Security Policy Enforcement Through Refinement Process

Nicolas Stouls⋆ and Marie-Laure Potet

Laboratoire Logiciels Systèmes Réseaux - LSR-IMAG - Grenoble, France
{Nicolas.Stouls, Marie-Laure.Potet}@imag.fr

Abstract. In the area of networks, a common method to enforce a security policy expressed in a high-level language is based on an ad-hoc and manual rewriting process [24]. We argue that it is possible to build a formal link between concrete and abstract terms, which can be dynamically computed from the environment data. In order to progressively introduce configuration data and then simplify the proof obligations, we use the B refinement process. We present a case study modeling a network monitor. This program, described by refinement following the layers of the TCP/IP suite protocol, has to warn for all observed events which do not respect the security policy. To design this model, we use the event-B method because it is suitable for modeling network concepts.
This work has been done within the framework of the POTESTAT[1] project [9], based on the research of network testing methods from a high-level security policy.
Key words: Security policy enforcement, refinement, TCP/IP layers.

1 Introduction

The separation between *policies* and *mechanisms* is considered as a main specification principle in security. The policy describes the authorized actions while the mechanism is the method to implement the policy [24, 17]. Those two concepts do not have the same abstraction level. The classical process to enforce a policy consists of manually rewriting the policy in the same terms as the mechanism, with ad-hoc methods. We argue that a policy can be formally enforced in a mechanism by gradually building, through a refinement process, a link between abstract and concrete terms. We propose to design a specification with the same abstraction level as the policy and to refine it to obtain the concrete mechanism. In the case of critical software, using an abstract specification is, for example, required for test and audit processes or for certification according to the Common Criteria [7].

To illustrate our approach, we describe a network security software which has to enforce an abstract security policy in a TCP/IP network. Modern TCP/IP

⋆ Work supported by CNRS and ST-Microelectronics by the way of a doctoral grant.

[1] Security policies: test directed analysis of open networks systems.
http://www-lsr.imag.fr/POTESTAT/

networks are heterogeneous and distributed, and their management becomes more and more complex. Thus, the use of an abstract security policy can give a global and comprehensive view of a network security [22]. We choose to focus more specifically on an access control policy because it is the main concept in network security [10, 20].

We aim at designing a monitor, which warns if an action, forbidden by the policy, is observed on the network. In order to achieve that, we use the event-B method [1] for modeling network concepts.

The next section is an overview of the event-B method. Section 3 introduces networks and their security policy concepts. Then, Section 4 presents our approach, Section 5 describes our method based on the refinement process. Section 6 is a presentation of the case study. Finally, we conclude by comparing this work to related ones and by giving some prospects.

2 Event-B

The B method [2] is a formal development method as well as a specification language. B components can be refined and implemented. The correctness of models and refinements can be validated by proof obligations.

Event-B [1] is an extension of the B language where models are described by events instead of operations. The most abstract component is called *system*. Each event is composed by a guard G and an action T such that if G is enabled, then T can be executed. If several guards are enabled at the same time then the triggered event is chosen in a nondeterministic way.

Through the refinement process, data representation can be changed. The gluing invariant describes the relationship between abstract and concrete variables. If an event e_A is refined by an event e_R, then the refinement guard has to imply the abstract one. Moreover, some events can be introduced during the refinement process (refining the skip event), according to the same principles as the *stuttering* in TLA [15]. Due to the guard strengthening through refinement process, we have to prove that there is always at least one enabled event (no dead-lock) and that new events do not introduce live-locks.

To conclude, Table 1 defines the set notations which are used thereafter and Table 2 summarizes generalised substitutions.

Operator		Meaning
$A \leftrightarrow B$	$\hat{=}$	$\{R \mid R \subseteq A \times B\}$
$\mathsf{dom}(R)$	$\hat{=}$	$\{a \mid \exists b \cdot ((a,b) \in R)\}$
$\mathsf{ran}(R)$	$\hat{=}$	$\{b \mid \exists a \cdot ((a,b) \in R)\}$
$R[A]$	$\hat{=}$	$\{b \mid \exists a \cdot (a \in A \ \wedge \ (a,b) \in R)\}$
R^{-1}	$\hat{=}$	$\{(b,a) \mid (a,b) \in R\}$
$R_1 \ ; \ R_2$	$\hat{=}$	$\{(a,c) \mid \exists b \cdot ((a,b) \in R_1 \ \wedge \ (b,c) \in R_2)\}$
$R_1 \mid\mid R_2$	$\hat{=}$	$\{((a,b),(c,d)) \mid (a,c) \in R_1 \ \wedge \ (b,d) \in R_2\}$
$R \rhd B$	$\hat{=}$	$\{(a,b) \mid (a,b) \in R \ \wedge \ b \in B\}$
$A \leftrightarrow B$	$\hat{=}$	$\{F \mid F \in A \leftrightarrow B \ \wedge \ \forall (b_1,b_2) \cdot ((a,b_1) \in F \ \wedge \ (a,b_2) \in F \Rightarrow b_1 = b_2)\}$

Table 1. Used sets operators

Substitution	Syntactical notation	Mathematical notation
Do nothing	skip	skip
Assignment	$x := E$	$x := E$
Unbounded choice	ANY z WHERE P THEN T END	$@z \cdot (P \Rightarrow T)$
Condition	IF P THEN T_1 ELSE T_2 END	$P \Longrightarrow T_1 \parallel \neg P \Longrightarrow T_2$

Table 2. Used primitives substitutions

3 Introduction to Networks and their Security Policies

3.1 The TCP/IP Protocol Suite

Computer networks use a standard connection model, called OSI (Open Systems Interconnection) [13], composed of seven layers. The TCP/IP protocol suite implements this model but is described with only four layers: application, transport, network and link. Each of these layers plays a particular role:

- The *Application* layer is the interface between the applications and the network (client-side protocol).
- The *Transport* layer manages the host-to-host communications, but not the route between them (peer-to-peer networks).
- The *Network* layer manages the route between networks by selecting the network interface to use and the first router.
- The *Link* layer performs the signal translation (analogic/numeric) and synchronizes the data transmission. This layer is most often provided by the hardware. Therefore, it is not considered in the following sections.

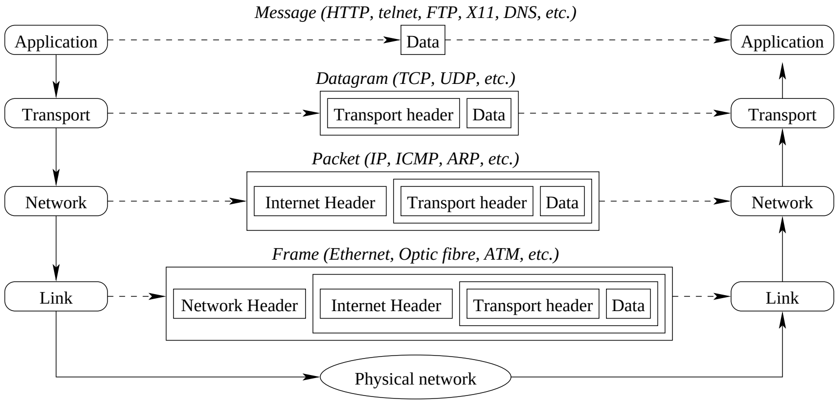


Fig. 1. Layers of the TCP/IP suite with some examples of protocols.

Communications using TCP/IP protocol are composed of protocols for each layer in the suite. For example, a TCP datagram (Transport layer) is contained in the data field of an IP packet (Network layer). Figure 1 shows an example of communication using TCP/IP.

3.2 Network Security Policies

In the area of networks, security is mainly expressed in terms of access rights. An access control policy is defined on a set of actions by a set of rules. These rules determine, for each action, whether the action is authorized or not. Among the various types of access control policies [11, 16], *open policies* and *closed policies* can be distinguished. An open policy (Fig. 2.A) expresses all *forbidden actions* (called *negative authorizations*): a not explicitly denied access is allowed. In a closed policy (Fig. 2.B), all *authorized actions* have to be fully specified (called *positive authorizations*). Finally, some policies are expressed with both positive and negative rules (Fig. 2.C). In this case, some actions can be *conflicting* or *undefined*.

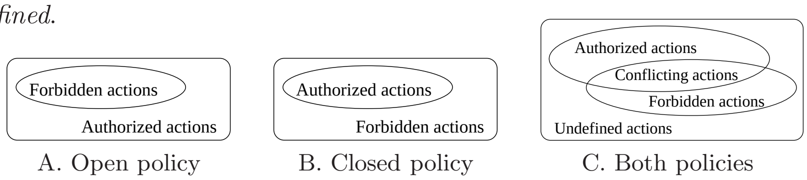


Fig. 2. Example of open, closed and both policies.

In the following, readers should distinguish *network events*, which are the elementary communication steps of the network, and B *events*, which are the description of actions in the B method.

In the proposed approach, a closed policy defined by a single set (*SP*) of authorized actions is used. However, each of these abstract actions can be associated to one or more concrete network events and conversely.

Definition 1 (Types of Events). *An event is* ***correct*** *with regard to a security policy if it corresponds only to authorized actions of the policy (Fig. 3.A). If the event is associated only to forbidden actions then this event* ***violates*** *the policy (Fig. 3.B). If an event is linked to some authorized actions and to some forbidden ones, then this event is in* ***conflict*** *with the policy (Fig. 3.C).*

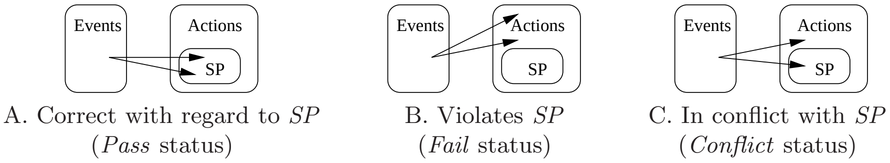


Fig. 3. Events correct with regard to *SP*, in violation of *SP* or in conflict with *SP*.

Several conformity relations [11, 19] can be used when conflicting events can occur. The approach proposed in this article is the following:

Definition 2 (Network Conformity). *A network conforms to a security policy if each event of this network is correct with respect to the policy.*

Finally, if a security policy is relevant only to a part of the network, then all events that are not associated to any action of the policy are *unspecified*.

4 Policy Security through TCP/IP Levels

4.1 Traceability from Policy to Implementation

The proposed approach aims to express a security policy at an abstract level (on actions) and to preserve it through the refinement process until its implementation (on network events). However, each refinement level can only access information from the protocol header of the corresponding TCP/IP layer (Fig. 1) and has to implement the same security policy as the specification.

The model used to illustrate this approach is a monitor. A monitor has to detect at least each network event which violates SP or which is in conflict with SP. In the ideal case, no event correct w.r.t. SP is detected. The monitor has then to guarantee, at each refinement level, the next two properties:

Property 1 (Monitor Correctness)
Each event that is not detected is correct with respect to the security policy.

Property 2 (Monitor Completeness)
Each event that is detected violates or is in conflict with the security policy.

In the model, the network events representation gradually changes at each refinement (from actions to concrete events). To implement these properties, the link between the different representations has then to be modeled, in a systematic way, at each refinement level. So, the end user (the administrator of a network) can choose the abstraction level of his policy (by using or not the more abstract levels of the model) and each event is traced through the refinement, as needed for some certification process such as the one of the Common Criteria [7].

Each refinement level represents how the communication is seen between two elements of the network. At level 0, events correspond to the access by a user to a service. They are considered as actions of the security policy. At level 1, events are messages between daemons (*Application layer*). At level 2, events are requests between hosts and are attached to particular ports (*Transport layer*). At level 3, each event is a connection between interfaces and is attached to particular ports (*Network layer*). Table 3 summarizes these different representations.

Level of specification	Network concepts	TCP/IP layer
0 (Policy level, actions)	Users, services	
1	Daemons, Terminal servers	Application
2	Hosts, ports	Transport
3 (implementation)	Interfaces, ports	Network

Daemon: software server providing some services.
Terminal server: particular daemon providing some logging services.
Host: machine of the network.
Ports: channels associated, on each host, to zero or one daemon.
Interface: network interface (e.g. a network card).

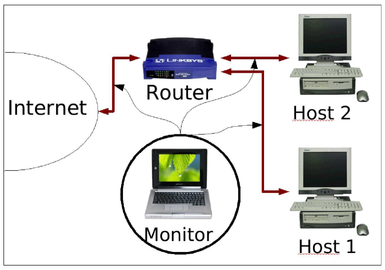
Table 3. Networks concepts by refinement level

These network concepts can be extracted from information contained in configuration files. For example the list of registered Linux users can be found in the `/etc/passwd` file and the list of daemons hosted on each machine can be found in `/etc/init.d/`. This information can then be used to associate each network event to an action of the policy.

Finally, an observer is introduced in the model to give the internal status (*Pass*, *Fail* or *Conflict* - Fig. 3) associated to each observed network event. As the event representation changes through the refinement process, the parameters of the observer are described in global variables.

4.2 Example

To illustrate the notions of conflict and failure, here is a short example (Fig. 4) of a monitor that receives a copy of each message from the network and that is parametrized by a security policy and the network configuration.



Network installation

$SP = \{(James, Intranet), (James, Mail), (Alice, Mail)\}$

Security policy

$Host_1$ is used by *James* or *Alice*
Intranet is hosted on the port 80 from $Host_2$
Mail is hosted on the port 993 from $Host_2$.

Network configuration information

Fig. 4. Example of the monitor installation

A message coming from $Host_1$ and going to $Host_2$ at port 80, is necessarily sent by *James* or *Alice*, because they are the only referenced $Host_1$ users. Moreover, due to the accessed port of $Host_2$, and according to the configuration information, the service can only be the *Intranet*. However, the security policy *SP* only allows *James* to access to the *Intranet*. Thus, the observed message is in *Conflict*.

Now, if the message comes from $Host_1$ and goes to port 993 of $Host_2$, then the accessed service is *Mail* and all the users of $Host_1$ are authorized to access it. Therefore, this message is correct w.r.t. the security policy (*Pass* case).

Finally, if a message is exchanged with a host ($Host_3$) which is not in the described part of the network, then no user or action is associated to it by the network configuration: the event is ignored.

5 Description of Refinement Levels

As previously said, each refinement level represents a different layer of the TCP/IP protocol suite (Table 3). In the first subsection, we define the data domain attached to each refinement level. Next, we present a systematic approach to model the links between refinements, based on configuration files. Then, we introduce journals in order to establish the monitor correctness and completeness properties (Properties 1 and 2). Finally, we describe the observer allowing to trace the status of each event through the refinement process.

5.1 Events Representation

Table 4 gives the representation of the events for each refinement level, according to the network concepts presented in Table 3. Moreover, the incoming ports are modeled while the outgoing ports are not so (Levels 2 and 3, Table 4). Outgoing ports are useless as long as the history of connections is not taken into account. Indeed, outgoing ports are dynamically and randomly chosen and cannot be used to identify a daemon or a user, contrary to the incoming ports, that are statically reserved for each service (managed by the IANA[2]).

Level of specification	Network events representation
0 *(Policy level)*	$Net_0 = USERS \times SERVICES$
1	$Net_1 = DAEMONS \times DAEMONS$
2	$Net_2 = HOSTS \times (HOSTS \times PORTS)$
3 *(Implementation)*	$Net_3 = INTERFACES \times (INTERFACES \times PORTS)$

Table 4. Network events representation for each refinement level

The policy is enforced only on the known part of the network. The constant $KnownNet_i$ represents the known network subset at each level i. Each set is divided into a known and an unknown part, as follows:

Known Network Definition:
$Users \subset USERS \wedge Services \subset SERVICES \wedge Daemons \subset DAEMONS \wedge Hosts \subset HOSTS$
$\wedge\ TerminalServers \subseteq Daemons \wedge Ports \subset PORTS \wedge Interfaces \subset INTERFACES$

$$\begin{aligned}
\wedge\ KnownNet_0 &= Users \times Services \\
\wedge\ KnownNet_1 &= TerminalServers \times Daemons \\
\wedge\ KnownNet_2 &= Hosts \times (Hosts \times Ports) \\
\wedge\ KnownNet_3 &= Interfaces \times (Interfaces \times Ports)
\end{aligned}$$

These abstract sets correspond to concrete data extracted from configuration files. Finally, the security policy is described by the constant set SP of all authorized actions of the known network.

Security Policy Definition:

$$SP \subseteq KnownNet_0$$

5.2 Representation Relation

The event representation changes through the refinement process. The relation $Represents_i$ defines the representation link between the i^{th} and $(i-1)^{th}$ refinement levels. For example, $Represents_1$ associates each terminal server to a set of users and each daemon to a set of services. Note that $Represents_i$ is a relation and not a function because a concrete event is not always associated to a single abstract event (as seen in the example from Section 4.2). It is neither a function from $KnownNet_{i-1}$ to $KnownNet_i$ because an action can be associated to several concrete events. These relations can be composed to define $Represents_{i \rightsquigarrow 0}$ between the ith level and the policy level.

[2] IANA = Internet Assigned Numbers Authority

Representation Relation axioms:

$$Represents_i \in KnownNet_i \leftrightarrow KnownNet_{i-1} \qquad (with\ i \in 1..3)$$
$$\wedge\ Represents_{i \rightsquigarrow 0} \in KnownNet_i \leftrightarrow KnownNet_0 \qquad (with\ i \in 1..3)$$
$$\wedge\ Represents_{i \rightsquigarrow 0} = (Represents_i; Represents_{i-1}; \ldots; Represents_1) \qquad (with\ i \in 1..3)$$

In order to simplify some further invariants, we define $Represents_0$ as the identity ($\mathsf{id}(Net_0)$). Finally, each element mentioned in the configuration has to be associated with at least one action and one network event:

The Described Sub-Network is Known as a Whole:

$$\mathsf{dom}(Represents_i) = KnownNet_i \quad \wedge \quad \mathsf{ran}(Represents_i) = KnownNet_{i-1}$$

5.3 Journalizing Observed Events

Three journals are maintained: the one of observed events ($Monitored_i$) and two other ones of warned events ($FAIL_i$ and $CONFLICT_i$ respectively for the events which violate and are in conflict with the policy). All these journals are defined as non-ordered sets, because the considered policy does not take into account the history. Moreover, all warned events are observed and all events observed in a concrete level are also observed in the abstract level. At the policy level, no conflict can occur, then $CONFLICT_0$ is empty.

Invariant (General Journals Definition)

$$Monitored_0 \subseteq KnownNet_0$$
$$\wedge\ Monitored_i \subseteq Represents_i^{-1}[Monitored_{i-1}] \qquad (with\ i \in 1..3)$$
$$\wedge\ CONFLICT_0 = \emptyset$$
$$\wedge\ FAIL_i \cup CONFLICT_i \subseteq Monitored_i \qquad (with\ i \in 0..3)$$
$$\wedge\ FAIL_i \cap CONFLICT_i = \emptyset \qquad (with\ i \in 0..3)$$

Properties 1 and 2 can be expressed on these journals, by the next two invariants: (1) All observed events associated with *Pass* status are correct w.r.t. *SP*:

Invariant (Monitor Correctness - Property 1)

$$Represents_{i \rightsquigarrow 0}[Monitored_i - FAIL_i - CONFLICT_i] \subseteq SP \qquad (with\ i \in 0..3)$$

(2) No event associated with *Conflict* or *Fail* status is correct w.r.t. *SP*:

Invariant (Monitor Completeness - Property 2)

$$Represents_{i \rightsquigarrow 0}[FAIL_i] \cap SP = \emptyset \qquad (with\ i \in 0..3)$$
$$\wedge\ \forall e_i \cdot (e_i \in CONFLICT_i \Rightarrow Represents_{i \rightsquigarrow 0}[\{e_i\}] \not\subseteq SP) \qquad (with\ i \in 0..3)$$

5.4 Observer Introduction

The B event *Get_status* is an observer of the network events. It returns the status of an event chosen in a nondeterministic way. Because of the change of event representation, the observer is modeled with two new global variables: Obs_{Event} and Obs_{Status}. Obs_{Event} is the chosen observed event ($Obs_{Event\,i} \in KnownNet_i$) and Obs_{Status} is its status ($Obs_{Status\,i} \in \{Pass, Fail, Conflict\}$). Fig. 5 gives the general implementation of the observer *Get_status*.

The variables Obs_{Event} and Obs_{Status} are defined at each refinement level with the following invariant:

$$
\begin{array}{|l|}
\hline
\mathit{Get\_status} \mathrel{\hat{=}} \text{ ANY } e_i \text{ WHERE } e_i \in \mathit{Monitored}_i \text{ THEN} \\
\quad \text{IF } e_i \in \mathit{FAIL}_i \text{ THEN} \qquad\qquad\qquad \mathit{Obs}_{\mathit{Event}\,i} := e_i \parallel \mathit{Obs}_{\mathit{Status}\,i} := \mathit{Fail} \\
\quad \text{ELSIF } e_i \in \mathit{CONFLICT}_i \text{ THEN } \mathit{Obs}_{\mathit{Event}\,i} := e_i \parallel \mathit{Obs}_{\mathit{Status}\,i} := \mathit{Conflict} \\
\quad \text{ELSE} \qquad\qquad\qquad\qquad\qquad\quad \mathit{Obs}_{\mathit{Event}\,i} := e_i \parallel \mathit{Obs}_{\mathit{Status}\,i} := \mathit{Pass} \\
\quad \text{END} \\
\text{END} \\
\hline
\end{array}
$$

Fig. 5. General definition of the *Get_status* event

Invariant (Observed Variables)

$$
\begin{aligned}
&\mathit{Monitored}_i \neq \emptyset \Rightarrow \\
&\qquad ((\mathit{Obs}_{\mathit{Status}\,i} = \mathit{Fail}) \Leftrightarrow (\mathit{Obs}_{\mathit{Event}\,i} \in \mathit{FAIL}_i)) \\
&\qquad \wedge\ ((\mathit{Obs}_{\mathit{Status}\,i} = \mathit{Conflict}) \Leftrightarrow (\mathit{Obs}_{\mathit{Event}\,i} \in \mathit{CONFLICT}_i)) \\
&\qquad \wedge\ ((\mathit{Obs}_{\mathit{Status}\,i} = \mathit{Pass}) \Leftrightarrow (\mathit{Obs}_{\mathit{Event}\,i} \in \mathit{Monitored}_i - \mathit{FAIL}_i - \mathit{CONFLICT}_i))
\end{aligned}
$$

The observed event is traced through the refinement process:

Invariant (Relation through Refinement)

$$(\mathit{Obs}_{\mathit{Event}\,i}, \mathit{Obs}_{\mathit{Event}\,i-1}) \in \mathit{Represents}_i$$

Finally, correctness and completeness of the monitor (Properties 1 and 2) are implemented on the journals with the invariants defined in Section 5.3. However, these properties can also be checked on the observed variables. So, if the following assertions hold, then Properties 1 and 2 are verified:

Assertion 1 (Monitor Correctness on Observed Variables)

$$\mathit{Monitored}_i \neq \emptyset\ \wedge\ \mathit{Obs}_{\mathit{Status}\,i} = \mathit{Pass} \Rightarrow \mathit{Obs}_{\mathit{Status}\,i-1} = \mathit{Pass} \qquad \textit{(with } i \in 0..3\textit{)}$$

Assertion 2 (Monitor Completeness on Observed Variables)

$$
\begin{aligned}
&\mathit{Monitored}_i \neq \emptyset \Rightarrow \qquad\qquad\qquad\qquad \textit{(with } i \in 0..3\textit{)} \\
&\qquad (\mathit{Obs}_{\mathit{Status}\,i} = \mathit{Fail} \Rightarrow \mathit{Obs}_{\mathit{Status}\,i-1} = \mathit{Fail}) \\
&\qquad \wedge (\mathit{Obs}_{\mathit{Status}\,i} = \mathit{Conflict} \Rightarrow \mathit{Represents}_{i \rightsquigarrow 0}[\{\mathit{Obs}_{\mathit{Event}\,i}\}] \not\subseteq \mathit{SP}) \\
&\qquad \wedge (\mathit{Obs}_{\mathit{Status}\,i} = \mathit{Conflict} \Rightarrow \mathit{Represents}_{i \rightsquigarrow 0}[\{\mathit{Obs}_{\mathit{Event}\,i}\}] \cap \mathit{SP} \neq \emptyset)
\end{aligned}
$$

Therefore we only discuss the verification of those assertions that are sufficient to show Properties 1 and 2.

6 Model Description

In the previous section, we have presented, in a systematic way, all data required for the model development. In this section, we describe successively each level by introducing: the configuration data, the B events and the construction of the $\mathit{Represents}_i$ relation. All invariants and properties described in the previous section are included at each description level.

In this description, we also focus on the verification of the correctness and the completeness properties of the monitor by checking Assertions 1 and 2.

6.1 Level 0: User-Service View

The *SP* constant set is given by the user while constants *Users* and *Services* are retrieved from configuration files. Journals are represented as abstract variables and are empty in the initial state ($Monitored_0 := \emptyset$ and $FAIL_0 := \emptyset$). Consequently, the observed variables are initially undefined ($Obs_{Event0} :\in Net_0 \wedge Obs_{Status0} :\in \{Pass, Fail, Conflict\}$). If an event e_0 occurs on the network then:

- if e_0 is not in the observed sub-network then *Event_filter* (Figure 6.B) is launched and e_0 is ignored,
- else *Check_event* (Figure 6.A) is launched and e_0 is stored in $Monitored_0$. Moreover, if e_0 violates the policy then it is journalized in $FAIL_0$.

A. *Check_event*	B. *Event_filter*
$Check\_event \mathrel{\hat{=}}$ ANY e_0 WHERE $e_0 \in KnownNet_0$ THEN $Monitored_0 := Monitored_0 \cup \{e_0\} \;\|\|$ IF $e_0 \notin SP$ THEN $FAIL_0 := FAIL_0 \cup \{e_0\}$ END END	$Event\_filter \mathrel{\hat{=}}$ ANY e_0 WHERE $e_0 \in Net_0$ $\wedge\; e_0 \notin KnownNet_0$ THEN skip END

Fig. 6. Code of B events *Check_event* and *Event_filter* at the policy level.

Finally, this level establishes Properties 1 and 2 by verifying Assertions 1 and 2 with the observed variables only, since there is no conflict at this level.

6.2 Level 1: Servers View

According to Table 3, the daemons sets (*Daemons* and *TerminalServers*) are now described. The relation between this level and the more abstract one, i.e. the policy level, is extracted from configuration files. Each daemon is configured with its registered users and its provided services. We model this information with two relations (*Provide* and *Used_By*) describing which user can be connected to a particular terminal server and which daemon provides a particular service. For example, the registered users list of a telnet server can be found in `/etc/passwd`.

$Represents_1$ **Definition:**

$$Used\_By \in TerminalServers \leftrightarrow Users \quad \wedge \quad Provide \in Daemons \leftrightarrow Services$$
$$\wedge\;\; Represents_1 = (Used\_By \;\|\; Provide)$$

A. *Check_event*	B. *Event_filter*
$Check\_event \mathrel{\hat{=}}$ ANY e_1 WHERE $e_1 \in KnownNet_1$ THEN $Monitored_1 := Monitored_1 \cup \{e_1\} \;\|\|$ LET E_0 BE $E_0 = (Used\_By \;\|\; Provide)[e_1]$ IN IF $E_0 \cap SP \neq \emptyset$ THEN $FAIL_1 := FAIL_1 \cup \{e_1\}$ ELSIF $E_0 \not\subseteq SP$ THEN $CONFLICT_1 := CONFLICT_1 \cup \{e_1\}$ END END END	$Event\_filter \mathrel{\hat{=}}$ ANY e_1 WHERE $e_1 \in Net_1$ $\wedge\; e_1 \notin KnownNet_1$ THEN skip END

Fig. 7. Code of the B events *Check_event* and *Event_filter* at level 1.

The journals and the observed variables are defined according to the invariants given in Section 5. At this level of refinement, the relation $Represents_1$ is used dynamically by the B event *Check_event* (Figure 7.A) to compute the status of the observed network event, while the B event *Event_filter* (Figure 7.B) ignores all messages exchanged with the unknown part of the network.

The observer refinement (Fig. 5) produces 18 proof obligations, which, associated to the three ones generated for Assertions 1 and 2, establish Properties 1 and 2 on the model.

6.3 Level 2: Hosts View

As described in Table 3, this level introduces the notions of *Hosts* and *Ports*. The relation between these concepts and the daemons is extracted from hosts configuration information. They are summarized by two functions: *Hosting*, which associates the hosts to the daemons, and *Run_on*, which precises the ports used by a particular daemon on a host. Configuration data is such that:

$Represents_2$ **Definition:**

$$Hosting \in Hosts \leftrightarrow Daemons \quad \wedge \quad Run\_on \in Hosts \times Ports \mathrel{+\!\!\!\leftrightarrow} Daemons$$
$$\wedge \; Represents_2 = (Hosting \rhd TerminalServers) \;||\; Run\_on$$

Just as in previous levels, the journals and the observed variables are defined according to the invariants given in Section 5. The relation $Represents_2$ is used by *Check_event* (Figure 8.A) to compute the status of the observed network event, while *Event_filter* (Figure 8.B) ignores all messages exchanged with the unknown part of the network.

The main contribution of this level is the implementation of the *Check_event* event (Figure 8.A), which progressively refines the method to compute the E_0 set of all actions associated to the observed network events e_2.

```
Check_event ≙  ANY e2 WHERE e2 ∈ KnownNet2 THEN
  Monitored2 := Monitored2 ∪ {e2} ||
  LET E1 BE E1 = ((Hosting ▷ TerminalServers) || Run_on)[{e2}] IN
    LET E0 BE E0 = (Used_By || Provide)[E1] IN
      IF  E0 ∩ SP ≠ ∅ THEN
        FAIL2 := FAIL2 ∪ {e2}
      ELSIF  E0 ⊈ SP THEN
        CONFLICT2 := CONFLICT2 ∪ {e2}
      END
    END
  END
END
```

A. *Check_event*

```
Event_filter ≙ ANY e2 WHERE e1 ∈ Net2 − KnownNet2 THEN skip END
```

B. *Event_filter*

Fig. 8. Code of the B events *Check_event* and *Event_filter* at level 2.

In the same way as in the previous level, Properties 1 and 2 are established by proving the three proof obligations generated for Assertions 1 and 2 and the 32 proof obligations generated for the observer event *Get_status*.

6.4 Level 3: Implementation

At this level, hosts are valuated into their IP address (32 bit natural which identifies hosts for the Network layer) and ports remain unchanged. For example, the host anchieta.imag.fr can be valuated into its IP address 129.88.39.37 by using its 32 bit natural value[3]: 2170038053.

The $Represents_3$ relation is thus the identity and all invariants are inherited and do not need to be proven again. All other constants (security policy and configuration information) have also to be valuated. Network parameters can be retrieved in configuration files while the security policy has to be given by the administrator. Table 5 gives some concrete examples for Fedora-Core (a Linux distribution) of files containing usable data.

Refinement level	Constant	Data file
0 (Policy level)	*Users* and *Services*	`/etc/passwd` and `/etc/init.d/`
1	*Provide* and *Used_By*	Configuration files of each server
2	*Run_on* and *Hosting*	`/etc/services` and `/etc/init.d/`
3 (Implementation)	*Interfaces*	`/etc/hosts`

Table 5. Example of configuration files for Fedora-Core system.

However, the event-B language cannot be directly implemented. The model is translated into classical B. This transformation is done by some ad-hoc methods based on the results of the MATISSE[4] project [6, 3]. Since the guards of *Event_filter* and *Check_event* are disjoint, the events are replaced by the single operation *Check_event_and_Event_filter* (Fig. 9). Moreover, the network event e_3 chosen in the guard ANY e_3 WHERE $e_3 \in Net_3$ is replaced by three input parameters IP_1, IP_2 and Po representing respectively the IP address of the two hosts and the incoming port.

```
Check_event_and_Event_filter(IP1, IP2, Po) ≙ BEGIN
   /* Typing precondition: (IP1, (IP2, Po)) ∈ Net3 */
  VAR tmp IN
    tmp ⟵ Is_In_KnownNet(IP1, IP2, Po) ;
    IF tmp =TRUE THEN                                   /* Case of Check_event */
      Src := IP1 ; Dest := IP2 ; Port := Po ;
      tmp ⟵ Is_e3_Out_Of_SP(IP1, IP2, Po) ;
      IF  tmp =TRUE THEN Status := Fail ; WriteFail(IP1, IP2, Po)
      ELSE
        tmp ⟵ Is_e3_In_SP(IP1, IP2, Po) ;
        IF  tmp =FALSE THEN Status := Conflict ; WriteConflict(IP1, IP2, Po)
        ELSE Status := Pass END
      END
    END                                      /* Else case of Event_filter: skip */
  END
END
```

Fig. 9. Implementation of the B event *Check_event*.

[3] $2170038053 = ((129 * 256 + 88) * 256 + 39) * 256 + 37$

[4] Methodologies and Technologies for Industrial Strength Systems Engineering (MATISSE): IST Programme RTD Research Project (2000-2003).

Figure 9 is the implementation of *Check_event_and_Event_filter* operation. It uses three local operations (*Is_In_KnownNet*, *Is_e3_Out_Of_SP* and *Is_e3_In_SP*) to compute the correctness of each observed event. These operations are implemented as refinements of the corresponding parts of *Check_event* at level 2. For example, the local operation *Is_e3_Out_Of_SP* is defined in Figure 10.

$rr \longleftarrow Is\_e3\_Out\_Of\_SP(IP_1, IP_2, Po) \hat{=}$
PRE $(IP_1, (IP_2, Po)) \in KnownNet_3$ THEN
$rr = \mathsf{bool}((Used\_By \parallel Provide)[$ $\quad$ /*$Represents_1$[*/
$((Hosting \rhd TerminalServers) \parallel Run\_on)[$ $\quad$ /* $Represents_2$[*/
$\{(IP_1, (IP_2, Po))\}$ $\quad$ /* $\{(IP_1, (IP_2, Po))\}$ */
$]$ $\quad$ /*]*/
$] \cap SP = \emptyset)$ $\quad$ /*$] \cap SP = \emptyset$*/
END

Fig. 10. Abstract definition of the *Is_e3_Out_Of_SP* local operation.

The $Monitored_i$ set, modeled to store the monitored events, is not implemented, while *FAIL* and *CONFLICT* sets are stored in files. That is managed by an external component providing the operations *WriteFail* and *WriteConflict*. However, Properties 1 and 2 still hold, since the observed variable $Obs_{Status2}$ and Obs_{Event2} remain unchanged.

Finally, constants (configuration data and security policy) are exported in an external component as shown in Fig. 11. Thus, the model is generic and can be completely proved independently of the configuration data. The user just needs to provide some network information, or to retrieve it from configuration files, and to fulfill the conditions on configuration stated in Sections 5.3 and 5.2. A similar approach has been used in Météor [5], the Parisian subway without driver, to develop some generic and reusable components.

If the model is valuated for a simple example of network with two hosts, two daemons, one service and one user, then 31 proof obligations are generated and only 26 of them are discharged by the automatic prover. The five remaining proof obligations have been interactively discharged, but are really obvious and only need two commands : replace (`eh`) and predicate prover (`pp(rp.0)`).

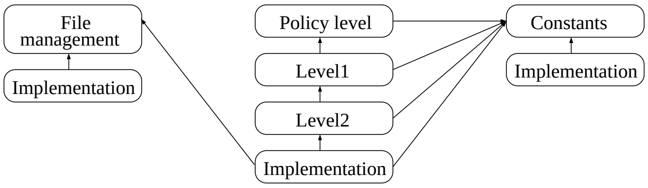


Fig. 11. General model organisation

7 Conclusion

This work has been done within the framework of the POTESTAT[5] project [9], which aims at proposing a methodology for network security testing from

[5] Security policies: test directed analysis of open networks systems.

high-level security policies. The main problem is to establish the conformity relation in order to automatically generate test cases and oracles from an abstract specification, as it has been implemented in the TGV tool [12] from IRISA and Verimag french laboratories.

Our contribution is a method to automatically enforce an abstract security policy on a network. In order to achieve that, we build a formal relation ($Represents_{i \rightsquigarrow 0}$) between abstract and concrete levels. The dynamic part of the program (*Check_event_and_Event_filter*) computes all actions associated to each observed event. Finally, we guarantee, by using an observer (*Get_status*), that all violations and conflicts are detected at each refinement level (Property 1) and that no warning is issued for an event which is correct with respect to the policy (Property 2).

The work of D. Senn, D. Basin and of G. Caronni [21] and of G. Vigna [23] are also dealing with the modeling of network conformity of a security policy. The model of [23] supports all the TCP/IP layers but does not provide a formal definition of the policy and needs human interaction to produce the test cases, while the model introduced in [21] only considers the first two layers and uses a low-level policy.

In this paper, we described the development of a network monitor, and the same approach can be used for generation, verification or test of a network configuration. The relationship is built in the same way and only the dynamic part of the model has to be modified.

In particular, many works have been developed relative to the generation of firewall configurations. For example, Firmato [4] is a tool generating the firewall configuration from a security policy and a network topology, and the POWER tool [18], of Hewlett-Packard, can rewrite a security policy into devices configuration. However, Firmato needs some topology information and uses a low-level policy, and POWER requires some human interactions during the process. The work presented here does not need human interaction (if all proof obligations are automatically discharged) or topology information. Due to the existence of the conflict status, it seems more adapted to the monitoring approach.

Finally, in our model, the conflict case can be removed if $Represents_{i \rightsquigarrow 0}$ is a function from $KnownNet_i$ to $KnownNet_0$, as done in [8] with a security policy expressed in the OrBAC framework [14]. In order to achieve that, we have to recognize the user and the service associated to each event. It can be realistic to associate only one service to each port, but it is too strict to impose that each host can be used by only one user. An investigation should be done to properly compare their approach with ours.

Acknowledgments. The authors would like to thank D. Bert, V. Darmaillacq, V. Untz, F. Dadeau and Y. Grunenberger for their advises and their reviews.

References

1. J.R. Abrial. Extending B without Changing it. In Nantes H. Habrias, editor, *First Conference on the B method*, pages 169–190, 1996. ISBN 2-906082-25-2.
2. J.R. Abrial. *The B-Book*. Cambridge University Press, 1996.

3. J.R. Abrial. Event driven sequential program construction. Technical report, ClearSy, 2001.
4. Y. Bartal, A. Mayer, K. Nissim, and A. Wool. Firmato: A novel firewall management toolkit. *ACM Transactions on Computer Systems*, 22(4), 1999.
5. P. Behm, P. Benoit, A. Faivre, and J-M. Meynadier. Météor: A Successful Application of B in a Large Project. In J. M. Wing, J. Woodcock, and J. Davies, editors, *FM'99 : World Congress on Formal Methods*, volume 1708 of *LNCS*, pages 369–387. Springer-Verlag, 1999.
6. M. Butler. A System-Based Approach to the Formal Development of Embedded Controllers for a Railway. *Design Automation for Embedded Systems*, 6(4), 2002.
7. Common Criteria. *Common Criteria for Information Technology Security Evaluation, Norme ISO 15408 - version 3.0 Rev. 2*, 2005.
8. F. Cuppens, N. Cuppens-Boulahia, T. Sans, and A. Mige. A formal approach to specify and deploy a network security policy. In T. Dimitrakos and F. Martinelli, editors, *Formal Aspects in Security and Trust (FAST)*. Springer, 2004.
9. V. Darmaillacq, J.-C. Fernandez, R. Groz, L. Mounier, and J.-L. Richier. Eléments 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.
10. D. Denning and P. Denning. Data Security. *ACM Computing Survey*, 11(3):227–249, 1979.
11. S. Jajodia, P. Samarati, and V. S. Subrahmanian. A logical language for expressing authorizations. In *IEEE Symp. on Research in Security and Privacy*, 1997.
12. C. Jard and T. Jeron. TGV: theory, principles and algorithms. *International Journal on Software Tools for Technology Transfer (STTT)*, 7(4):297–315, 2005.
13. JTC1. Information technology – Open Systems Interconnection (OSI model). Technical report, Standard ISO 7498, 1997.
14. A. Abou El Kalam, R. El Baida, P. Balbiani, S. Benferhat, F. Cuppens, Y. Deswarte, A. Mige, C. Saurel, and G. Trouessin. Organization Based Access Control. In *IEEE 4th International Workshop on Policies for Distributed Systems and Networks (POLICY'03)*, pages 120–131, 2003.
15. L. Lamport. The Temporal Logic of Actions. *ACM Transactions on Programming Languages and Systems*, 16(3):872–923, may 1994.
16. T. E Lunt. Access control policies for database systems. *Database Security II: Status and Prospects*, pages 41–52, North–Holland, Amsterdam, 1989.
17. M. Masullo. Policy Management: An Architecture and Approach. In *IEEE First International Workshop on Systems Management*, pages 13–26, 1993.
18. M. Casassa Mont, A. Baldwin, and C. Goh. POWER Prototype: Towards Integrated Policy-Based Management. Technical report, HP Laboratories, 1999.
19. P. Samarati and S. de Capitani di Vimercati. Access Control: Policies, Models, and Mechanisms. In R. Focardi and R. Gorrieri, editors, *Foundations of Security Analysis and Design*, volume 2171 of *LNCS*, pages 137–196. Springer, 2000.
20. R. S. Sandhu, E.J. Coyne, H.L. Feinstein, and C.E. Youman. Role-Based Access Control Models. *IEEE Computer*, 29(2):38–47, 1996.
21. D. Senn, D. Basin, and G. Caronni. Firewall Conformance Testing. In F. Khendek and R. Dssouli, editors, *TestCom*, volume 3502 of *LNCS*. Springer, 2005.
22. M. Sloman. Policy Driven Management for Distributed Systems. *Journal of Network and Systems Management*, 2(4):333–360, 1994.
23. G. Vigna. A Topological Characterization of TCP/IP Security. In K. Araki, S. Gnesi, and D. Mandrioli, editors, *FME*, volume 2805 of *LNCS*. Springer, 2003.
24. T. Y.C. Woo and S.S. Lam. Authorization in Distributed Systems: A Formal Approach. In *Symposium on Security and Privacy*. IEEE Computer Society, 1992.