Executable Security Policies: Specification and Validation of Security Policies

Ryma Abassi[1] and Sihem Guemara El Fatmi[2]

[1]Sup'Com, University of 7th November, Tunisia
abassi.ryma@gmail.com
[2]Sup'Com, University of 7th November, Tunisia
sihem.guemara@supcom.rnu.tn

1 Abstract

Security Policies constitute the core of network protection infrastructures. However, their development is a sensitive task because it can be in opposition with the security requirements (e.g. lack of rule or conflicting rules). A specification task seems to be indispensible in order to clarify the desired exigencies. A validation process for security policies becomes then necessary before their deployment to avoid resources network damages. Nowadays, there is no automated tool in the network security world allowing such task. Moreover, we have found that the theory developed for this aim in the software engineering domain can be adapted for security policies because several similarities exist between the expressions of the needs in the two domains as mentioned in several studies. Hence, we propose in this paper a specification and validation framework for security policies, inspired from software engineering tools, where: (1) we introduce the concept of executable specifications to build the concept of Executable Security Policies (2) we propose a new specification language based on an adapted modeling and inspired from Promela (3) we build a validation model based on the newly introduced language and (4) we define a 3-steps validation process of the executable security policy. The validation process is based on the main security properties, i.e. consistency, completeness and preservation of safety and liveness. Moreover, the consistency related to multiple security policies is treated through a detection algorithm and a resolution method.

2 Keywords

Security Policy, Executable Security Policy, Specification, S-Promela, Validation, Consistency, Completeness, domain, conflict.

1. Introduction

Organizations are aware about the importance of securing their information systems to guarantee basic security requirements, i.e. confidentiality, integrity and availability. Hence, security solutions that are implemented without any previous analysis of the real security needs can lead to important network assets damages. In order to prevent such lacks, it is relevant to define before implementing any security solution, a set of security rules defining for each request an adequate response allowing or denying the access according to the network security requirements. When defined, these rules are grouped in a document and are commonly called 'Security Policy' (SP). So, it is essential to prove that the rules composing the SP are conforming to a set of the security properties defined by the organization owner of the network. The proof can be obtained by a validation process. Unfortunately, there is nowadays no automated tool allowing this task in the security domain. The work presented in this paper proposes an automated environment allowing the specification and the validation of SP. This method is inspired from the theory established in the software engineering domain that presents several similarities with the security domain concerning the two previous aspects [2, 3] that are essential in any system development project and have been favorably considered. Their aim is to determine whether the requirements for a system or a component are complete and correct and if the product of each development phase fulfils the requirements or conditions imposed by the previous phase. They also determine whether the final system or component complies with specified requirements. Hence, one of the major contributions of this paper is the use and/or adaptation of tools and principles defined in software engineering to manipulate the SP specification and validation. Mainly, we have focused on the executable specification concept as a specification technique for validation purposes and have presented an approach to validate a SP based on its executable specification.

Our contribution is 4-fold. First, the executable specification concept is introduced as a useful tool to formally represent the SP components and hence the executable security policies (ESP) concept. Second, a specification language inspired by Promela is proposed. This language is based on a formal SP modeling intended to support the representation of all the aspects inherent to SP and to provide the basis facilitating their validation. Hence, a well formed syntax is proposed as well as a clear semantics. This latter cope principally with the system state representation, the transition concept definition and the reachability graphs construction i.e. represent the system state evolution). Third a validation process checking whether a candidate security solution is conforming to a SP is proposed. This process, which is based on the construction and the verification of the RGs, compares the evolution of the studied system while running according to the SP or without it. The main shortcut of such proposition is that it may return infinite state sets and consequently, infinite RG that are impossible to use. To remedy to such problems, we introduced two hypotheses (uniformity and regularity) allowing the reduction of the size of states set while preserving mandatory properties. Fourth, the special case of multiple security policies environment is treated. In fact, consistency proving is different in such environment.

The remaining part of this paper is structured as follows. Section 2 presents some Security Policies basics. Section 3 introduces the concept of Executable Security Policy as well as its inherent concepts'. Section 4 introduces S-Promela, our Executable Security Policy specification language through a well defined syntax and a clear semantics. In Section 5, a three-step validation process is proposed in order to deal with the consistency proving, the completeness proving and the preservation of security properties proving. Finally, Section 6 concludes this paper.

2. Security Policies Basics

The RFC 2196 [1] defines a SP as a "*formal statement of the rules by which people who are given access to an organization technology and information assets must abide*". More generally, the main objective of a SP is to maintain the principles of the organization's general security strategy. These principles cover several aspects such as detailed in [1]. However, due to the diversity of these aspects, SP definition may generate some inconsistency or contain errors concerning for example the needs expression. To avoid such problems, each SP definition requires a validation process to check if the policy matches the security needs. The deployment of such process is generally made through a SP modeling.

SP modeling constitutes a very important task because it helps the definition of the security rules and allows their validation. We have modeled in previous works [2,3,4] a SP as a communication mean following the several rules composing the SP, where a subject *s* reach an object *o* only if the requested action *a* is granted by the SP. According to the previous SP definition and considering the whole system in which a SP can be deployed, we have found that a modeling task requires the definition of the following concepts: (1) *subject* (s) that represents an active entity in the system like human users, employees, processes, applications or programs (2) *object (o)* that represents a passive entity in the system like ports, data or hosts (3) *action (a)* that represents an action that can be performed by a subject on an object like connection or read and/or write requests (4) *constraints (c)* that used to precise an action applicability scope (5) *events (e)* that are triggers of rules and (6) *security rule* that expresses the appropriate security decisions (allow or deny) to be taken for each action attempt made by an object relatively to a specific object.

These concepts are useful for the modeling of rules on which a SP can be based. For our part, we have found that the following rules are sufficient to represent a SP and to formally specify SP while specification is used principally to support the verification of the conformance of the SP with the defined security requirements. This functionality is called validation. It is performed at the specification level through the use of specific tools.

Authorization rule: allows making difference between the authorized and the unauthorized subject's actions. It could be viewed as a request for which a response is expected. Such rule can be expressed as follows:

$$req(s \times o \times a \times c \times [e]) \rightarrow resp$$

where *resp* is the response expected by the security rule. This response may evolve over time i.e. according to the satisfaction of certain constraints; it can be *yes* or *no*. For example, someone trying to access to his office is authorized to so only during work hours. Formally, this situation can be expressed as: *req(employee, access, office, time,-) → yes* if time is during work hours or *req(employee, access, office, time,-)→ no* if time is outside work hours.

Obligation rule: expresses actions that a subject *s* is forced to perform in response to the occurrence of some event *e*. Such rule can be expressed as follows:

$$ob\ (s \times a \times o \times [c] \times e)$$

An obligation rule can be considered as an ECA rule (Event-Condition-Action) e.g "ON event IF condition DO action".

For example, *ob (teacher, return, student-notes,-, at the latest 3 days after the exam)* means that a '*teacher*' has to '*return*' '*student-notes*' '*at the latest three days after the examination'*.

Prohibition rule: states that the SP prohibits the occurrence of a certain action in the protected system. The prohibition syntax is similar to the request rule syntax in that sense that it is a request made by a subject and to which the SP must respond. However, prohibition response is always '*no*'. Formally, it is expressed by:

$$phb\ (s \times a \times o \times [c] \times [e]) \rightarrow no$$

For example, someone trying to withdraw money from a bank account that does not belong to him will always be forbidden to carry out this action. Formally, this can be expressed as: *phb (client, withdraw, money, foreign account, -) → no*.

Delegation rule: enables a given subject *s* to delegate his permissions (according to an existing SP) to perform a given action *a* to another subject *r* who wasn't initially able to perform them. Formally, it is expressed by:

$$delg\ (s \times [a] \times [o] \times [c] \times r \times associated\text{-}rule) \rightarrow resp$$

where *r* (recipient) is the delegation beneficiary, *associated-rule* is the rule by which *s* has a given permission and *resp* can have the value 'yes' (positive delegation) or 'no' (negative delegation). For example, the following rule states that an administrator cannot delegate his right to modify passwords to students. This right was accorded to administrator by the rule r1.

delg (administrator, change, password, -, student, r1) → no

Let's note that '[]' introduce an optional argument and that '-' replaces an empty argument e.g. an optional argument not used in the rule.

In addition to the rules, another concept can be associated to a SP: the domain, usually noted by *Dom (SP)*. In [14], Hosmer defines a policy domain as "a logical construct defining the area of responsibility of an authority". In this work, we propose the following formalisation:

DEFINITION 1 **(SP Domain)** *a SP domain is a set of objects (O), subjects (S) and actions (A) where any subject can potentially manipulate any object through any operation according to the SP. It can be expressed by:*

$$Dom\ (SP) = S \times O \times A.$$

This definition leads to a rule domain definition that corresponds to a given object, subject and operation where the subject can handle the object through the action.

3. Executable Security Policies

Let's recall that we have found it useful to associate to SP specific tools allowing their representation, proof and verification. Moreover, we have found several similarities between SP engineering and software engineering concerning the three previous aspects handling such mentioned and used in [3]. So, one of the major contributions presented in this paper concerns the use of the tools and principles defined in software engineering to manipulate the major SP aspects. Our aim is to define a SP by the mean of a formal specification and to validate it by the mean of executable specification, like it is usually made in the software engineering domain. So, because an executable specification can be considered as an extension of formal specification, we have found it useful to propose Executable Securities Policies as an extension of Security Policies.

3.1. **ESP Definition**

A SP can be viewed as a specification for security solutions as well as a software specification has been defined as "*a document that prescribes, in a complete, precise, verifiable manner, the requirements, design, behaviour, or characteristics of a system or system component*" [5]. Similarly, a SP can be depicted as giving a precise description of the required behaviour of any secured network or network component. In addition, SP, like software specification must have a clear syntax and a precise semantic.

However, SPs present some differences with software specifications. The main one is that software specifications declare a process, using a modeling method, while a SP is just a set of requirements. The second difference is that manipulated variables in software specifications are essentially predefined types such as integers and floats, while objects and subjects manipulated in SP are elements of the network such as work stations, servers, routers, firewalls and switches.

In this paper, only the similarities between software engineering and security engineering are considered and used to construct the whole of our contribution.

In [2], we have proposed the concept of Executable Security Policy (ESP) as a mean of SP validation. In fact, we defined an ESP as *a SP model that can generate the expected behaviour of a secured system communicating with its environment according to the security exigencies specified by the SP*. Moreover, when using ESP, the behaviour of the SP can be observed and tested before it is actually performed on the desired system.

3.2. **ESP Modeling basis**

The ESP representation needs the use of an adequate specification language. In our context and knowing the studied environment, we have found interesting to propose a new specification and validation model inspired by Promela [6]. This choice is motivated by the following three reasons. Firstly, Promela's type objects can be adequately used to represent the SP model components depicted previously e.g. subjects by variables, rules by processes, etc. The second reason justifying our choice is that a SP, like a protocol, formalizes the interaction of subjects with their environment by standardizing the use of network assets. There is a third reason allowing a proposition of a like-Promela model: Promela is associated with a model checker (SPIN) that: (1) provides diagnostic information in the case where the property is not validated (counterexample); (2) supports partial validation (no complete requirement specification is needed) and (3) uses temporal logic that is required when specifying SP.

Figure 1 represents a SP as a mean of communication between two network components where the communication is made following the several rules composing the SP. In this Figure, four actors are depicted: the subject, the object, the SP and the trigger of events. All potential interaction between a subject and an object must be made through the SP i.e. a subject cannot interact directly with an object.

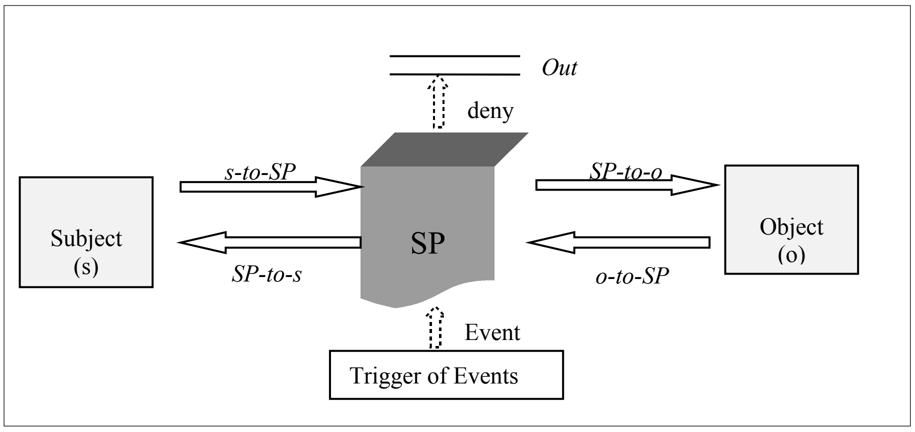

Figure 1. ESP modeling

In this Figure, the communication channel is split into four half duplex channels depending on the actor where the request come from and the actor where the request is addressed. A subject *s* submits his request via the channel *s-to-SP*. The SP verifies the legitimacy of the request from the set of SP rules. In the case where the request is granted, it is transmitted to the corresponding object *o* via the *SP-to-o* channel. The response of this request is then sent back by the object via the *o-to-SP* channel. Once received by the SP, this response reaches the subject *s* via the *SP-to-s* channel. In the case where the requested access is denied, the SP reject it into the *out* channel and delivers to the subject an error message without implying the *SP-to-o* and *o-to-SP* channels. Moreover, each channel can be accessed either for insertion or extraction. Hence, a mode is associated to each one of these operations: the *write* mode for insertion and the *read* mode for extraction.

The model represented by the Figure 1 considers also a trigger of events allowing the generation of all potential events for which the SP must react. These events are useful for obligation rules as explained previously.

Let's note that the model depicted by Figure 1 is a generic one. It can be customized following a particular rule type. In our context, three customized models respectively represented by Figure 2, Figure 3 and Figure 4 can be defined as follows.

3.2.1. **Authorization rule modeling**

As depicted by Figure 2, such rule can be described by the following elementary operations:

(1) A request is sent by s (s-write).
(2) The request is extracted by the SP (SP-read) and its legitimacy is verified.
(3) In the case where this request is granted, the SP forwards it in the adequate channel corresponding to the destination object (SP-write).
(3') However, if the request is denied, then it is simply dropped by SP into the channel out while a reject notification is sent back to s.
(4) The object o extracts the request (o-read).
(5) , (6), (7) and (8) allows the object response to reach the subject.

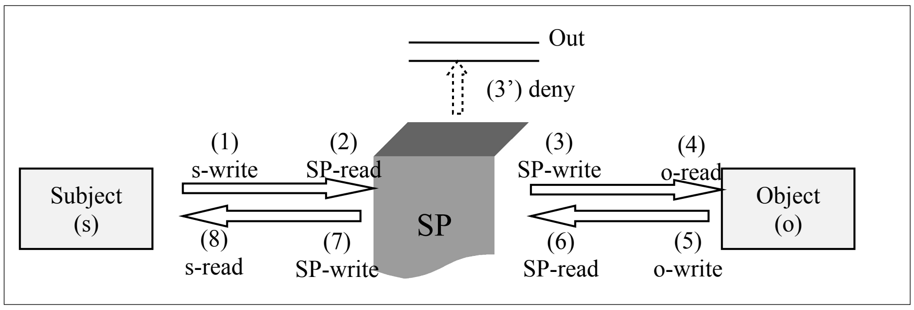

Figure 2. Authorization rule modeling

3.2.2. **Obligation rule modeling**

Assuming that the SP has a predefined table containing all the events for which it must react as well as their corresponding procedures, Figure 3 depicts the obligation rule modeling by the following elementary steps:

(1) The event *e* is triggered,
(2) The SP initiates the corresponding obligation procedure to inform *s* that it has to perform a particular action relatively to a particular object *o* (*SP-write*).
(3) The subject *s* extracts this action (*s-write*).
(4) The subject *s* inserts the action into the SP channel (*s-write*).
(5) The SP receives this action (*SP-read*).
(6) The SP forwards the received action without any verification, to the object *o (SP-write)*.
(7) The object *o* extracts the action (*o-read*) which is so, performed.

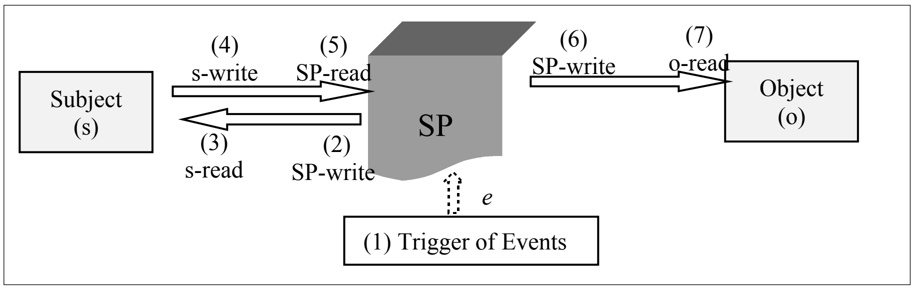

Figure 3. Obligation rule modeling

3.2.3. **Prohibition modeling**

As represented by Figure 4, an obligation rule is depicted by the following elementary steps:

(1) A subject s sends a request to the SP that is known to be prohibited (s-write),
(2) The SP extracts the action (SP-read).
(3) The SP drops it on the Out channel and sends back a reject notification to the subject (SP-write).
(4) The subject s is then acquainted with the prohibition (s-read).

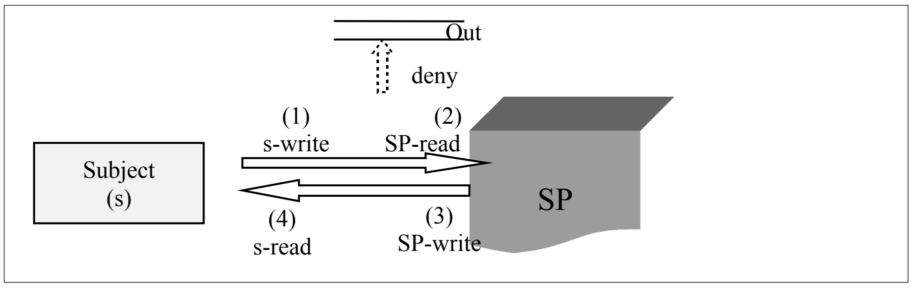

Figure 4. Prohibition rule modeling

3.2.4. **Delegation modeling**

A delegation rule expresses what a subject can delegate (or not) to another subject. Hence, it can be seen as authorization request handling with a right to delegate (by analogy to a request dealing with an action to perform). In fact, when a subject requests to delegate a given right, the SP verify the legitimacy of such request and then grants it or not. If the request is granted, it is forwarded to the recipient. But, if it is forbidden, then it simply dropped (into the channel out) and a notification is sent back to the subject. Hence, Figure 2 is still valid and depicts the elementary operations generated by a positive delegation rule:
(1) The request is inserted into the channel by the subject (s-write).
(2) The SP extracts the request (SP-read).
(3) The SP inserts the delegation request into the adequate channel (SP-write).
(4) The object o (the recipient) extracts the request (o-read) and so, appropriates the rights.
However, steps (5), (6), (7) and (8) are not involved in such communication.

4. S-PROMELA: AN ESP SPECIFICATION LANGUAGE

Although, Promela is not adequate to specify SP because it was initially developed for SE, it offers interesting concepts and basis that can be useful for SP domain. Hence, the aim of this section is to introduce a new Promela based language, called *S-Promela* (Security-based Promela). This language resumes the fundamental Promela basis and extends them with some SP specificities needs. Moreover, *S-Promela* allows specifying in a precise way the desired behavior of a subject interacting with objects and thus, in conformance with the SP rules. To be able to define such language, we assume that each subject interacting, in a secured system (e.g. by the use of a SP), with a given object does it through the SP and using elementary statements, that we call security-policy based primitives.

4.1. S-Promela Syntax

An S-Promela specification defines a set of process. A process describes a rule.

SP	*S-PromelaSPec*	*::=*	*Procs*
	Procs	*::=*	*Procd* |*Procs*
	Procd	*::=*	*Rule*

Figure 5. S-Promela Specification structure

A '*Rule*' is an iterative expression. As depicted by Figure 6, while a given condition holds, a sequence of actions is performed. A '*Condition*' can be either an expression or the occurrence of an event. A '*Sequence*' can be a primitive, a single action or another rule.

Rules	*Rule*	*::=*	*'While' '('* Condition*')' 'do'* Sequence [*andor* Sequence]*	
	Condition	*::=*	*Expr*	*|Evt-occur*
	Expr	*::=*	*Expr* *|Expr logic Expr*	*|Expr Binaop Expr* *|Sequence*
	Evt-occur	*::=*	*'occurs' '(' event ')'*	
	Sequence	*::=*	*'{' Primitive '}'* *|'{' Procd'}'*	*| '{' pfm-ation'}'*

Figure 6. BNF rule syntax

An important part of the S-Promela syntax is declaration. As depicted by Figure 7, an S-Promela specification can use several variables types e.g. channel, subject, object, action, message, event, constraint, notification. Moreover, a specification can include procedures, structures with procedure and tables. A '*Procedure*' is constituted by an action (or a set of actions) that must be performed. A '*Structure*' associates an event to a Procedure executed when the event is triggered. A '*Table*' is a set of '*Structure*'.

Declaration	*channel*	*::=*	*name*
	|subject	*::=*	*name*
	|object	*::=*	*name*
	|action	*::=*	*name*
	|message	*::=*	*name*
	|event	*::=*	*name*
	|constraint	*::=*	*name*
	|notification	*::=*	*name*
	|Procedure	*::=*	*'procedure' name '(' [event] ')'*
	|Struct	*::=*	*'struct' name '{' (event, Procedure) * '}'*
	|Table	*::=*	*Struct name '[' integer ']'*

Figure 7. S-Promela Declaration Part

Pre-defined Terms define constants and terms such as '*skip',* shorthand for a dummy; '*Entity*' that can be a subject, an object or a SP or '*Recipient'* that can be only a subject.

Pre-def terms	*Boolean*	*::=*	*true*	*|false*
	|Comment	*::=*	*/'*' comment '*'/*	
	|skip	*::=*	*'skip'*	

	|nil	*::=*	*'nil'*		
	|Pfm-action	*::=*	*'pfm-action' '(' Entity ',' Parameters ')'*		
	|name	*::=*	*char [char | number] **		
	|Entity	*::=*	*subject*	*|object*	*|SP*
	|Recipient	*::=*	*subject*		
	|Parameters	*::=*	*channel ',' [action | message] ',' object*		
			|action	*|event*	*|constraint*
	|Affectation	*::=*	*Var '=' Expr*		
	|Var	*::=*	*Entity*	*|event*	*|message*

Figure 8. Predefined terms BNF syntax

An S-Promela specification uses also control flow expressions as depicted by Figure 9. A '*Conditional*' expression resumes the classical if-then-else statement. A '*Separator*' allows enumerating expressions. The logical operators '&&' and '||' are used respectively for expressing conjunction and disjunction. Binary operators '*Binarop*' are used in order to perform expression comparison.

Ctrl-flow	*Conditional*	*::=*	*'If' '('* Condition *')'* '*then*' Sequence [*andor* Sequence]* [*'else'* Sequence]		
	|Separator	*::=*	*Expr ';' Expr*		
	|Andor	*::=*	*'&&'*	| *'||'*	
	|Binarop	*::=*	*'=='*	| *'<'*	| *'>'*
			| *'≠'*	|*'≥'*	|*'≤'*

Figure 9. Control flow BNF syntax

Figure 10 depicts S-Promela basic statements. In accordance to our modeling, introduced previously, two primitives can be used: '*write*' and '*read*'. The *write* statements allows to an entity (subject, object or SP) to insert into a channel where the *read* statements allows to an entity to extract from a channel.

Basic stmts	*|Primitive*	*::=*	*Write*	*|Read*
	|Write	*::=*	*entity '-' 'write' '(' parameters ')'.*	
	|Read	*::=*	*entity '-' 'read' '(' parameters ')'.*	

Figure 10. Basic statements BNF syntax

Let's recall that obligation rules are triggered by an event. More precisely, the trigger can be a single or a composed event. A composed event is either a '*Synchronization*' of events e.g. conjunction or disjunction, a '*Precedence*' between events or a '*Repetition*' of a given event.

Evt operators	*Synchronization*	*::=*	$\wedge$	|$\vee$
	|precedence	*::=*	*event1 → event2*	
	|repetition	*::=*	*n * event*	

Figure 11. Event composition operators BNF syntax

4.2. S-Promela Semantics

The S-Promela semantics, defines the behaviour of an S-Promela model by describing how the global directed graph of all reachable system state, for any given S-Promela model is to be generated. Similarly to Promela, the semantics of S-Promela is defined in terms of an operational model [6]. This model contains one or more processes, zero or more variables, zero or more channels, and a semantics engine' that defines how the actions of the processes may be interleaved in time. The processes are defined by Reachability Graphs (RG) defined as (P, P_S, $\sum$, $\rightarrow$) such that:

- P is the starting state.
- P_s is the set of states.
- $\sum$ is the set of labels = {write, read, evt-occur}.
- $\rightarrow \subseteq P_s \times \sum \times P_s$ is a transition relation.

Let's recall that seven components representing the studied environment have been revealed: (1) four unidirectional channels, (2) two operation modes, *read* and *write* and (3) a SP. Each channel can be characterized by a state observed at a given time. A channel state is defined by an ordered list of the operations inserted to it (by s-write**,** f-write) but haven't been extracted yet (by *SP-read***,** *o-read*). Any operation made on the channel, is registered as an event and leads to a channel content modification and so to a channel state modification. Formally, to each channel state $S_i \in P_s$, we associate the time t_i that corresponds to the occurrence time of the event leading to it.
Figure 12 depicts two successive occurrence channel state modifications observed at the instants t_{i-1} and t_{i+1} and generated respectively by the arrival of the requests r_{i-1} and r_{i+1} which are two legitimate SP inputs. Let's note that in this figure, the request r_i has been dropped because denied by the associated rule and hence, the current state S_{i-1} is maintained.

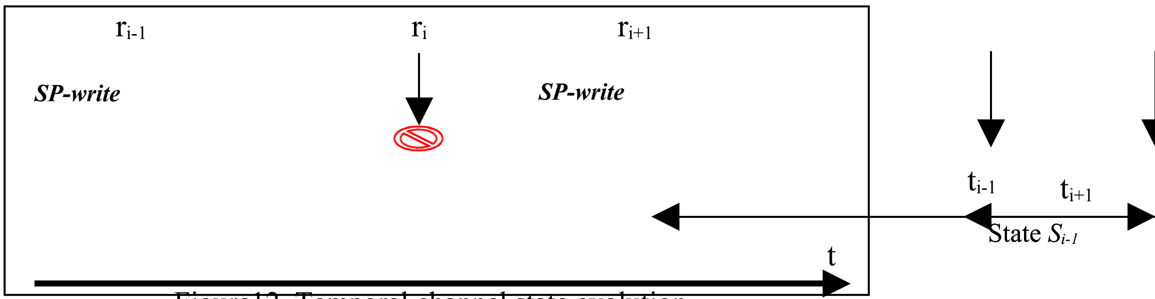

Figure12. Temporal channel state evolution

The channel state observed at a given time t_{obs} correspond to the channel state S_i established at the time t_i where $t_i < t_{obs}$ and t_i is the last time before t_{obs} corresponding to the last channel modification. Hence, the description of the system state is based on a set of couples depicting the manipulated input and the destination object.
As denoted by Figure 13, the system state observed at $t_{obs} \in [t_{i-1}, t_{i+2}]$ is defined by $\{t_i, (r_i, o_i), (r_{i+1}, o_{i+1})\}$ knowing that t_{i+1} is the time corresponding to the last observed event, (r_i, o_i), (r_{i+1}, o_{i+1}) are the content of the channel and (r_{i-1}, o_{i-1}) has been already extracted by the destination.

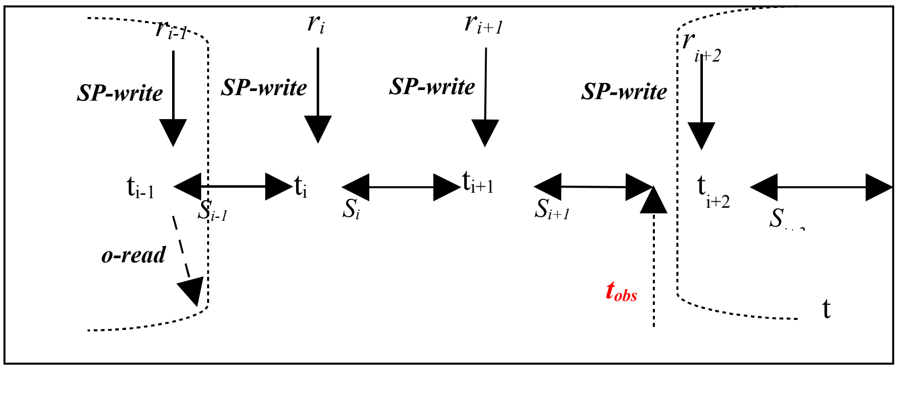

Figure13. System State Representation

Let's note that the considered channels are managed following a FIFO discipline in that sense that the extraction of the request r_{i-1} implies that all the precedent requests were already extracted.

5. Security Policies validation using Executable Security Policies

According to Lindsay [7], "the validation of a SP model can be done by showing that the specification is mathematically consistent, the security enforcing functions preserve the desired security properties, and the specification is complete with respect to its input space". Similarly, our SP validation process can be defined by 3 steps: (1) the consistency proof, (2) the completeness proof and (3) the SP properties preservation as presented by the remaining part of this section.

5.1. Consistency proof

In the software engineering domain, consistency is defined as a "property stating that there are no requirements that contradict each other" [8]. In SP context, we define consistency as a property stating that there are no rules that contradict each other. In order to prove consistency, we propose to look about inconsistencies and thus based on rules relations. These relations are based on the five rules components, i.e. *subjects*, *objects*, *actions, constraints* and *events*. Let R_i and R_j be two rules. We note by S_i (respc S_j) the R_i *(respc* R_j*)* subject set; by O_i (respc O_j) the R_i *(respc* R_j*)* objects set; by A_i (respc A_j) the R_i *(respc* R_j*)* action; by C_i *(respc* C_j *)* the R_i *(respc* R_j*)* constraint set and by E_i *(respc* E_j *)* the R_i *(respc* R_j*)* event set. Six relations can be observed between these sets. Let's note that these relations are widely used in the literatture [11]. So, we tried to propose a new formalization adapted to the SP field as well as two new relations appropriate for SP. This can be expressed by the following six definitions.

DEFINITION 2 **(Equality, '=')** *Rules R_i and R_j are equal if every component in R_i is equal to its corresponding component in R_j.* Formally, *Ri = Rj* iff *:*

t

$$S_i = S_j \text{ and } O_i = O_j \text{ and } A_i = A_j \text{ and } C_i = C_j \text{ and } E_i = E_j$$

DEFINITION 3 **(Intersection,'$\cap$')** *Rules R_i and R_j have an intersection if every component in R_i is a subset or a superset or equal to its corresponding component in R_j.*
Formally $R_i \cap R_j \neq \emptyset$ iff

$$\exists s, o, a, c, e \ /\ s \in S_i \cap S_j,\ o \in O_i \cap O_j,\ a \in A_i \cap A_j,\ c \in C_i \cap C_j,\quad e \in E_i \cap E_j \quad \text{And}$$
$$\exists s', o', a', c', e' \ /\ s' \notin S_i \cap S_j,\ o' \notin Oi \cap O_j,\ a' \notin A_i \cap A_j,\ c' \notin C_i \cap C_j,\ e' \notin E_i \cap E_j,$$

DEFINITION 4 **(Generalization**, '$\subset$'**)** *Rule R_i generalizes R_j if they do not exactly match and if every component in R_i is a subset or equal to its corresponding component in R_j.*
Formally, $R_j \subset R_i$ iff

$$\forall s \in S_j,\ \forall o \in O_j,\ \forall a \in A_j,\ \forall c \in C_j,\ \forall e \in E_j / s \in S_i,\ o \in O_i,\ a \in A_i, c \in C_i,\ e \in E_i \quad \text{And}$$
$$\exists s' \in S_i,\ o' \in O_i,\ a' \in A_i,\ c' \in C_i,\ e' \in E_i,\ / s' \notin Sj,\ o' \notin O_j, a' \notin Aj,\ c' \notin Cj,\ e' \notin Ej.$$

DEFINITION 5 **(Disjunction**, '$\neq$'**)** *Rules R_i and R_j are disjoint if there is at least one component of R_i does not interfere with its corresponding component in R_j.* Formally, *$Ri \neq Rj$* iff

$$S_i \cap S_j = \emptyset \text{ or } O_i \cap O_j = \emptyset \text{ or } A_i \cap A_j = \emptyset$$

In addition to these classical relations, we propose two new relations: dependence and cross dependence.

DEFINITION 6 **(Dependence**, '$\oplus$' **)** *Rule R_i is dependent of rule R_j if R_i subjects and objects are equal or a superset to their corresponding in R_i while the operation field of R_i is a superset to its corresponding in R_i.* Formally, $R_j \oplus R_i$ iff

$$S_i \supseteq S_j \text{ and } O_i \supseteq O_j \text{ and } A_i \subset A_j$$

DEFINITION 7 **(Cross dependence**, '$\otimes$' **)** *Rule R_i is cross dependent of rule R_j if R_i subjects are equal or a superset to their corresponding in R_i , the R_i objects are subset of their corresponding in R_i while the operation field of R_i is a superset to its corresponding in R_i .* Formally, $R_j \otimes R_i$ iff

$$S_i \supset S_j \text{ and } O_i \subset O_j \text{ and } A_i \subset A_j$$
$$or$$
$$S_i \subset S_j \text{ and } O_i \supset O_j \text{ and } A_i \subset A_j$$

The consistency proof proposed in this paper is summarized by Algorithm 1 in which the SP is checked by evaluating rules two by two. The domain of a rule corresponds to the set of potential packets that can be treated by the rule filter and for which it can give a response. The first action is then to compute the domain intersection *R*. If it is empty then R_i and R_j are consistent; else the second alternative is to look for an element, belonging to the intersection-rule, and that lead to two different responses. If such element exists, then an inconsistency is detected.

```
For each couple of rules (Ri, Rj)
do {  R ← dom (Ri) ∩ dom (Rj)
      if (R = ∅ ) then consistency
      else  if (∃p ∈ R / (resp_i(p)≠ resp_j(p)))
      then    inconsistency
```

Algorithm1. Consistency proof

We have to note, however, that this detection procedure has an exponential complexity. But, the examples we have considered have few rules, thus problem is still tractable.

5.2. Completeness proof

Completeness in the software engineering domain is defined as a "property stating that all significant requirements are included, and responses to all possible inputs are defined" [8]. In SP context, we define completeness as a property stating that all significant security requirements are included and that rules, allowing the SP to respond to all possible inputs, are defined.

In order to prove the completeness of SP, we propose to use a reachability analysis of the states set. This analysis is done via two RGs. Figure 12 depicts this proposition: the first step is to compute all the potential acceptable actions (that a subject can perform) from the initial state.

Each action leads to a state which is attached to the attempts RG $\mathcal{V}ar$ (the right one in the figure). Then, a label "secure" is associated to states declared as being acceptable. Moreover, the graph is supposed to be made finite-state. To that effect, a depth limit is imposed. The second step is to compute, parting from the same initial state, all the states that can be reached according to the SP rules. These states constitute the secure RG $\mathcal{V}sr$ (the left one in the figure).

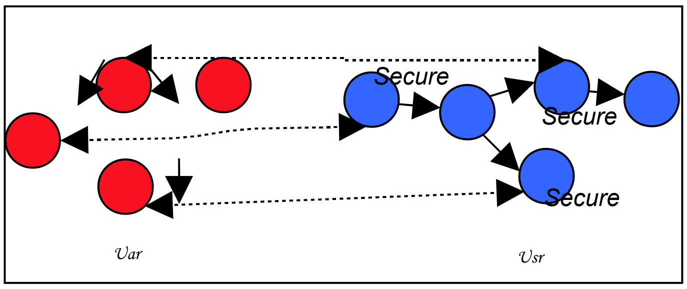

Figure 12. Completeness proof

The third step is hence, to compare the two graphs: each labelled state must appear in $\mathcal{V}ar$ otherwise, the SP is incomplete and rules must be added. The completeness problem is then reduced to the verification that all secure-marked states belonging to $\mathcal{V}ar$ belong also to $\mathcal{V}sr$.

5.3. Security properties preservation proof

Security properties are essentially integrity, confidentially, access control and availability [1]. In order to prove the preservation of these properties we started by associating them to two classes: liveness and safety. Then, we generated appropriate claims and proposed an adequate methodology to each one of the previous classes. The use of a model checker allows to validate (or invalidate) the previous properties. Generally, the latter is reduced to checking emptiness of automata. In our case, it has to prove safety and liveness properties. This can be done with the help of a reachability analysis of a finite state model as mentioned by the authors of [9]. The remaining part of this subsection deals with the proof of the two properties.

Liveness property: is a statement claiming that something "good" will "eventually" happen. In other words, a liveness property dictates that a given activity will eventually be performed, presumably because it is good and desirable. Good condition can be represented by an assertion P_L and the fact that it will eventually occur during the execution by $\Diamond P_L$.
For SPs, "good things" are actions that if happen do not compromise the system security. However, the formalisation presented above cannot be directly transposed to SP. In fact, a future response is not sufficient: a temporal limit must be specified and respected. Let us consider the context of the Dos attack. In this attack, a user is prevented from using a remote resource by, for instance, flooding the network with bogus messages. Here, the "good thing" is the possibility for a host to reply in a limited time. Hence, liveness is the guarantee of a maximum waiting delay for each operation. This can be formalized by: $p \Rightarrow \Diamond^d q$ where p and q are assertions and '$\Diamond^d$' denotes: finally before a given delay.

Safety property: is a statement claiming that something "bad" will not happen. In other words, a safety property dictates that a given activity will never be performed presumably because the activity is bad and undesirable.
For SP, "bad things" are actions that if they happen compromise the system security. Hence, SP safety properties are statement claiming that actions compromising system security never happen. Bad things can be represented by an assertion P_s which is mapped to true in exactly those states in which the condition is true. For a safety property to be true $\neg P_s$ must be an invariant. A property P_s is an invariant if for each execution $\rho = S_0 S_1 \ldots S_n$ where S_0 = initial state

and $S_1 \ldots S_n$ successor states and for any $i \in N$, $S_i \models P_s$. Thus, an invariant is expressed as $\Box P_S = \forall i\ P_S(Si)$.
The verification of such properties can be made by the use of invariants trough a search of the state space. First, we generate invariant corresponding to the compromising action. Second, while exploring the RG, the invariant is verified with a Boolean test. If a specification violates the invariant and consequently the safety property, then there is a finite behaviour that displays the violation. The following depicts an invariant modeling through the use of never claim as in Promela [6] and depicted as follows:

```
Never {/* two state machine: initial and final */
  do
     :: SP    /* must be verified in the initial state of the SP model */
     :: ! SP → break /*final state /
  od
}
```

5.4. Deriving Finite Reachability Graphs from Security Policies Specification

In order to validate a SP, two kinds of RGs are generated from it specification as presented previously. These graphs have to be finite, complete and correct. In fact, an exhaustive RG is not usable in practice since it is infinite. For this reason, we have adapted the techniques used by Gaudel in [10], especially those used to reduce an exhaustive set.
Let *SS* be a SP specification and *ESS* an ESP under validation. A validation is successful if it concludes to the satisfaction of the validation experiment by *ESS*, and we note it $ESS \models \Gamma$ where Γ is the validation, i.e. completeness properties. Given a specification *SP*, the exhaustive validation set for it, noted $Exhaust_{RG}$ is the set of all the states of the RG:

$$Exhaust_{RG} = \{\Phi_\sigma \mid \Phi \in SS,\ \sigma = \sigma_i : var(\Phi)_i \rightarrow T_i \mid i \in I\}$$

where T_i is the validation properties set and I is the set of states components.

An exhaustive validation of *ESS* against *SS* is the set of all the validation experiments of *ESS* against the states belonging to the exhaustive RG $Exhaust_{RG}$.
Because it is practically impossible to consider the exhaustive RG due to its infiniteness, we introduce the concept of hypotheses that is shown to considerably reduce the RG size.

Let's have $Exhaust_{RG}$, the validation set for the policy specification *SS* resulting from the construction of the exhaustive RG. Even though it covers all the validation space, this validation set is practically not useful to assess all SP implementations because it is often infinite. Hence, during RG construction, only a subset of $Exhaust_{RG}$ may be sufficient. However, some properties should be guaranteed by this subset. Eliminating infiniteness must be done according to a procedure that preserves the properties of $Exhaust_{RG}$. The two major requirements consist in the fact that the selected states set should be valid and unbiased [10]; meaning that incorrect SP implementations should be discarded, and that all correct SP implementations are accepted. To this end, we use selection hypotheses in order to reduce the exhaustive validation set (particularly RG) to a finite validation set (and consequently finite RG). Two hypotheses are considered: uniformity and regularity. Formally, they are expressed by the following definitions.

Definition 1: ***Uniformity hypothesis.*** Given a rule $\Phi(X)$ where X is a variable, a *uniformity hypothesis* on a sub-domain D for an ESP *ESP* is the assumption:

$$(\forall t_0 \in D)\ (P \models \Phi(t_0))\ (\forall t \in D)\ (P \models \Phi(t))$$

This hypothesis states that the validation result can be generalized to a whole domain if the validation is performed at a single point of this domain. This corresponds to the determination of sub-domains of the variables where the program is supposed to have the same behaviour. Assuming that, it is no more necessary to have all the ground instances of the variables but only one by sub-domain. Such criteria are modelled in our framework by uniformity hypotheses
Another type of hypothesis, called the regularity hypothesis, relies on generating states for several variables that do not exceed a defined 'size'. This notion of size can be customized to represent multiple aspects of SP objects.

Definition 2: ***Regularity hypothesis***. Given a vocabulary Σ, a rule Φ (X) where X is a variable, a function of interest |t|, a regularity hypothesis for a program P is the assumption:

$$((\forall t \in \Sigma)\ (|t| \leq k \Rightarrow P \models \Phi(t))) \Rightarrow (\forall t \in T_{\Sigma})\ (P \models \Phi(t)).$$

6. DEALING WITH MULTIPLE SECURITY POLICIES ENVIRONMENT

When various SPs are used, several problems can be observed, of whom we can note the three more important. The first one concerns a modeling problem: when several SP models are used (R-BAC [16], LaPadula [15] …), a communication problem due to the heterogeneity of these latter may occurs. The second problem concerns the SP implementation where several mechanisms can be employed for the same security service. The third problem, that constitutes our interest object in this paper, is related to a coexistence problem: various SPs are used and conflicting responses are obtained for the same request.
According to [12], a policy conflict "occurs when the actions of two rules that are both satisfied simultaneously contradict each other" and creates a problem because "the entity implementing the policy would not be able to determine which action to perform".
Similarly, a multi-policy conflict occurs when the interaction between a given subject and a given object can be achieved via several SPs for the same operation. This situation creates a conflict if the involved SPs give contradictory responses.
In the following sub-sections, a formalization of multi SPs conflicts is given based on a definition of potential SP domain relations.

6.1. Multi Policy Environment Conflict definition

According to the SP rules formalization introduced in Sub-Section 5.1, we have brought out two conflict kinds i.e. "modality conflict" and "type conflict". Modality conflicts are due to the existence of a domain relation (such presented above) as well as a rule modality difference. Type conflicts, however, are due to the existence of a domain relation as well as a rule type and modality difference.
Let's recall that we specified a rule by the following 7-uplets: (*type, modality, subject, object, action, [constraint], [event]*) and let's note by T_i, the type of the rule R_i *and* by M_i *its* modality.

DEFINITION 7 **(Modality Conflict)** *A modality conflict, between two rules R_1 and R_2, occurs if they have the same type ($T_1 = T_2$), a domain relation (equality, dependence, intersection or generalization) and different modalities ($M_1 \neq M_2$).*

According to this definition and to the previously introduced rules formalization, one can note two modality conflicts: the first one occurs when obligation/ interdiction rules are interacting. The second one occurs when positive request/ negative request rules are interacting.

DEFINITION 8 **(Type Conflict)** *A type conflict, between two rules R_1 and R_2, occurs if they have a domain relation (equality, dependence, intersection or generalization), different modalities ($M_1 \neq M_2$) and different types ($T_1 \neq T_2$).*

According to the rules formalization specified previously, two type conflicts can arise. The first one occurs when positive request / interdiction rules are interacting. The second one occurs when negative request/ obligation rules are interacting.

6.2. **Multi Policy Environment Conflict Detection**

The first step towards conflict resolution is the detection. In the following, we propose an algorithm for SP conflicts detection according to the formalization introduced above.

Let's start by giving the assumptions used in this algorithm. First, we associate to each rule the structure: *R= {type t, modality m, Subject s, Object o, Action a}*

Second, we assume that the intersection of two rules (let's say R_i and R_j) is a rule (let's say R) such that:

$$Ri \cap Rj \equiv \begin{array}{l} Ri.s \cap Rj.s \\ Ri.o \cap Rj.o \\ Ri.a \cap Rj.a \end{array} = \begin{array}{l} R.s \\ R.o \\ R.a \end{array}$$

Third, we note by D_i=*Dom*(R_i); D_j=*Dom*(R_j); *D= Dom(R).*

The detection algorithm can be then, depicted following two steps. In the first step, the intersection of rules domain is computed: all the presented relations can be inferred from the intersection i.e. they are special cases of the intersection. In the second step, this intersection is tested. If it isn't empty then a potential conflict may exist. Else (if the rule domains do not coincide), the rules are disjoint and there is no conflict. These steps are detailed as follows.

```
Input: Ri ; Rj
Output: conflict

1: For each couple of rules (Ri, Rj) do
2:    If mi ≠ mj  then {different modalities}
3:        D ← Dom(Ri) ∩ Dom(Rj)
4:    if (D = Di = Dj) then {Domains are equal}
5:                 relation ← true
6:                 conflict ← EQUALITY
7:          else if ((D = Di ) or (D = Dj)) then {one domain is completely included in the other}
8:                    relation ← true
9:                    conflict ← GENERALIZATION
10:          else if  (R.a = Ri.a) or (R.a = Rj.a)  then  {this rule is dependent of the other}
11:                   relation ← true
12:                  conflict ← DEPENDENCE
13:            else
14:                    relation ← true
15:                   conflict ← INTERSECTION
16:         end if
17:   else  {no relations exist}
18:            relation   ← false
19:   end if
20:  if ti ≠ tj and mi ≠ mj  and relation = true then {different types and modalities but there is a domain relation}
21:               conflict ← TYPE {type conflict}
22:   end if
23: end for
```

Algorithm2. SP conflict detection Algorithm

Algorithm2 discovers rule conflicts by implementing the rule relations presented previously. Moreover, the algorithm can detect modalities conflicts corresponding to domain relations (lines 3-19) as well as type conflicts (lines 20-22).

6.3. **Multi Policies Environment Conflicts Resolution**

The resolution approach proposed in this paper creates a specific view each time a conflict situation occurs. This view considers all the non conflicting rules added to a set of alternative rules, each alternative rule can be one of the conflicting rules or a new rule, defined by the conflict resolution process. The alternative rule choice can be done according to two extreme approaches:

Permissive approach: states that the positive rule is retained. However, this solution is not secure because it may allow the achievement of forbidden actions.

Restrictive approach: states that the negative rule is retained. However, this decision is also not useful because it is too limitable.

In the following, a resolution approach mixing the permissive and the restrictive approaches and based on SP combination is presented i.e. generating a conflict free SP view from two conflicting SP as defined in DEFINITION 10.

DEFINITION 9 **(rule combination)** *A combination of two rules (R_1 and R_2) associated respectively to D_1=Dom (R_1) and D_2=Dom (R_2) is a symmetric operation that can modify R_1 and R_2 and/or generates a new rule R_3 with a domain D_3 and that using some alternative rules.*

DEFINITION 10 **(SP combination)** *A SP combination is the set of rules combination where the combination of two sub SPs (SP_1 and SP_2) generates a new policy SP_3.*

The combination process is based on two steps: modality conflict resolution and type conflict resolution.

6.3.1. Modality conflict resolution

When two conflicting rules have the same type but different modalities, they are treated according to the relation they're involved in. Formally, given two SPs, *SP1* and *SP2* such that *R1* $\in$ *SP1* and *R2* $\in$ *SP2*,

Equality conflict: in this case, the retained rule can be the negative one if the restrictive approach is privileged or the positive one if the permissive approach is the privileged.

$$\text{If } R1=R2 \text{ then } SP3 \leftarrow \{R1 \text{ or } R2\}^1$$

Generalization conflict: the proposed solution is to conserve the more specific rule and to modify the more general to remove the common part (already treated by the first rule).

$$\text{If } R1 \subset R2 \text{ then } SP3 \leftarrow \{R1 \text{ and } R2 \backslash R1\}$$

where ' \' stands for except.

Intersection conflict: the proposed solution is to add a third rule dealing with the common part while removing it from the two existing rules.

$$\text{If } Rc = R1 \cap R2 \text{ then } SP3 \leftarrow \{R1 \backslash Rc, R2 \backslash Rc, Rc\}$$

The problem arise here is the *Rc* modality. Our proposition is to apply a modality according to the chosen approach.

Dependence conflict: two solutions are proposed: remove the dependent rule (if there is equality between subjects and objects) or remove the common part from the dependent rule.

1 According to the adopted approach

If R1.s=R2.s and R1.o=R2.o and R1.op $\subset$ R2.op then
SP3 $\leftarrow$ {R2}

However, if the subject and the object of one rule are included in their corresponding in the other rule, then we propose to remove the common part from the dependent rule.

If R1.s $\supset$ R2.s and R1.o $\supset$R2.o and R1.op $\subset$ R2.op then
SP3 $\leftarrow$ {R1.s\R2.s and R2}

Cross Dependence conflict: the proposed solution is to remove the common part from the dependent rule.

If R1.s$\supset$ R2.s and R1.o$\subset$R2.o and R1.op $\subset$ R2.op then
SP3 $\leftarrow$ {R1.s\R2.s, R2}

For the second case of cross dependence:

If R1.s$\subset$R2.s and R1.o$\supset$ R2.o and R1.op $\subset$ R2.op then
SP3 $\leftarrow$ {R1.o\R2.o, R2}

6.3.2. Type conflict resolution

The second kind of conflict is "type conflict". Our resolution proposition is to give priority to the obligation type regard to request. In fact, we assume that obligation includes implicitly the authorization e.g. a subject that is required to update his password is implicitly authorized to achieve this update. This is expressed by the following precedence principle:

"Positive (respectively Negative) obligation overrides the negative (respectively positive) request"

Let's consider the following two rules:
R1: *req (teacher, change, password) → no*
R2: *ob (teacher, change, password, beginning- month)*
There is a type conflict because a teacher is obliged to change his password monthly by R2 but he isn't authorized to do this change by R1. Using the precedence principle proposed previously, the first rule is removed and the SP view will contain only *R2*.

6.3.3. Non conflicting rules combination

Since the proposed conflict resolution method is the combination; one must be able to deal with non conflicting rules belonging to the conflicting SPs.
The compound rule (noted R_{12}) of two not conflicting rules (let's say R_1 and R_2), is the union of the two rules given by:

$$R_{12} = R_1 \cup R_2.$$

For example, a rule disallowing access to files can be used in combination with a rule disallowing access to the network; the resulting rule disallows access to both files and the network.

6.3.4. SP combination completeness

Once, SP combination was presented, one must be sure that the obtained SP view preserves all the domain elements. Similarly to the completeness definition presented in [13], we propose the following:

DEFINITION 11 **(Completeness of SP combination)** *a SP combination is complete if any element of the sub-SP domain belongs after combination, to the compound domain. The completeness of SP = Φ (SP1… SPk) (where Φ is a combination of alternative rules) can be expressed by* $\;_{i}(dom(SP_i)) \subseteq dom(SP)$.

PROPERTY 1. *The SP combination by alternative rules is complete.*
PROOF. From Definition 1, a SP domain, *Dom (SP),* can be split in three sub domains at the most, *S*, *O* and *A*. In Definition 10, all elements of *Dom (SP)* are remapped by the rules defined

in Sub-Section 6-3-1, where each element is remapped to a new sub-domain. Therefore, the SP combination is complete.

7. Conclusion

Developing a SP is a sensitive task because the policy itself can lead to security weaknesses if it is not conform to the security needs of the organization. Hence, appropriate techniques are necessary to check whether a SP verifies the desired properties. These techniques, that should be the basis of a SP validation task, are unfortunately unavailable in the network security domain.
In this paper, we have proposed a SP modeling, specification and validation technique, inspired from the techniques used in software engineering and mainly based on the executable specification policy (ESP) and the reachability graph concepts. In fact, we introduced in this paper the ESP concept allowing the specification of the SP and the verification of its conformance with regard to the security needs. For the SP specification, we proposed S-Promela, a new executable language inspired from the well known, Promela. S-Promela syntax's is based on 2 components (channel and process) and 2 operations (read and write) where its semantics is based on Labelled Transition Systems. The validation process is then achieved via 3 steps. The first one performs the inconsistency detection by looking for SP rules leading contradictory decisions. The second step provides a completeness proof by the use of the reachability graph concept. Finally, the third step allows the proof of the preservation of security properties by. In this step, the security requirements are divided in two classes: safety and liveness. The safety properties proof is made using invariant generation and verification (by a model checker). The liveness properties proof is allowed if no cycle containing the opposite of the studied property is detected in the reachability graph. The infiniteness problem has been encountered for the reachability graph and has been resolved by the use of two hypotheses concerning the uniformity and regularity properties. The specific case of environment with multiple SP is also treated.

References

[1] Fraser, B. editor, (1997) RFC 2196, Site Security Handbook.

[2] R. Abbassi and S. Guemara El Fatmi, (2008) “A Model for Specification and Validation of Security Policies in Communication Networks: the firewall case”, In Proceedings of the Third International Conference on Availability, Reliability and Security, ARES 2008, pp. 467-473, Barcelona, Spain.

[3] R. Abbassi and S. Guemara El Fatmi, (2008) “An Automated Validation Method for Security Policies: the firewall case”, In Proceedings of the Fourth International Conference in Information and Security, IAS 2008, pp 291-294. Naples, Italy.

[4] R. Abbassi and S. Guemara El Fatmi, (2008) “Towards an Automated Firewall Security Policies Validation Process”, In Proceedings of the Third International Conference on Risks and Security of Internet and Systems, CRISIS 2008, pp. 267-272, Tozeur, Tunisia.

[5] Institute of Electrical and Electronic Engineers, IEEE (1993). Draft Guide for Information Technology, IEEE, 345 East 47th Street, New York, NY.

[6] G.J. Holzmann, (1991) "Design and Validation of Communication Protocols", Prentice Hall.

[7] P. A. Lindsay, (1997) "Specification and validation of a network security policy model", Tech.Rep. 97-05, Software Verification Research Centre, the University of Queensland.

[8] IEEE Guide to Software Requirements Specification. ANSI / IEEE Std 830.

[9] G.J. Holzmann and D. Pelled, (1994) “An Improvement in Formal Verification”, In Proceedings of FORTE 1994 Conference, Bern, Switzerland.

[10] M.C. Gaudel, (1995) "Testing can be formal too" TAPSOFT95, LNCS. Springer Verlag, Vol. 915, pp. 82-96.

[11] E. Al-Shaer and H. Hamed, (2004) “Discovery of Policy Anomalies in Distributed Firewalls.” In Proceedings of IEEE INFOCOM’2004.

[12] RFC 3198, (2001) “Terminology for Policy-Based Management”, IETF Request for Comments 3198.

[13] J. Dai and J. Alves-Foss, (2003) “A Formal Authorization Policy Model”. *Proc. Software Engineering Research & Applications (SERA '03)*.

[14] Hosner, H.H, (1993) “The Multipolicy Paradigm for Trusted Systems”, In Proceedings of the 1992-1993 workshop on New security paradigms, p.19-32.

[15] D. E. Bell and L. J. LaPadula, (1976) “Secure Computer Systems: Unified Exposition and Multics Interpretation”. Technical Report MTR-2997 Rev.1, MITRE Corporation, Bedford, Mass.

[16] E. Bertino, B. Catania, E. Ferrari, and P. Perlasca, (2003) “A LogicalFramework for Reasoning about Access Control Models”. ACM Transactions on Information and System Security.

Authors

Ryma Abassi is a PH.D candidate in the Higher School of Communication of Tunis and member of the ‘Communication Networks and Security’ research Lab. She received a Master’s Degree in Telecommunications in 2005. She is also working as a teacher in the Higher Institute of Technological Studies. Her research interests include network security, formal specification as well as validation and verification techniques.

Sihem Guemara EL Fatmi is a Professor in the Higher School of Communication of Tunis and member of the ‘Communication Networks and Security’ research Lab. She received her PH.D in Computer Science from the University Pierre and Marie Curie, Paris VI in 1983. Dr. Sihem Guemara El Fatmi has many journal and conference publications in the area of network security, formal specification, validation and verification techniques, quality of Service and optical Networks.