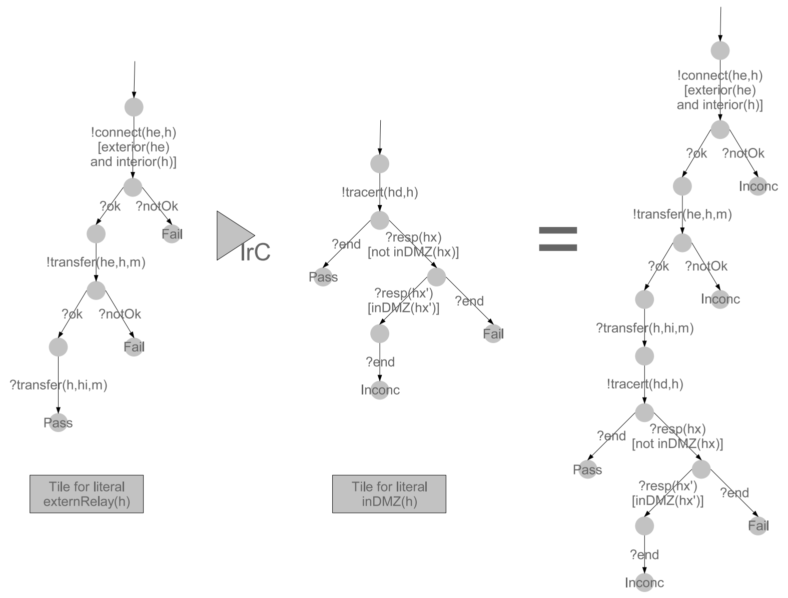

Fig. 3. Composition for example of $\mathcal{C}-$**Rule**

5.2 $\mathcal{F}-$**Rule**

Consider the requirement "*If an electronic mail is infected by a virus, the virus shall be deleted from the mail*". It can be modelled by the $\mathcal{F}-$**Rule** rule:

$$\begin{aligned}&\mathit{enterNetwork}(m)[\mathit{infected}(m)] \Rightarrow \\ &\ \mathcal{F}\ \mathit{transfer}(h_1, h_2, m)[\mathit{interior}(h_2) \wedge \neg \mathit{infected}(m)]\end{aligned}$$

The goal of this test is to verify that if a mail infected by a virus is sent to a user in the network, eventually one of the hosts the mail is passing through will suppress the virus, before a certain time elapses.

As always in our approach, a choice is made concerning which predicates are sure or unsure. The formula $t = (t_{enterNetwork(m)} \rhd_{\mathrm{llX}} t_{pass}) \rhd_{\mathrm{lrF}} (t_{transfer(h_1,h_2,m)} \rhd_{\mathrm{rrF}} (t_{pass} \rhd_{\mathrm{rrF}} t_{\neg infected(m)}))$ corresponds to the case when we build a test tile with a parameter m made of an infected message. This is the case because we choose to actively test the conformity of this particular rule against infected messages. One could also use a "passive" mode, checking the *infected*(m) literal in the left part of the formula on incoming messages.

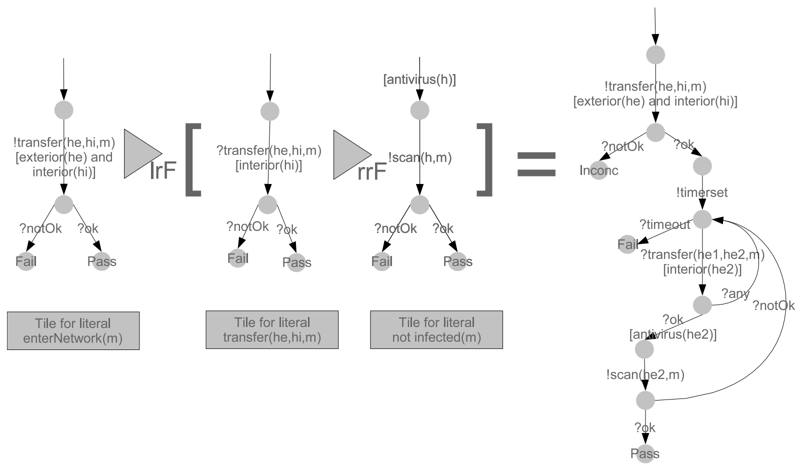

Fig. 4. Composition for example of $\mathcal{F}-$**Rule**

On the other hand the literal $\neg$*infected*(m) in the right part shall be tested. The event predicate *enterNetwork*(m) and the static predicate $\neg$*infected*(m) are tested by the tiles shown in figure 4. Using these tiles, the formula $t_{enterNetwork(m)} \rhd_{\mathrm{lrF}} (t_{transfer(m)} \rhd_{\mathrm{rrF}} t_{\neg infected(m)})$ gives the test on figure 4. The `tinconc` and `t2inconc` variables have been suppressed since they cannot be true, and also the corresponding transitions.

6 Perspectives

In this paper we have proposed a "tile-based" approach to derive test cases from network security rules expressed using a restricted set of logical operators. Complete test cases (dedicated to a whole formula) are obtained by combinations of more elementary ones (the tiles), following a syntax driven approach (a test combinator is associated to each logical operator of the formula). Elementary test cases, allowing to test basic events or predicates appearing in the security policy, have to be provided by the user (the way of testing such predicate depends on the network architecture and protocols involved). Our test generation method is based on the fact that security policies are most of the time expressed by rules which can be captured by a restricted logic as the one we described in section 3. At this point this work should be viewed essentially as a first step towards a formal approach to (automatically) test the compliance of a network with a given security policy. Therefore it should be extended into several directions.

First of all, the test cases produced are still very *abstract*. Turning them into executable test cases needs to take into consideration a concrete test architecture. Assuming that each elementary test case complies with this architecture, it would remain to ensure that it is also the case for the complete test case (or alternatively to take this architecture into account during the combination process). Moreover, these abstract test cases also need to be *instantiated* with concrete data (e.g. by selecting particular machines of the network). Suitable selection strategies should therefore be investigated (for instance a test could focus on the more recent changes in a network configuration, as in regression testing).

Furthermore, the proposed generation technique itself could be improved. In particular, the test case currently produced to test a condition (i.e., a disjunction of static predicates) consists in executing each corresponding elementary test case in sequence (according to the definition of our test combinators). An alternative way would have been to consider the *parallel* execution of such test cases (when it is compatible with the test architecture).

Another improvement could be to extend the formalism we considered to specify the security rules. This initial choice was motivated by our case study, and it was sufficient to demonstrate the effectiveness of the approach. However, it is clear that this formalism may be not sufficient to deal with arbitrary security rules, and that more specific operator/modalities need to be considered. One can think for instance of a triggered obligation bounded by an event (and not by an arbitrary timeout), or of some of the general operators proposed in the NOMAD logic [3]. Further work remains to be done in order to check which of these operators could be supported by our tile-based approach.

Finally, we also intend to evaluate this work on other case studies, and to prototype it on a real network.

References

1. A. Abou El Kalam, R. El Baida, P. Balbiani, S. Benferhat, F. Cuppens, Y. Deswarte, A. Miège, C. Saurel, and G. Trouessin. Organization Based Access

Control. In *IEEE 4th International Workshop on Policies for Distributed Systems and Networks*, 2003.
2. S. Benferhat, R. E. Baida, and F. Cuppens. A Stratification-Based Approach for Handling Conflicts in Access Control. In *8th ACM Symposium on Access Control Models and Technologies*, 2003.
3. F. Cuppens, N. Cuppens-Boulahia, and T. Sans. Nomad: A security model with non atomic actions and deadlines. In *18th IEEE Computer Security Foundations Workshop, (CSFW-18 2005)*, pages 186–196, Aix-en-Provence, France, 2005.
4. F. Cuppens, N. Cuppens-Boulahia, T. Sans, and A. Miège. A formal approach to specify and deploy a network security policy. In *Second Workshop on Formal Aspects in Security and Trust (FAST)*, 2004.
5. N. Damianou, N. Dulay, E. Lupu, and M. Sloman. The Ponder Policy Specification Language. In *International Workshop on Policies for Distributed Systems and Networks*, 2001.
6. V. Darmaillacq, J.-C. Fernandez, R. Groz, L. Mounier, and J.-L. Richier. Éléments de modélisation pour le test de politiques de sécurité. In *Colloque sur les RIsques et la Sécurité d'Internet et des Systèmes, CRiSIS*, Bourges, France, 2005.
7. V. Darmaillacq and N. Stouls. Développement formel d'un moniteur détectant les violations de politiques de sécurité de réseaux. In *AFADL2006 - Approches Formelles dans l'Assistance au Développement de Logiciels*, March 2006. To appear.
8. ITU. Framework on formal methods in conformance testing. ITU-T Recommendation Z.500, ITU, 1997.
9. J. Lobo, R. Bhatia, and S. Naqvi. A Policy Description Language. In *AAAI'99*, 1999.
10. J.-C. Meyer, F. Dignum, and R. Wieringa. The Paradoxes of Deontic Logic Revisited: A Computer Science Perspective. Technical Report UU-CS-1994-38, Utrecht University, 1994.
11. D. Senn, D. Basin, and G. Caronni. Firewall Conformance Testing. In *TestCom 2005, 17th IFIP TC6/WG6.1 International Conference on Testing of Communicating Systems, Montréal, LNCS 3502*, June 2005.
12. A. Virmani, J. Lobo, and M. Kohli. Netmon: Network Management for the SARAS Softswitch. In *IEEE/IFIP Network Operations and Management Symposium*, 2000.
13. G. H. von Wright. Deontic Logic. *Mind*, 60:1–15, 1951.

7 Annex

Formal definition of $\rhd_{\mathrm{rrC}}$

$$\begin{array}{lcl}
X_b & = & X_{b_1} \cup X_{b_2} \cup \{\texttt{t1pass}\}, \\
Q & = & (Q_1 \setminus V_1) \cup (Q_2 \setminus V_2) \cup V, \\
A & = & A_1 \cup A_2, \\
q^0 & = & q_1^0, \\
T & = & T_1 \setminus \{(p,a,q) \mid q \in V_1\} \cup T_2 \setminus \{(p,a,q) \mid q \in V_2\} \\
 & & \cup \{(p,a,\mathit{fail}) \mid (p,a,\mathit{fail}_1) \in T_1\} \ \cup \ \{(p,a,\mathit{fail}) \mid (p,a,\mathit{fail}_2) \in T_2\} \\
 & & \cup \{(p,(a,\texttt{t1pass} := \texttt{False}),q_2^0) \mid (p,a,\mathit{inconc}_1) \in T_1\} \\
 & & \cup \{(p,(a,\texttt{t1pass} := \texttt{True}),q_2^0) \mid (p,a,\mathit{pass}_1) \in T_1\} \\
 & & \cup \{(p,a,\mathit{inconc}) \mid (p,a,\mathit{inconc}_2) \in T_2\} \\
 & & \cup \{(p,(\texttt{t1pass},a),\mathit{pass}) \mid (p,a,\mathit{pass}_2) \in T_2\} \\
 & & \cup \{(p,(\neg\texttt{t1pass},a),\mathit{inconc}) \mid (p,a,\mathit{pass}_2) \in T_2\}
\end{array}$$

Formal definition of $\rhd_{\mathrm{rrF}}$

$$\begin{array}{lcl}
X_b & = & X_{b_1} \cup X_{b_2}, \\
Q & = & (Q_1 \setminus V_1) \cup (Q_2 \setminus V_2) \cup V \cup \{q_0\}, \\
A & = & A_1 \cup A_2, \\
T & = & T_1 \setminus \{(p,a,q) \mid q \in V_1\} \cup T_2 \setminus \{(p,a,q) \mid q \in V_2\} \\
\end{array}$$

$$\begin{array}{l}
\cup \{(q_0, (\mathit{timerset}, \texttt{tinconc} := \texttt{False}), q_1^0)\} \\
\cup \{(p,a,q_2^0) \mid (p,a,\mathit{pass}_1) \in T_1\} \ \cup \ \{(p,a,q_1^0) \mid (p,a,\mathit{fail}_1) \in T_1\} \\
\cup \{(p,(a,\texttt{tinconc} := \texttt{True})), q_1^0) \mid (p,a,\mathit{inconc}_1) \in T_1\} \\
\cup \{(p,(a,\texttt{tinconc} := \texttt{True})), q_1^0) \mid (p,a,\mathit{inconc}_2) \in T_2\}\} \\
\cup \{(p,a,q_1^0) \mid (p,a,\mathit{fail}_2) \in T_2\}\} \ \cup \ \{(p,a,\mathit{pass}) \mid (p,a,\mathit{pass}_2) \in T_2\} \\
\cup \{(q_1^0, (\neg\texttt{tinconc}, \mathit{timeout}), \mathit{fail})\} \ \cup \ \{(q_1^0, (\texttt{tinconc}, \mathit{timeout}), \mathit{inconc})\}
\end{array}$$

Formal definition of Inv

$$\begin{array}{ll}
T & = \{(p,a,\mathit{inconc}) \mid (p,a,\mathit{inconc}_1) \in T_1\} \\
 & \ \cup \{(p,a,\mathit{pass}) \mid (p,a,\mathit{fail}) \in T_1\} \cup \{(p,a,\mathit{fail}) \mid (p,a,\mathit{pass}) \in T_1\}
\end{array}$$