

Heterogeneous Security Policy Validation: From Formal to Executable Specifications

Jihène Krichène, Mohamed Hamdi, and Nouredine Boudriga
 Communication Networks and Security Research Lab., Carthage University, Tunisia
 Email: jkrichene@gmail.com, {mmh,nab}@supcom.rnu.tn

Abstract—This paper develops a prototyping technique for information systems security policies. Starting from the algebraic specification of a security policy, we derive an executable specification that represents a prototype of the actual policy. Executing the specification allows determining sequences of actions that lead to security policy violations. We propose a composition framework to build compound algebraic specifications. We show that the mechanism we provide to translate algebraic specifications to executable specifications preserves the composition rules, which is of utmost importance from the engineering perspective. Through accurate examples, we show how executables specifications can be used in conjunction with formal specification in the frame of the security policy engineering process.

Index Terms—Algebraic specifications, executable specifications, security policy engineering.

I. INTRODUCTION

Many security problems are directly or indirectly related in security-critical components of the information system. Attackers may exploit such vulnerabilities to gain unauthorized access or to exceed their privileges. Tests and verification is an alternative approach to detect information system flaws besides vulnerability detection. In fact, vulnerability assessment tools (e.g., Nessus, SARA, Nikto) show a great potential in identifying technological weaknesses, which may exist in operating systems, software applications, and networking nodes. However, these tools fail in detecting vulnerabilities in the security policy, which are by far more critical.

The security policy can be defined as a set of rules that determine how a particular set of assets should be secured. It appears as a multi-faceted concept that can be implemented in different components of the information system. In the literature, many techniques have been proposed to specify and validate security policies. The main advantage of formally representing a security policy is to discard some relatively insignificant details. Furthermore, mathematical modeling allows, through the application of a sequence of abstraction levels, to build a set of views that represent the system in a manner that is increasingly close to the reality. Generally, a formal modeling framework consists of: (a) a set of entities that represent the elements of the information system; (b) a logic allowing to build clauses and formulas; (c) a set of axioms defining the main properties of the system; and (d) a set of deduction rules that show how logical formulas can be inferred one from the others.

Even though algebraic specifications come up with powerful tools for syntactic validation, they do not intrinsically encompass semantic verification techniques. In other terms, they cannot be used by the security engineer to state whether the *behavior* of the security policy is correct with regard to the security objectives. Recognizing these difficulties, the designer of an information system that has security requirements is well advised to develop an executable form of the security policy in order to check whether the security properties are fulfilled by this policy.

This paper proposes a security policy design technique based on executable specifications. From an algebraic specification of a security policy, we extract an executable policy that can be manipulated as if it were actually implemented. While formal specifications allow detecting inconsistencies in the security policy (e.g., consistency, completeness), executable specifications allow to identify the potential actions an intruder would carry out to attack the system. To build executable specifications, we use a logic called S-TLA⁺. This logic is based on actions that can be used to state what must be done if any violation of the policy is detected. Moreover, it is accompanied by a model checker allowing the automated execution of S-TLA⁺ specifications. The major contributions of the paper are:

- 1) Definition of a new concept, called executable security specification, to verify a security policy through the use of model checking techniques
- 2) Definition of a composition framework allowing to build compound security policies using elementary policies
- 3) Definition of rewriting rules to transform algebraic specifications into executable S-TLA⁺ specifications
- 4) Introduction of a heterogeneous proof system encompassing both formal and executable specifications to build consistent, complete, and correct security policies

The rest of the paper is organized as follows. Section II reviews the approaches proposed in the literature to develop algebraic and executable specifications. We highlight the complementarity of these two techniques. Section III sets a novel security engineering process based on heterogeneous validation using formal and executable specifications. An algebraic framework to formally express elementary and compound security policies is dis-

cussed in Section IV. Section V introduces a behavioral technique to build executable security policy specifications. A deduction system allowing the conduction of formal proofs of security policy correctness is presented in Section VI. Two case studies, related to physical and logical access control, are given in Section VII to illustrate the formal concepts developed throughout the paper. Finally, Section VIII concludes the paper.

II. RELATED WORK

The state of the art of verification and validation has shown that formal methods, which have been thoroughly used during last three decades for verification and validation purposes, suffer from limited commercial success due to several factors including their complexity and the high level of mathematical skills needed for effective use [1]. Recent research has focused on executable specifications as a new technique for verification purpose with a notable simplicity and not suffering from the complexity limitations of classic formal methods. [1] defines executable specifications as follows:

“a new class of applications of formal specifications whereby specification rules are executed on a computer much like any high-level programming language”

More precisely, [2] defines an executable specification as “a system model that can generate the behavior of the target system interacting with its environment”.

[3], [4], and [1] stress the importance of executable specifications as they are used to validate specifications against informal requirements. This early validation increases the correctness and reliability of the software to be developed and allows to save a lot of time and money in later phases of the software development life-cycle. Executable specifications are not reduced to conceptual models, they also represent a behavioral model of future software systems and thus they are used for testing if implementations will satisfy or not the specifications.

Executable security policies play a relevant role in validation and verification of security policies like the executable specifications do within the software engineering field. In fact, they allow to save time and money as they help in checking the conformity to the security needs at early phases and before implementing security solutions. This avoids the performance of many attacks against the information systems which take advantage of the absence of suitable security rules and solutions. Early validation will also provide the purchase of non-appropriate security solutions that are very expensive.

Furthermore, executable security policies allow automated verification of implemented security solutions. This avoids the omission of certain scenarios that may cause harmful damages to the information system.

Despite their similarities, security policies present some differences with software specifications. The main difference consists in the fact that specifications are concerned as a process (following a modeling method such as UML) and a product at the same time, while a security policy is

just a product. The remaining differences are summarized in the following issues:

- **Structure:** Specifications define a set of variables (inputs and outputs) and the different methods that will constitute a single software product. However, security policies delimit the boundary of the system to be secured and the actions that are considered in this system. A network analysis is then performed to define the objects and subjects in the information system and the different rules they must abide. It comes without saying that a security policy deals with a set of products. This is due to the heterogeneity of the rules constituting the policy documents.
- **Nature:** Software specifications define methods that can be seen as a set of instructions having in input some variables and providing results or outputs depending on the given inputs. Hence, software programs are considered as a set of serial or parallel instructions. However, in security policies we deal with rules to be applied and security properties to be respected and controlled. Security rules specify what must be done within the objects and roles situated around the delimited boundary. Therefore, security policy can be seen as continuous set of programs since it depends on the state of the system to be secured.
- **Types of variables, objects and subjects:** Variables manipulated in software specifications are essentially predefined types such as integers, reals, floats and strings or constructed types that are composed of the previous ones and types used in graphical interfaces such as buttons, and events. While objects and subjects manipulated in security policies are elements of the information system. They are essentially: equipments such as work stations, servers, routers, firewalls and switches, locals, documents such as security or technical or administrative documents; and roles such as technical or administrative personnel, visitors and trainee persons. These lists are not exhaustive. In fact, equipments list may grow as new equipments arise. Documents and roles may also differ from one organization to an other according to the nature of their activities.
- **Temporal considerations:** Unlike software specifications, security policies do not require a refined representation of the concept of time. Time is often used either to assess the average occurrence of an event or to express the maximum tolerated time separating the occurrence of the attack from the implementation of the security solution. In fact, some rules include temporal expressions such that before, after, and periodically. For example, a security rule states that a backup must be performed periodically.
- **Decisions type during the design phase:** Unlike software specifications where a set of instructions are to be implemented, security policies involve complex decisions. In fact, within information systems, we have to decide about the limit between the secured

and non-secured areas. We have also to make the right decisions about where to implement security solutions to fulfill security needs. Moreover, decisions in security policies are evolutive by nature due to the constant evolution of network attacks.

III. SECURITY POLICY ENGINEERING USING EXECUTABLE SPECIFICATIONS

For many years it has been common to consider the security policy as a document specifying the security objectives and countermeasures. In our sense, the security policy is a 'product' that should be subjected to traditional engineering activities (i.e., specification, validation, testing). A set of mathematical tools used to specify the policy, assess it with respect to a set of requirements, and test the implementation of the security policy.

Enhancing security policies is highly required since security in modern network systems is becoming more and more important. Furthermore, security policies have to be enhanced after new attacks/vulnerabilities arise or after the creation of new services in the organization. The consideration of new services generally leads to the modification of the security properties making inappropriate the current policy. We develop a seven-step multi-loop methodology for improving security policies based on the concept of executable security policy specification introduced above.

- 1) *Definition of security policy properties and criteria:* In this step, the security policy and properties are expressed in natural language. A set of criteria is added to the current policy. These criteria are the bases of choosing the best enhanced security policy. Examples of criteria are the number of properties that are violated, the losses owed to the the violation of a/some security property/ies, and the cost of security solutions implementing the security policy.
- 2) *Algebraic specification:* This step consists in translating the security policy properties and criteria to a mathematical language. The algebraic form of the security policy should include a description of the assets to be protected, a list of the actions that can be performed on these assets, and a set of security rules that should be satisfied by the actions.
- 3) *Security policy validation:* In this step, the algebraic specification of the security policy is analyzed for syntactic verification purpose. Any detected syntactic error will lead to the modification of the security policy and its algebraic specification. In addition, formal semantic verification is performed to detect inconsistencies (i.e., conflicting rules) or incompleteness (i.e., missing rules). Any modification to the security policy triggers steps 1 and 2.
- 4) *Executable specification generation:* This step consists in translating the syntactic error-free algebraic specification of the security policy in an executable language, called S-TLA+. The translation activity follows a set of rules that have to present some

properties (i.e. completeness, confluence, termination, soundness).

- 5) *Executable specification checking:* The executable policy is run at this level in order to detect vulnerabilities in the security policy with respect to the properties and criteria defined in step 1. Detected semantic errors lead to the modification of the security policy. Hence, steps 1 to 4 must be re-applied to prove the syntactic and the semantic correctness of the updated policy after the analysis of the detected errors.
- 6) *Implementation:* This step enforces the application of the security policy. During this step, operational and technical controls are put in place. Operational controls are security mechanisms that are essentially implemented and executed by the users themselves while technical controls include the automated security countermeasures.
- 7) *Implementation testing:* Implemented security policy must be tested in order to guarantee its effectiveness. A set of test cases must be generated and applied to prove that the behavior of the security solutions is conform to what has been expected according to the properties and criteria built in step 1. Test case generation is based on the algebraic specification of the security policy. This process is based on our test case generation approach introduced in [5].

The reader may notice that , within the aforementioned engineering process, three different techniques are used to assess the security policy. First, the algebraic specification is checked using formal tools such as theorem proving. Second, the executable form of the security policy is run to determine the potential attack scenarios that would lead to security properties violation. Third, the security policy implementation is tested in order to limit the technological flaws.

One important remark is that the first verification phase is a kind of white-box testing since the validation is done on the content of the security policy specification. However, the two remaining assessment steps can be rather considered as black-box testing techniques since external actions are submitted to the security policy in order to state whether it conforms to the requirements or not based on the results of this execution. In [], the authors developed an algebraic tool for testing the implementation of a security policy. In the following, we show how executable specifications can be used to perform the two first validation steps.

IV. A COMPOSITIONAL FRAMEWORK FOR ALGEBRAIC SECURITY POLICY SPECIFICATION

This section presents algebraic specifications as a fundamental tool to carry out the engineering process discussed in the previous section. We show how the structure of traditional algebraic specifications can be used to build a mathematical form of the security policy. In addition, we develop a set of algebraic operators to build compound

security policies using elementary policies. This may be particularly helpful, from the engineering point of view, for a gradual construction of the formal security policy.

A. Algebraic security policy specifications

As it has been mentioned above, security policies are used to handle the interaction between a set of subjects and a set objects in a complex sensitive environment. Therefore, a security policy should convey sufficient information about system entities, operations, and rules. Developing a framework that allows to model generic security policies should necessarily consider these three components. Entities refer to the elements constituting the system of interest. For instance, an entity can be, depending on the application, a human user, a computer, or a process. Operations are used to represent the actions that can be performed by the different entities; they range from physical access (for a human being) to network connections (for networked computers). Operations slightly differ from actions in the sense that they are more abstract with regard to the system entities. For example, if $connect(.,.,.,.)$ is an operation representing the connection establishment between two machines, then $connect(ip_1, ip_2, p_1, p_2)$ is an action that represents the connection establishment between two specific hosts (i.e., ip_1, ip_2) using source and destination ports (i.e., p_1, p_2). Security rules define the constraints that allow differentiating between legitimate and prohibited actions. Many-sorted signatures can be used to manipulate the previous sets as they are commonly used to handle 'typed' data values [6]. In [7], we have used algebraic specifications to model network security policies. An algebraic security policy is expressed by the triplet $\pi = \langle \epsilon, \omega, \rho \rangle$ where:

- ϵ is a set representing the elements of the protected system,
- ω is a set of operations that can change the system state,
- and ρ is a set of rules that allow to define the secure and the insecure actions that can be performed on the system. It is noteworthy that a rule can be either positive or negative, meaning it can allow the action or deny it. In other terms, $\rho = \rho^+ \cup \rho^-$, where $\rho^+ = \{ \bigwedge_i t_i \Rightarrow t, t_i, t \in |T_\pi(X)| \}$ and $X \subseteq \mathfrak{Var}(\epsilon)$ and $\rho^- = \{ \bigwedge_i t_i \Rightarrow t, t_i, t \in |T_\pi(X)| \}$.

An illustrative example is given in Figure 1 where we propose a signature π_1 assigning security levels to the roles, rooms, and equipments.

Four sets are given in this restricted example, *Roles*, *Rooms*, *Objets*, and *int* that represent information system roles, rooms, objects, and their security levels. Furthermore, an operation, i.e. *level* is used to formally assign security levels to the roles, rooms and objects.

This signature is characterized by the presence of three axioms that express the security rules considered in the studied physical security policy. Three rules, expressed respectively by axioms φ_1, φ_2 , and φ_3 in the signature

π_1 , are given in this example. A brief explanation of these rules is given in the following:

- 1) Any object can be localized in a given room only if its security level is less than or equal to the room's level.
- 2) Any user can access a given room only if his security level is greater than or equal to the room's level.
- 3) A physical access of a user to an object is granted only if he and the object are both localized in the same room and if the user has at least the same security level as the accessed object.

B. Security policy composition

We have pointed out in the previous discussion that a security policy contains generally more than one component addressing different security issues in the secured system. The algebraic security policy framework should include various composition rules. Effectively, it should support the combination of policies expressed in different languages. This allows using heterogenous security mechanisms to achieve the security objectives. Even though a single abstract representation of such a unified policy can be reached, developing implementable policies for all of these mechanisms would be unfeasible. Hence, formal representation is the unique context the soundness and the consistency of the compound security policy can be assessed. The main composition rules that we consider in this paper are given in the following:

- Addition ($\pi_1 + \pi_2$): According to this combination principle, an action is allowed if at least one of the two policies grants it. It is also prohibited if at least one of the two policies denies it.
- Product ($\pi_1 * \pi_2$): In this case, the set of rules related to the compound security policy is, in some sense, the intersection of the elementary rule sets. This means that an operation is allowed (resp. denied) if both of of rules sets allow (resp. deny) it.
- Substraction ($\pi_1 - \pi_2$): This composition returns the intersection between the positive rules of the first policy and the negative rules of the second policy. In other terms, the compound policy allows an action if it is granted by the first policy and prohibits it if the second policy interdicts it.

An important property to be highlighted at this level is that these operators are coherent with respect to the implementation testing strategy. In fact, the test case set for the sum (resp. product) of two algebraic security policies is the union (resp. intersection) of the test case sets corresponding to the elementary policies. Similarly, the test case set for a policy $\pi_1 - \pi_2$ is the union of the test cases generated by the positive rules of π_1 and the negative rules of π_2 . The formal expressions of these expression rules are given in the following.

This feature is extremely interesting since it allows not only a progressive construction of the security but also a gradual testing of the policy implementations. In

π_1	Spec	Sorts	<i>Roles, Rooms, Objects, int</i>
		Opns	<i>level : Rooms \cup Roles \cup Objects \longrightarrow int</i> <i>location : Roles \cup Objects \longrightarrow Rooms</i> <i>use : Roles \longrightarrow Objects</i>
	Axioms	φ_1	$\forall o : Objects. r : Rooms. l1, l2 : int. level(o) = l1 \wedge level(r) = l2 \wedge l1 \leq l2 \implies location(o) = r$
		φ_2	$\forall u : Roles. r : Rooms. l1, l2 : int. level(u) = l1 \wedge level(r) = l2 \wedge l1 \geq l2 \implies location(u) = r$
		φ_3	$\forall u : Roles. o : Objects. l1, l2 : int. level(u) = l1 \wedge level(o) = l2 \wedge location(u) = location(o) \wedge l1 \geq l2 \implies use(r) = o$

Figure 1. An example of security rules defined using algebraic specifications

$\pi_1 + \pi_2 =$	$\langle \epsilon_1 \cup \epsilon_2, \omega_1 \cup \omega_2, \rho_1 \cup \rho_2 \rangle$
$\pi_1 * \pi_2 =$	$\langle \epsilon_1 \cup \epsilon_2, \omega_1 \cup \omega_2, \rho_1 \cap \rho_2 \rangle$
$\pi_1 - \pi_2 =$	$\langle \epsilon_1 \cup \epsilon_2, \omega_1 \cup \omega_2, \rho_1^+ \cup \rho_2^- \rangle$

the sequel, we give a set of mathematical rules allowing progressive verification of executable security policies.

In the previous sub-section, we have presented a physical security policy. Let us consider here a second example of security policies dealing with access rights of the roles within the same information system. Two rules are considered in this policy. Their algebraic representation is depicted by Figure 2. These rules state that each role has three access rights (i.e. *read(r)*, *write(w)* and *execute(x)*) on a given data stored on a given object, and that the administration data stored on the administrator’s machine are restricted to the administrator. The absence of a given right will be replaced by the character (*n*).

Several operations may be applied to composed policies i.e. union and intersection. Intersection between two security policies infers that an action is granted only if it is granted by both of the two policies. In this example we will use the intersection of the two policies and study the effect of this operation on the information system security.

Notice that in the physical policy, the role *user* can get access to data stored on the administrator’s machine. In fact, *user* can access physically to the administrator’s machine since he has access to the room where the machine is located as stated by the axiom φ_3 . This example supposes the existence of two rooms: the first room holds server machines to which we assign the highest security level and the second room holds workstations to which we assign a less important security level. To the first room we assign the highest security level and to the second room we assign a lower security level (but not the minimal one). Administrator and user are the main roles in this example. The role of administrator has the highest security level, while the role of user has a less important security level. Here, we extend this policy by adding a security rule stating that the role administrator can establish a remote access between two objects to which he has physical access for reading or writing information purposes. This rule is algebraically specified in signature Φ_3 by axiom φ_4 . Figure 3 depicts the resulting specification.

According to the axiom φ_4 , the administrator can establish remote access to the servers located in the administration room. Consequently, the role *user* can read information originated from the servers.

Let us now consider the product of the physical security policy with the access rights policy. The latter states that access to data stored on the servers is only granted to the administrator role as given by axiom φ . Hence, the role *user* has (*n, n, n*) to any data originally stored on the servers. So, even if the physical security policy has allowed the user to access to the server’s data, the access rights policy has denied any access to these data.

V. LOGIC-BASED EXECUTABLE SPECIFICATIONS

Once the algebraic security policy specification has been validated, we develop an executable form of the policy in order to carry out black-box testing based on the execution of policy using different input actions. The objective is to find an action sequence that leads to a violation of the security requirements corresponding to the security policy. To achieve this objective, we use a logic called S-TLA⁺ (Security-Temporal Logic of Actions) that was introduced in []. We first review the main features of this logic. Then, we extend S-TLA⁺ as well as the corresponding model checker so as to support security policy engineering functionalities.

A. S-TLA⁺ basics

S-TLA⁺ is an extension to TLA⁺ introduced in [8] where the concept of hypothesis is used to help reasoning with uncertainty. TLA⁺ has been defined in [9] for the specification of reactive, distributed and asynchronous systems.

In the following, the components of S-TLA⁺ which show a particular interest, from the security policy representation point of view, are highlighted:

Logic predicates: SPs enumerate the security rules within the ISs to be secured. Security rules can be seen as logical predicates as they state the actions that must be performed if some conditions are present in the system.

Active actions: SPs involve active actions stating for instance what must be done if any violation of the policy is detected.

π_2	Spec	Sorts	<i>Roles, Rooms, Objects, int, Rights, Data</i>
		Opns	$r : \rightarrow Rights$ $w : \rightarrow Rights$ $x : \rightarrow Rights$ $n : \rightarrow Rights$ $level : Rooms \cup Roles \cup Objects \rightarrow int$ $stored : Data \rightarrow Objects$ $right : Roles \times Data \rightarrow Rights \times Rights \times Rights$ $max : int$
		Preds	
	Axioms	φ	$\forall r : Roles. d : Data. l1, l2 : int. o : Objects. stored(d) = o \wedge level(o) = l1 \wedge max(l1) \wedge level(r) = l2 \wedge max(l2) \implies right(r, d) = (r, w, x) \vee right(r, d) = (r, n, n) \vee right(r, d) = (r, w, n) \vee right(r, d) = (r, n, x) \vee right(r, d) = (n, w, x) \vee right(r, d) = (n, w, n) \vee right(r, d) = (n, n, x)$

Figure 2. An example of using algebraic specifications to define access rights policy

π_3	Spec	Sorts	<i>Roles, Rooms, Objects, int</i>
		Opns	$level : Rooms \cup Roles \cup Objects \rightarrow int$ $location : Roles \cup Objects \rightarrow Rooms$ $use : Roles \rightarrow Objects$ $rmtaccess : Objects \rightarrow Objects$ $max : int$
		Preds	
	Axioms	φ_1	$\forall o : Objects. r : Rooms. l1, l2 : int. level(o) = l1 \wedge level(r) = l2 \wedge l1 \leq l2 \implies location(o) = r$
		φ_2	$\forall u : Roles. r : Rooms. l1, l2 : int. level(u) = l1 \wedge level(r) = l2 \wedge l1 \geq l2 \implies location(u) = r$
		φ_3	$\forall u : Roles. o : Objects. l1, l2 : int. level(u) = l1 \wedge level(o) = l2 \vee o : Objects. location(u) = location(o) \wedge l1 \geq l2 \implies use(r) = o$
		φ_4	$\forall o1, o2 : Objects. a : Roles. l : int. level(a) = l \wedge max(l) \wedge use(a) = o1 \wedge use(a) = o2 \implies rmtaccess(o1) = o2 \vee rmtaccess(o2) = o1$

Figure 3. Extended physical access control security policy where a remote access rule is added.

Invariants: SPs must conform with some properties expressed by the system owner. Security properties can be seen as invariants in S-TLA⁺ context.

Temporal logic: SPs involve temporal expressions such as for, before and after. For instance, a rule states that “If workstations are idle for a period of time, then they will eventually be locked”. Temporal logic is then required when specifying SPs.

Distribution concept: Nowadays, organizations cooperate via networked systems. Assuming that each organization has its own SP, properties and solutions, the communication with the other organizations complicates the validation of security properties within each organization.

Hypotheses: The enhancement of SPs supposes the omission and addition of security rules to check which combination of rules is more conform to the properties. Hypothesis introduced in S-TLA⁺ help manipulating rules for comparison and enhancement purposes.

B. Representing security policies using S-TLA⁺

It comes from the foregoing subsection that S-TLA⁺ is appropriate for specifying security policies. In the following, we show how security policies can be represented

using S-TLA⁺ and give an example of a SP specified using this language. An executable S-TLA⁺ SP can be generically expressed using the following components:

- Constants = $\{c_i, i = 1, \dots, n_c\}$,
- Variables = $\{v_i, i = 1, \dots, n_v\}$,
- TypeInvariant = $\bigwedge_{i=1, \dots, n_t} T_i$ (where $T_i = [\bigcup_{j=1, \dots, n_{T_i}^1} c_j \rightarrow \bigcup_{k=1, \dots, n_{T_i}^2} c_j]$),
- Actions, representing predicate-based logical formulae. Informally, the T_i can be viewed as assertions that globally determine the behavior of the variables.

Moreover, we take advantage from the flexibility of S-TLA⁺ to write SPs libraries in form of S-TLA⁺ modules. We define SPs libraries to first hold the definition of the different entities constituting SPs and check their enforcement. Figure 4 presents a library that defines the structure of a log file and delineates some operators to manipulate its content in accordance with a given SP. The content of such library is described as follows:

- *Logs(C, s)*: The set of all finite log files containing s entries and holding data of type C . As S-TLA⁺ is based on a mathematical foundation, we represent the log file using a function from the domain of supported entries numbers to the domain of content C , including empty record $\langle \rangle$.

- *EmptyLog(s)*: Represents an empty log file that holds s records $\langle \rangle$.
- *Len(l)*: The length of a log file. Such value is returned by looking for the highest entry number whose content is non empty.
- *Append(l, e)*: The log file l obtained after appending a record e to its end (tail). Two cases can be distinguished. In the former, the log file l has enough space to hold e . The first empty record, starting from the beginning of the file, is therefore replaced by the new record e . In the latter case, the log file is full of data, and its content is then shifted, the oldest record is removed, and the new record e is written to the tail of the log l .

This library can be used for instance by a SP governing physical access to information system assets and facilities. In this example, it is important to store assets use in log files specified here.

C. Adding operators to S-TLA⁺

All the components and the rules that constitute SPs must apply simultaneously on the secured system. S-TLA⁺ logic used in the previous examples does not support this concept. In fact, when faced to more than one action that can be applied, S-TLA⁺ makes a random choice for executing one of them. The choice can not be random when dealing with composed security rules. In addition, product and substraction introduced in Section 2 govern the way two or more candidate actions are manipulated. Moreover, security rules are related to the environment to which they are applied. It is sometimes required to wait for the termination of a given action on the system before starting a second action or applying a given security rule. S-TLA⁺ does not hold an operator expressing the termination of an action. In fact, it implicitly considers this concept via the use of a variable in the first action. The output value of this variable represents an input condition to the second action.

We introduce in this section three operators helping reasoning about simultaneous appliance of security rules and supporting the concept of action termination: addition denoted by (+), product denoted by (*) and substraction denoted by (-). These operators are added to the \vee operator generally used in the *Next* section of a S-TLA⁺ program.

The + operator: Suppose that two actions $A1$ and $A2$ are considered in this program. If $A1$ and $A2$ are combined using the addition, the $A1 + A2$ expression states the following:

- 1) If $A1$ modifies a subset $S1$ of TypeInvariant and $A2$ modifies a second subset $S2$ such that $S1 \cap S2 = \{\}$ (informally they do not infer the same modifications to the system), then the execution of $(A1 + A2)$ would be equivalent to the execution of either $A1$ or $A2$ (expressed by the action $A1 \vee A2$).
- 2) If $A1$ modifies a subset $S1$ of TypeInvariant and $A2$ modifies a second subset $S2$ such that $S1 \cap S2 \neq \{\}$ (informally they infer in some way the same

modifications to the system), then we apply the action that involves the minimal set of predicates.

For example, if $A1 (P1 \wedge P2 \wedge P3 \implies allow)$ and $A2 (P1 \wedge P2 \wedge P3 \wedge P4 \implies allow)$ are two security rules modifying respectively the subsets of TypeInvariant $\{s1, s2, s3\}$ and $\{s2, s4, s5\}$, then $A1 + A2 \equiv A1$.

The * operator: Suppose that two actions $A1$ and $A2$ are considered in this program. If $A1$ and $A2$ are combined using the product, the $A1 * A2$ expression states the following:

- 1) If $A1$ modifies a subset $S1$ of TypeInvariant and $A2$ modifies a second subset $S2$ such that $S1 \cap S2 = \{\}$, then the execution of $(A1 * A2)$ would be equivalent to the execution of both $A1$ and $A2$ (expressed by the action $A1 \Delta A2$).
- 2) If $A1$ modifies a subset $S1$ of TypeInvariant and $A2$ modifies a second subset $S2$ such that $S1 \cap S2 \neq \{\}$, then we apply the action that involves the maximal set of predicates.

For example, if $A1 (P1 \wedge P2 \wedge P3 \implies allow)$ and $A2 (P1 \wedge P2 \wedge P3 \wedge P4 \implies allow)$ are two security rules modifying respectively the subsets of TypeInvariant $\{s1, s2, s3\}$ and $\{s2, s4, s5\}$, then $A1 * A2 \equiv A2$.

The fin operator: This operator is used to express the termination of an action. To this purpose, we associate a flag to each action which indicates the execution state of an action. This flag will be taken in consideration if the *fin* operator is used. The flag is defined to be either red or green. A red flag indicates that the action is being executed while a green flag states that the action has terminated. For example, if $A1$ and $A2$ are two actions, then $A1 \vee fin(A2)$ states that the action $A1$ is executed if and only if the flag of the action $A2$ is green. The distribution of the *fin* operator on the \vee and Δ operators respects the following rules:

$$fin(A1 \vee A2) \implies fin(A1) \vee fin(A2)$$

$$fin(A1 \Delta A2) \equiv fin(A1) \Delta fin(A2)$$

Finally, it can be noticed that, in this paper, we have not developed an operator in S-TLA⁺ corresponding to security policy substraction.

D. S-TLC extension

S-TLA⁺ came up with a model checker, called S-TLC allowing various forms of action execution. In S-TLC, a state can be represented in the generated graph as a valuation of all its variables including the constrained ones. Node representation involves two notions: *node core* and *node label*. The core of a node represents a valuation of the entire non-constrained variables, and the node label represents the potential sets of hypotheses (a set of hypotheses is a valuation of the entire constrained variables) under which the node core is reached. The S-TLC algorithm employs three data structures \mathcal{G} , \mathcal{U}_F and \mathcal{U}_B . The first refers to the reachability directed graph under construction generated during forward chaining and backward chaining phase. The last two are FIFO queues, containing states whose successors have not being

MODULE *Logs*LOCAL INSTANCE *Naturals*LOCAL INSTANCE *TLC* $Log(C, s) \triangleq$ The set of all finite log files containing s entries and holding data of type C $[1 .. s \rightarrow C \cup \{\langle \rangle\}]$ $EmptyLog(s) \triangleq$ An empty log file that contains s entries $[i \in (1 .. s) \mapsto \langle \rangle]$ $Len(l) \triangleq$

The length of the log file

IF $(\forall x \in \text{DOMAIN}(l) : l[x] \neq \langle \rangle)$ THEN $(\text{CHOOSE } z \in (\text{DOMAIN } l) : \forall y \in (\text{DOMAIN } l) : z \geq y)$ ELSE IF $l[1] = \langle \rangle$ THEN 0 ELSE $\text{CHOOSE } n \in \text{DOMAIN}(l) :$ $\forall v \in \{m \in \text{DOMAIN}(l) : l[m] \neq \langle \rangle\} : \wedge n \geq v$
 $\wedge l[n] \neq \langle \rangle$ $Append(l, e) \triangleq$ The log file l obtained after appending a record e to its endLET $z \triangleq \text{CHOOSE } w \in \text{DOMAIN}(l) : \forall y \in \{v \in \text{DOMAIN}(l) : l[v] = \langle \rangle\} :$ $\wedge w \leq y$ $\wedge l[w] = \langle \rangle$

IN

IF $(\forall x \in \text{DOMAIN}(l) : l[x] \neq \langle \rangle)$ THEN $[i \in \text{DOMAIN}(l) \mapsto \text{IF } (i + 1 \in \text{DOMAIN}(l)) \text{ THEN } l[i + 1] \text{ ELSE } e]$ ELSE $[i \in \text{DOMAIN}(l) \mapsto \text{IF } (i = z) \text{ THEN } e \text{ ELSE } l[i]]$ $DoContain(l, m) \triangleq$ True if the log file contains the record m $\exists x \in \text{DOMAIN}(l) : l[x] = m$ Figure 4. Example of a library module written in S-TLA⁺ syntax.

yet computed respectively during forward and backward chaining phases. The model checking process using S-TLC includes two phases:

- 1) Forward chaining: All the scenarios that originate from the set of initial system states are inferred in a forward chaining manner. This involves the generation of new sets of hypotheses and evidences that are consequent to these scenarios.
- 2) Backward chaining: During this phase, the algorithm starts with a queue \mathcal{U}_B holding the set of terminal states; the states that satisfied the termination condition in forward chaining phase. Afterwards, and until the queue becomes empty, the tail of \mathcal{U}_B , described by state t , is retrieved and its predecessor states (the set of states s_i such that the pair of states (s_i, t) satisfies the S-TLA⁺ action $Next$) are computed.

Of course, the modifications introduced in S-TLA⁺ require adequate modifications of S-TLC. The modifications introduced to S-TLC do not concern the initialization or chaining phases while constructing scenarios with the model checker, but the construction of the nodes that will be added to the graph. A node represents a system

state. It is constructed using the actions within S-TLA⁺. Generally, S-TLA⁺ chooses an action and uses it to determine the next state(s). In our case, the next state(s) calculation depends on the operator used to combine actions.

If the $+$ operator is used, S-TLC will check if the combined actions share a non empty subset of TypeInvariant. Two cases are considered at this level: (a) if yes, S-TLC proceeds as in the general case, and (b) if no, S-TLC looks at the predicate list of each action and chooses the action that involves the minimal number of predicates to calculate the next state(s).

If the $*$ operator is used, S-TLC checks if the combined actions share a non empty subset of TypeInvariant. Two cases are considered at this level: (a) if yes, S-TLC uses the two actions to calculate the new state(s), and (b) if no, S-TLC looks at the predicate list of each action and chooses the action that involves the maximal number of predicates to calculate the next state(s).

VI. HETEROGENEOUS SECURITY POLICY VALIDATION

In [7] we have specified security policies using algebraic specification. The latter provides a solid basis

for reasoning about security policies; i.e. extracting test cases for testing the correctness of security policies. In this paper, we use security policy algebraic specifications for the purpose of semantic validation. In fact, from the formal representation, an executable program is derived in order to validate the security rules.

A. From algebraic to executable specifications

To derive the ESP from the algebraic specification, we use a rewriting system. We define the binary relation $\rightsquigarrow \subseteq \Pi \times \Xi$ which derives an executable specification from an algebraic, where Π is the set of algebraic security policy specifications and Ξ is the set of executable security policy specifications. This means that the executable policy ξ is the result of applying rewriting rules to the formal policy π . Obviously, these rules will depend on the language used to express executable security policies.

However, independently of the ESP language, we have to prove the correctness of the obtained executable specifications.

a) *Termination*: Any computation sequence ends up in some element which can not be further rewritten.

b) *Confluence*: The choice of which term to rewrite in an expression does not affect the result of the computation.

c) *Soundness*: There is no result that the transformed program returns but that the initial program can not return.

d) *Completeness*: Any result returned by the initial program can also be returned by the transformed program.

Automated conversion of algebraic SPs into S-TLA⁺ policies is rendered possible via a rewriting system. For instance, a rewriting system can be expressed by three relations $\rightsquigarrow = \{\rightsquigarrow_\epsilon, \rightsquigarrow_\omega, \rightsquigarrow_\rho\}$ where $\rightsquigarrow_\epsilon \subseteq (\epsilon \times \text{CONSTANTS})$, $\rightsquigarrow_\omega \subseteq (\omega \times \bigcup_{c_i \in \text{CONSTANTS}} [c_1, \dots, c_n \rightarrow c_0])$, and $\rightsquigarrow_\rho \subseteq (\rho \times \text{ACTIONS})$. Within this context, the rewriting system should possess the following property:

$$(\omega = \epsilon_1, \dots, \epsilon_n \rightarrow \epsilon_0, [c_1, \dots, c_n \rightarrow_\omega c_0]) \in \rightsquigarrow \text{iff. } (\epsilon_i, c_i) \in \rightsquigarrow_{\epsilon} \text{ for } 1 \leq i \leq n.$$

Now, we describe the execution of the ESP $\xi \in \Xi$. To this end, we give here some judgements and inference rules. In the following, Σ denotes the set of network states. Our goal is to prove that the execution of ξ terminates and returns a network state $\sigma_f \in \Sigma$. To start, we define the relation describing the execution of any rule in the policy:

$$\varepsilon \subseteq \Xi \times \Sigma \times \Sigma.$$

Then, the triple $(\xi, \sigma_i, \sigma_f) \in \varepsilon$ expresses that given an initial network state σ_i , the execution of the policy ξ returns the final state σ_f .

We also introduce two relational operators $\hat{\varepsilon}(\cdot, \cdot)$ and $\check{\varepsilon}(\cdot, \cdot)$ defined as follows:

$$\hat{\varepsilon}(\xi, \sigma) = \{\sigma' \in \Sigma \mid (\xi, \sigma', \sigma) \in \varepsilon\},$$

$$\check{\varepsilon}(\xi, \sigma) = \{\sigma' \in \Sigma \mid (\xi, \sigma, \sigma') \in \varepsilon\}.$$

Informally, $\hat{\varepsilon}(\xi, \sigma)$ denotes the set of initial states that lead to the final state σ when ξ is executed. Similarly, $\check{\varepsilon}(\xi, \sigma)$ refers to the set of final states obtained through the execution of ξ on the final state σ .

B. Proving the correctness of formal security policies

At this stage, we have transformed a formal security policy, π , into an executable one, ξ . We have also proved, using the S-TLC model checker, that the execution of ξ terminates and returns the network state σ_f . The objective is to prove the correctness of an algebraic security policy π using the corresponding executable specification ξ . To this purpose, we divide the set of network states Σ into two subsets Σ_S and Σ_N referring to the sets of secure and non-secure states; respectively. Obviously, these subsets should satisfy $\Sigma_S \uplus \Sigma_N = \Sigma$. The following axiom expresses the conditions for a security policy π to be correct.

$$(\alpha) \frac{(\xi, \pi) \in \rightsquigarrow, (\forall \sigma_i \in \Sigma : (\xi, \sigma_i, \sigma_f) \in \varepsilon \Rightarrow \sigma_f \in \Sigma_S)}{\iota(\pi)},$$

where $\iota(\cdot)$ is a predicate symbol indicating whether an algebraic security policy is correct.

This means that a security policy π is correct if the state returned by the corresponding executable policy is secure independently from the input state. Intuitively, this condition is equivalent to saying that the behavior of ξ is conform with the properties defined in π for all the actions that can be performed on the system.

Moreover, we define an inference system (see Figure 5) that allows to assess an executable security policy corresponding to a compound specification.

Here is an informal explanation of these rules. Given two executable security policies ξ_1 and ξ_2 corresponding to the algebraic policies π_1 and π_2 ; respectively:

- 1) (ρ_1) : the executable specification ξ corresponding to $\pi_1 + \pi_2$ satisfies the following:
 - The set of input states of ξ is the union of the sets of input states of ξ_1 and ξ_2
 - The set of output states of ξ is the union of the sets of output states of ξ_1 and ξ_2
- 2) (ρ_2) : the executable specification ξ corresponding to $\pi_1 * \pi_2$ satisfies the following:
 - The set of input states of ξ is the intersection of the sets of input states of ξ_1 and ξ_2
 - The set of output states of ξ is the intersection of the sets of output states of ξ_1 and ξ_2
- 3) (ρ_3) : the executable specification ξ corresponding to $\pi_1 - \pi_2$ satisfies the following:
 - The set of input states of ξ is the intersection of the set of positive input states of ξ_1 and the set of negative input states of ξ_2
 - The set of output states of ξ is the intersection of the set of positive output states of ξ_1 and the set of negative output states of ξ_2

Using the axiom α and the inference rules $(\rho_i)_{i=1,2,3}$, one can demonstrate the three following theorems.

$$\begin{array}{l}
(\rho_1) : \frac{(\xi_1, \pi_1) \in \rightsquigarrow, (\xi_2, \pi_2) \in \rightsquigarrow, (\xi, \pi_1 + \pi_2) \in \rightsquigarrow}{(\xi, \sigma_i, \sigma_f) \in \varepsilon \Rightarrow ((\sigma_i \in \hat{\varepsilon}(\xi_1, \sigma_f) \cup \hat{\varepsilon}(\xi_2, \sigma_f)) \wedge (\sigma_f \in \hat{\varepsilon}(\xi_1, \sigma_i) \cup \hat{\varepsilon}(\xi_2, \sigma_i)))} \\
(\rho_2) : \frac{(\xi_1, \pi_1) \in \rightsquigarrow, (\xi_2, \pi_2) \in \rightsquigarrow, (\xi, \pi_1 * \pi_2) \in \rightsquigarrow}{(\xi, \sigma_i, \sigma_f) \in \varepsilon \Rightarrow ((\sigma_i \in \hat{\varepsilon}(\xi_1, \sigma_f) \cap \hat{\varepsilon}(\xi_2, \sigma_f)) \wedge (\sigma_f \in \hat{\varepsilon}(\xi_1, \sigma_i) \cap \hat{\varepsilon}(\xi_2, \sigma_i)))} \\
(\rho_3) : \frac{(\xi_1, \pi_1) \in \rightsquigarrow, (\xi_2, \pi_2) \in \rightsquigarrow, (\xi, \pi_1 - \pi_2) \in \rightsquigarrow}{(\xi, \sigma_i, \sigma_f) \in \varepsilon \Rightarrow ((\sigma_i \in \hat{\varepsilon}(\xi_1^+, \sigma_f) \cup \hat{\varepsilon}(\xi_2^-, \sigma_f)) \wedge (\sigma_f \in \hat{\varepsilon}(\xi_1^+, \sigma_i) \cup \hat{\varepsilon}(\xi_2^-, \sigma_i)))}
\end{array}$$

Figure 5. Proof system for the correctness of security policies.

$$(\tau_1) : \vdash \iota(\pi_1) \wedge \iota(\pi_2) \wedge \gamma(\pi_1, \pi_2) \wedge \gamma(\pi_2, \pi_1) \Rightarrow \iota(\pi_1 + \pi_2)$$

$$(\tau_2) : \vdash \iota(\pi_1) \wedge \iota(\pi_2) \Rightarrow \iota(\pi_1 * \pi_2)$$

$$(\tau_3) : \vdash \iota(\pi_1) \wedge \iota(\pi_2) \wedge \gamma(\pi_1, \pi_2) \Rightarrow \iota(\pi_1 - \pi_2)$$

where $\gamma(\pi_1, \pi_2)$ is a predicate that takes the value 1 if there is no conflict between the positive rules of π_1 and the negative rules of π_2 . Obviously, the operator $\gamma(\cdot, \cdot)$ is not symmetric. To this purpose, as stated in τ_1 , a necessary condition for the correctness of $\pi_1 + \pi_2$ is that there is no conflict between two rules of opposite signs in π_1 and π_2 . However, in τ_3 , we only require consistency between the positive rules of π_1 and the negative rules of π_2 because of the structure of $\pi_1 - \pi_2$.

For the sake of parsimony, we do not give a formal proof of all these theorems in this paper. We rather focus on their use in security policy engineering. The main contribution is that the security engineer can build a modular (or compound) security policy using composition operators and elementary security specifications. The major advantages of such approach are given in the following:

- 1) Reducing the complexity of the security policy engineering process: Decomposing the global security policy into sub-policies consistently decreases the complexity of the security policy engineering process. The first step, consisting in determining the security properties to be satisfied, would be intractable if all the components of the information system are considered as a monolithic component.
- 2) Reusing security policies: Modularity is perfectly compatible with security solution reuse. In fact, due to the proliferation of attack versioning (i.e., the attackers use multiple versions of the same attack in order to evade prevention and detection mechanisms), security policies can be simply taken on-the-shelves and adapted to fit the information system context.
- 3) Simplifying the security policy revision process: As it has been mentioned in Section III, a flaw detected during the security policy validation or testing processes results in a revision of the security policy in order to fix this error. Naturally, managing a compound policy is useful to easily detect potential errors and thwart the underlying security flaws.

VII. CASE STUDIES

A. Physical security policy

In this example, we define a physical security policy in which we consider the access of network users to organization's areas and objects (objects are essentially network elements). The physical security policy states that (1) users can only use objects that have a security level equal or less to their level, (2) users are localized in areas that have a security level equal or less to their level, (3) objects are localized in areas that have a greater security level, (4) an object with a minimum security level can get remote access to a second one if and only if the security level of the second object is not the maximum level and (5) an object which security level is not the minimum level can get remote access to other objects in the network. The algebraic specification of this policy is given in Figure 6.

The security rules of the policy are informally explained in the following:

- 1) Each object is assigned a security level
- 2) Each room is assigned a security level
- 3) Each role is assigned a security level
- 4) Any object can be localized in a given room only if its security level is less than or equal to the room's level.
- 5) Any user can access a given room only if his security level is greater than or equal to the room's level.
- 6) A physical access of a user to an object is granted only if he and the object are both localized in the same room and if the user has at least the same security level as the accessed object.
- 7) The role administrator can establish a remote access between two objects to which he has physical access for reading or writing information purposes.
- 8) Rooms must be air-conditioned
- 9) The organization holding the information system must have more than one emergency exit.

Using the translation rules discussed in the paper, the executable S-TLA⁺ specification corresponding to this algebraic specification is given in Figure 7.

B. Firewall security policy

To illustrate the validation methodology presented in this paper, we consider an example of a network for which we define some security rules applied to the traffic coming in and going out from its nodes.

Users of this network are classified into three categories: (1) internal users, (2) a web server located at

Φ	Spec	Sorts	<i>Roles, Rooms, Objects, int, emergencyexits</i>
		Opns	<i>level : Rooms \cup Roles \cup Objects \longrightarrow int</i> <i>location : Roles \cup Objects \longrightarrow Rooms</i> <i>use : Roles \longrightarrow Objects</i> <i>rmtaccess : Objects \longrightarrow Objects</i>
		preds	<i>max : int</i> <i>airconditioned : Rooms</i> <i>protected : emergencyexits</i>
	axioms	φ_1	$\forall o : Objects. r : Rooms. l1, l2 : int. level(o) = l1 \wedge level(r) = l2 \wedge l1 \leq l2 \implies location(o) = r$
		φ_2	$\forall u : Roles. r : Rooms. l1, l2 : int. level(u) = l1 \wedge level(r) = l2 \wedge l1 \geq l2 \implies location(u) = r$
		φ_3	$\forall u : Roles. o : Objects. l1, l2 : int. level(u) = l1 \wedge level(o) = l2 \forall o : Objects. location(u) = location(o) \wedge l1 \geq l2 \implies use(r) = o$
		φ_4	$\forall o1, o2 : Objects. a : Roles. l : int. level(a) = l \wedge max(l) \wedge use(a) = o1 \wedge use(a) = o2 \implies rmtaccess(o1) = o2 \vee rmtaccess(o2) = o1$
		φ_5	$\neg(\exists r : Rooms \neg airconditioned(r))$
		φ_6	$\neg(\exists e1, e2 : emergencyexits \neg protected(f1) \wedge protected(f2) \implies f1 = f2)$

Figure 6. Algebraic specification of a physical access control security policy.

the network and which can be accessed from external users and (3) external users. The network policy states that internal users are authorized to use the web service to connect either to the internal web server or to web servers external to the network. An administrator can administrate the internal web server remotely using telnet service. In addition, the web server runs an application that should be updated using FTP service. Finally, external users are only authorized to connect to the internal web server.

These rules are algebraically specified in Figure 8.

The S-TLA⁺ specification of this policy is given in Figure 9. The model checker has loaded 13 different states from a total of 148 generated states. A state of graph search having a depth equal to 3 has been detected as presenting a security policy violation. The validation of the security rules is given in Figure 10. It consists in the ability of an internal user (the administrator) to connect to a FTP server (at state 3) exploiting its privilege of connecting to the web server using the Telnet service and the right given to the web server to open FTP connections for update purpose (at state 2).

```

Finished computing initial states: 1 distinct state generated.
Error: Invariant SecPol is violated. The behavior up to this
point is:
STATE 1: <Initial predicate>
Use = [ U |-> [U |-> 0, WS |-> 0, FS |-> 0, 0 |-> 0],
WS |-> [U |-> 0, WS |-> 0, FS |-> 0, 0 |-> 0],
FS |-> [U |-> 0, WS |-> 0, FS |-> 0, 0 |-> 0],
0 |-> [U |-> 0, WS |-> 0, FS |-> 0, 0 |-> 0] ]

STATE 2: <Action line 16, col 14 to line 21, col 58 of module
Firewall>
Use = [ U |-> [U |-> 0, WS |-> 23, FS |-> 0, 0 |-> 0],
WS |-> [U |-> 0, WS |-> 0, FS |-> 0, 0 |-> 0],
FS |-> [U |-> 0, WS |-> 0, FS |-> 0, 0 |-> 0],
0 |-> [U |-> 0, WS |-> 0, FS |-> 0, 0 |-> 0] ]

STATE 3: <Action line 24, col 10 to line 26, col 65 of module
Firewall>
Use = [ U |-> [U |-> 0, WS |-> 23, FS |-> 21, 0 |-> 0],
WS |-> [U |-> 0, WS |-> 0, FS |-> 21, 0 |-> 0],
FS |-> [U |-> 0, WS |-> 0, FS |-> 0, 0 |-> 0],
0 |-> [U |-> 0, WS |-> 0, FS |-> 0, 0 |-> 0] ]
    
```

Figure 10. Security rules validation

VIII. CONCLUSION

Throughout the foregoing discussion, we have highlighted the important potential shown by combining both algebraic and executable specifications. Such combination allows to verify syntactically and semantically the SP, which constitutes a prominent need from the security perspective. The focal concept in the proposed approach is a novel rewriting system allowing to automatically map algebraic SPs into executable specifications. A study of the properties of this rewriting system (investigating more properties than those defined in this paper) is currently under study.

REFERENCES

- [1] D. Drusinsky and J. L. Fobes, "Executable specifications: Language and applications," *CROSSTALK, The journal of Defense Software Engineering*, vol. 17, no. 9, pp. 15–18, September 2004.
- [2] M. P. Fromherz, "Explore : An approach to executable specifications using logic programming and an object-oriented framework," Ph.D. dissertation, University of Zurich, Department of Computer Science, Winterthurerstr. 190, CH-8075 Zurich, 1991.
- [3] N. E. Fuchs, "Specifications are (preferably) executable," *Software Engineering Journal*, September 1992.
- [4] A. G. Y. Guan, "Executable specifications for agent-oriented conceptual modeling," 2005.
- [5] P. A. Lindsay, "Specification and validation of a network security policy model," Software Verification Research Centre, The University of Queensland, Tech. Rep. 97-05, April 1997.
- [6] M. Hamdi and N. Boudriga, "Algebraic specification of network risk management," in *ACM Workshop on Formal Methods in Security Engineering*, Washington, DC, USA, 30th October 2003, pp. 52–60.
- [7] N. B. Jihène Krihcène, Mohamed Hamdi, "Algebraic test case generation of security policies in communication networks," in *Proceedings of the 10th Nordic Conference on Secure IT Systems*, 2005.

MODULE *SecPol4*

EXTENDS *Naturals, TLC*
 CONSTANTS *Rooms, Roles, Objects*
 VARIABLES *level, location, use, rmtaccess*

noobject \triangleq CHOOSE $x : x \notin \text{Objects}$
outside \triangleq CHOOSE $x : x \notin \text{Rooms}$

TypeInvariant \triangleq
 $\wedge \text{level} \in [\text{Rooms} \cup \text{Roles} \cup \text{Objects} \rightarrow (0 \dots 2)]$
 $\wedge \text{location} \in [\text{Roles} \cup \text{Objects} \rightarrow \text{Rooms} \cup \{\text{outside}\}]$
 $\wedge \text{use} \in [\text{Roles} \rightarrow \text{Objects} \cup \{\text{noobject}\}]$
 $\wedge \text{rmtaccess} \in [\text{Objects} \rightarrow [\text{Objects} \rightarrow \{\text{"T"}, \text{"F"}\}]]$

Init \triangleq
 $\wedge \text{level} = [x \in (\text{Rooms} \cup \text{Roles} \cup \text{Objects}) \mapsto 0]$
 $\wedge \text{location} = [x \in (\text{Roles} \cup \text{Objects}) \mapsto \text{outside}]$
 $\wedge \text{use} = [x \in \text{Roles} \mapsto \text{noobject}]$
 $\wedge \text{rmtaccess} = [x \in \text{Objects} \mapsto [y \in \text{Objects} \mapsto \text{"F"}]]$

SetRoomsLevel(x) \triangleq
 $\wedge \exists y \in (1 \dots 2) : \forall z \in \text{Objects}, w \in \text{Roles} :$
 $\wedge \text{location}[z] = x \implies \text{level}[z] \leq y$
 $\wedge \text{location}[w] = x \implies \text{level}[w] \geq y$
 $\wedge \text{level}' = [\text{level EXCEPT } ![x] = y]$
 $\wedge \text{UNCHANGED } \langle \text{location}, \text{use}, \text{rmtaccess} \rangle$

SetObjectsLevel(x) \triangleq
 $\wedge \exists y \in (1 \dots 2) : \forall z \in \text{Rooms}, w \in \text{Roles} :$
 $\wedge \text{location}[x] = z \implies \text{level}[z] \geq y$
 $\wedge \text{use}[w] = x \implies \text{level}[w] \geq y$
 $\wedge \text{level}' = [\text{level EXCEPT } ![x] = y]$
 $\wedge \text{UNCHANGED } \langle \text{location}, \text{use}, \text{rmtaccess} \rangle$

SetRolesLevel(x) \triangleq
 $\wedge \exists y \in (1 \dots 2) : \forall z \in \text{Rooms}, w \in \text{Objects} :$
 $\wedge \text{location}[x] = z \implies \text{level}[z] \leq y$
 $\wedge \text{use}[x] = w \implies \text{level}[w] \leq y$
 $\wedge \text{level}' = [\text{level EXCEPT } ![x] = y]$
 $\wedge \text{UNCHANGED } \langle \text{location}, \text{use}, \text{rmtaccess} \rangle$

OrganizeObject(x, y) \triangleq x is an object, y is a room
 $\wedge \forall v \in \{x, y\} : \text{level}[v] \neq 0$
 $\wedge \text{level}[x] \leq \text{level}[y]$
 $\wedge \text{location}' = [\text{location EXCEPT } ![x] = y]$
 $\wedge \text{UNCHANGED } \langle \text{level}, \text{use}, \text{rmtaccess} \rangle$

AccessRoom(x, y) \triangleq x is a role, y is a room
 $\wedge \forall v \in \{x, y\} : \text{level}[v] \neq 0$
 $\wedge \text{location}' = \text{IF } \text{level}[x] \geq \text{level}[y] \text{ THEN } [\text{location EXCEPT } ![x] = y] \text{ ELSE } \text{location}$
 $\wedge \text{UNCHANGED } \langle \text{level}, \text{use}, \text{rmtaccess} \rangle$

UseObjects(x, y) \triangleq x is a role, y is an object
 $\wedge \forall v \in \{x, y\} : \wedge \text{level}[v] \neq 0$
 $\wedge \text{location}[v] \neq \text{outside}$
 $\wedge \text{location}[x] = \text{location}[y]$
 $\wedge \text{use}' = \text{IF } \text{level}[x] \geq \text{level}[y] \text{ THEN } [\text{use EXCEPT } ![x] = y] \text{ ELSE } \text{use}$
 $\wedge \text{UNCHANGED } \langle \text{level}, \text{location}, \text{rmtaccess} \rangle$

- [8] S. Rekhis and N. Boudriga, "A formal logic-based language and an automated verification tool for computer forensic investigation," in *First International Track on Computer-aided Law and Advanced Technologies in association with ACM Symposium of Applied Computing*, 2005.
- [9] S. Merz, "On the logic of tla+," *Computing and Informatics*, vol. 22, no. 4, 2003.

Jihene Krichene is a Ph.D student in the Engineering School of Communications (Sup'Com, Tunisia). She received her Engineering Diploma and Master Diploma in telecommunications from the National Institute of Applied Sciences and Technology (INSAT, Tunisia) and the Engineering School of Communications (Sup'Com, Tunisia) on February 2002 and November 2002, respectively. From 2002 to 2005 she worked for the National Digital Certification Agency (NDCA, Tunisia) in the Risk Analysis Team. Actually, Ms. Krichene is an assistant at the Sciences Faculty at Bizerte. She is also member of the Communication Networks and Security Lab where Ms. Krichene is conducting research activities in the areas of risk management, security projects management, security policies verification and validation and security incidents response.

Mohamed Hamdi received his Engineering Diploma, Master Diploma, and PhD in telecommunications from the Engineering School of Communications (Sup'Com, Tunisia) on 2000, 2002, and 2005; respectively. He is recipient of the Best Thesis Award from the Tunisian Telecommunications Scientific Society. From 2001 to 2005 he has worked for the National Digital Certification Agency (NDCA, Tunisia) where he was head of the Risk Analysis Team. Dr. Hamdi was in charge to build the security strategy for the Tunisian root Certification Authority and to continuously assess the security of the NDCA's networked infrastructure. He has also served in various national technical committees for securing e-government services. Currently, Dr. Hamdi is serving as assistant professor for the Engineering School of Communications at Tunis. He is also member of the Communication Networks and Security Lab (Coordinator of the Formal Aspects of Network Security Research Team), where Dr. Hamdi is conducting research activities in the areas of risk management, algebraic modeling, relational specifications, intrusion detection, network forensics, and wireless sensor networks.

Noureddine Boudriga is an internationally known scientist/academic. He received his Ph.D. in Algebraic topology from University Paris XI (France) and his Ph.D. in Computer science from University of Tunis (Tunisia). He is currently a Professor of Telecommunications at University of Carthage, Tunisia and the Director of the Communication Networks and Security Research Laboratory (CNAS) He is the recipient of the Tunisian Presidential award in Science and Research (2004). He has served as the General Director and founder of the Tunisian National Digital Certification Agency. He was involved in very active research and authored or coauthored many chapters and books. He published over 200 refereed journal and conference papers. Prof. Boudriga is the President of the Tunisian Scientific Telecommunications Society.

$$\begin{aligned}
& \text{SetRemoteAccess}(x, y, z) \triangleq x \text{ is a role, } y \text{ and } z \text{ are objects} \\
& \quad \wedge \forall v \in \{x, y, z\} : \wedge \text{level}[v] \neq 0 \\
& \quad \quad \quad \wedge \text{location}[v] \neq \text{outside} \\
& \quad \wedge \text{location}[y] \neq \text{location}[z] \\
& \quad \wedge \text{level}[x] = 2 \\
& \quad \wedge \text{rmtaccess}' = [\text{rmtaccess EXCEPT } ![y][z] = \text{"T"}] \\
& \quad \wedge \text{UNCHANGED } \langle \text{level}, \text{location}, \text{use} \rangle \\
& \text{AccessRemotely}(x, y) \triangleq x \text{ and } y \text{ are two objects} \\
& \quad \wedge \forall v \in \{x, y\} : \wedge \text{level}[v] \neq 0 \\
& \quad \quad \quad \wedge \text{location}[v] \neq \text{outside} \\
& \quad \wedge \text{location}[x] \neq \text{location}[y] \\
& \quad \wedge \text{rmtaccess}[x][y] = \text{"T"} \\
& \quad \wedge \exists z \in \text{Roles} : \wedge \text{level}[z] \neq 0 \\
& \quad \quad \quad \wedge \text{location}[z] \neq \text{outside} \\
& \quad \quad \quad \wedge \text{use}[z] = x \\
& \quad \quad \quad \wedge \text{use}' = [\text{use EXCEPT } ![z] = y] \\
& \quad \wedge \text{UNCHANGED } \langle \text{level}, \text{location}, \text{rmtaccess} \rangle \\
& \text{SecPolPr1} \triangleq \\
& \quad \forall x \in \text{Roles}, y \in \text{Objects} : \text{use}[x] = y \implies \text{level}[x] \geq \text{level}[y] \\
& \text{SecPolPr2} \triangleq \\
& \quad \forall x \in \text{Rooms}, y \in \text{Objects} : \text{location}[y] = x \implies \text{level}[y] \leq \text{level}[x] \\
& \text{SecPolPr3} \triangleq \\
& \quad \forall x \in \text{Rooms}, y \in \text{Roles} : \text{location}[y] = x \implies \text{level}[y] \geq \text{level}[x] \\
& \text{Next} \triangleq \exists x \in \text{Rooms}, y \in \text{Roles}, z \in \text{Objects}, v \in \text{Objects} : \\
& \quad \vee \text{SetRoomsLevel}(x) \vee \text{SetObjectsLevel}(z) \vee \text{SetRolesLevel}(y) \\
& \quad \vee \text{OrganizeObject}(z, x) \vee \text{AccessRoom}(y, x) \vee \text{UseObjects}(y, z) \\
& \quad \vee \text{SetRemoteAccess}(y, v, z) \vee \text{AccessRemotely}(z, v) \\
& \text{Spec} \triangleq \text{Init} \wedge \square[\text{Next}]_{\langle \text{level}, \text{location}, \text{use}, \text{rmtaccess} \rangle} \\
\hline
& \text{THEOREM } \text{Spec} \implies \square(\text{TypeInvariant} \wedge \text{SecPolPr1} \wedge \text{SecPolPr2} \wedge \text{SecPolPr3}) \\
\hline
\end{aligned}$$

Figure 7. S-TLA⁺ translation of the physical access control security policy.

SecPol	Sorts	<i>bool, int, IP, decision</i>			
	Opns	<i>True</i> :→ <i>bool</i>			
		<i>Allow</i> :→ <i>decision</i>			
		<i>internal</i> :→ <i>bool</i>			
		<i>dmz</i> → <i>bool</i>			
		<i>external</i> → <i>bool</i>			
		<i>connect</i> : <i>IP</i> × <i>IP</i> × <i>int</i>			
Axioms	φ ₁	∀ <i>ip1, ip2</i> :	<i>IP.internal(ip1)</i> = <i>True</i> ∧ <i>dmz(ip2)</i> = <i>True</i> ∧		
			<i>connect(ip1, ip2, 80)</i> ⇒ <i>Allow</i>		
	φ ₂	∀ <i>ip1, ip2</i> :	<i>IP.internal(ip1)</i> = <i>True</i> ∧ <i>dmz(ip2)</i> = <i>True</i> ∧		
			<i>connect(ip1, ip2, 23)</i> ⇒ <i>Allow</i>		
	φ ₃	∀ <i>ip1, ip2</i> :	<i>IP.internal(ip1)</i> = <i>True</i> ∧ <i>external(ip2)</i> = <i>True</i> ∧		
			<i>connect(ip1, ip2, 80)</i> ⇒ <i>Allow</i>		
	φ ₄	∀ <i>ip1, ip2</i> :	<i>IP.dmz(ip1)</i> = <i>True</i> ∧ <i>external(ip2)</i> = <i>True</i> ∧		
			<i>connect(ip1, ip2, 21)</i> ⇒ <i>Allow</i>		
	φ ₅	∀ <i>ip1, ip2</i> :	<i>IP.external(ip1)</i> = <i>True</i> ∧ <i>dmz(ip2)</i> = <i>True</i> ∧		
			<i>connect(ip1, ip2, 80)</i> ⇒ <i>Allow</i>		

Figure 8. Algebraic specification of a firewall security policy.

MODULE <i>Firewall</i>
<p>EXTENDS <i>Naturals</i> VARIABLE <i>Use</i></p> <p>$Services \triangleq \{0, 21, 23, 80\}$ $PC \triangleq \{\text{"U"}, \text{"WS"}, \text{"FS"}, \text{"O"}\}$ $Interfaces \triangleq \{\text{"I"}, \text{"E"}, \text{"D"}\}$</p>
<p>$TypeInvariant \triangleq \wedge Use \in [PC \rightarrow [PC \rightarrow Services]]$</p> <p>$Init \triangleq Use = [x \in PC \mapsto [y \in PC \mapsto 0]]$</p> <p>$Inspect \triangleq \exists p1 \in PC, p2 \in PC, s \in Services \setminus \{0\} :$ $Use' = \text{IF } ((p1 = \text{"U"} \wedge p2 = \text{"WS"} \wedge s = 80)$ $\quad \vee (p1 = \text{"U"} \wedge p2 = \text{"WS"} \wedge s = 23)$ $\quad \vee (p1 = \text{"U"} \wedge p2 = \text{"O"} \wedge s = 80)$ $\quad \vee (p1 = \text{"WS"} \wedge p2 = \text{"FS"} \wedge s = 21)$ $\quad \vee (p1 = \text{"O"} \wedge p2 = \text{"WS"} \wedge s = 80)) \text{ THEN}$ $\quad [Use \text{ EXCEPT } ![p1][p2] = s] \text{ ELSE } Use$</p> <p>$Admin \triangleq \wedge Use[\text{"U"}][\text{"WS"}] = 23$ $\wedge \exists s \in Services \setminus \{0\} : Use' = \text{IF } (s = 21) \text{ THEN}$ $\quad [Use \text{ EXCEPT } ![\text{"WS"}][\text{"FS"}] = s, ![\text{"U"}][\text{"FS"}] = s] \text{ ELSE } Use$</p> <p>$SecPol \triangleq \forall x \in PC, y \in PC, z \in Services \setminus \{0\} : Use[x][y] = z$ $\implies \vee (x = \text{"U"} \wedge y = \text{"WS"} \wedge z = 80)$ $\quad \vee (x = \text{"U"} \wedge y = \text{"WS"} \wedge z = 23)$ $\quad \vee (x = \text{"U"} \wedge y = \text{"O"} \wedge z = 80)$ $\quad \vee (x = \text{"WS"} \wedge y = \text{"FS"} \wedge z = 21)$ $\quad \vee (x = \text{"O"} \wedge y = \text{"WS"} \wedge z = 80)$</p> <p>$Next \triangleq Inspect \vee Admin$</p> <p>$Spec \triangleq Init \wedge \square [Next]_{(Use)}$</p>
<p>THEOREM $Spec \implies \square (TypeInvariant \wedge SecPol)$</p>

Figure 9. S-TLA⁺ executable specification of the firewall security policy.