Model-Driven Security Policy Deployment: Property Oriented Approach

Stere Preda, Nora Cuppens-Boulahia, Frédéric Cuppens, Joaquin Garcia-Alfaro, and Laurent Toutain

IT TELECOM Bretagne CS 17607, 35576 Cesson-Sévigné, France
`{first_name.surname}@telecom-bretagne.eu`

Abstract. We address the issue of formally validating the deployment of access control security policies. We show how the use of a formal expression of the security requirements, related to a given system, ensures the deployment of an anomaly free abstract security policy. We also describe how to develop appropriate algorithms by using a theorem proving approach with a modeling language allowing the specification of the system, of the link between the system and the policy, and of certain target security properties. The result is a set of proved algorithms that constitute the certified technique for a reliable security policy deployment.

1 Introduction

Security is concerned with assets protection. Securing the access to a file server, guaranteeing a certain level of protection of a network channel, executing particular counter measures when attacks are detected, are appropriate examples of security requirements for an information system. Such security requirements belong to a guide usually called the access control security policy. Deploying the policy means enforcing (i.e., configuring) those security components or mechanisms so that the system behavior is the one specified by the policy. Successfully deploying the policy depends not only on the complexity of the security requirements but also on the complexity of the system in terms of architecture and security functionalities.

Specifying, deploying and managing the access control rules of an information system are some of the major concerns of security administrators. Their task can be simplified if automatic mechanisms are provided to distribute or update, in short to deploy the policy in complex systems. A common approach is the formalization of the security policy, based on an access control model and the application of a *downward* process (i.e., the translation and refinement of the formal policy requirements into concrete security device configurations). Though PBNM (*Policy Based Network Management*) architectures (cf. [rfc 3198]) are such examples, a challenging problem persists: proving the deployment process to be *correct* with respect to some initial target security properties and ensuring that no ambiguities (e.g., inconsistencies [13]) are added within this process.

In this paper we aim at establishing a formal frame for the deployment of security policies in information systems. We formally prove the process of deploying a security policy related to an information system. To do so, we require (1) an expressive security access control model that covers a large diversity of security requirements, (2) a modeling language for system specification and (3) a formal expression of security properties modeling the relationships between the security policy and the system it was designed for. We propose a formal technique that combines the use of access control policies expressed in the OrBAC (Organization-Based Access Control) language [1] together with specifications based on the B-Method [2]. Our proposal avoids, moreover, the existence of inconsistencies derived from the deployment [13].

Paper Organization — Section 2 gives the motivation of our work and some related works. Section 3 presents the model on which we base our approach and establishes some prerequisites necessary for our proposal. Section 4 formally defines the link between a policy and a system, including the expression of some security properties. Section 5 provides a discussion upon our approach.

2 Motivation and Related Work

The policy-based configuration of security devices is a cumbersome task. Manual configuration is sometimes unacceptable: the security administrator's task becomes not only more difficult but also error-prone given the anomalies he/she may introduce. Guaranteeing the deployment of anomaly-free configurations in complex systems is achievable if the policy is first formalized based on an access control model and then automatically translated into packages of rules for each security device. This is the current approach in PBNM architectures where the PDP, *Policy Decision Point*, is the intelligent entity in the system and the PEPs, *Policy Enforcement Points*, enforce its decisions along with specific network protocols (e.g., Netconf [rfc 4741]). Obtaining these packages of rules (i.e., the configurations of PEPs) is the result of the *downward* translation process: the abstract policy, given the system architecture, is compiled through a set of algorithms at the PDP level into, for example, firewall scripts and IPsec tunnel configurations — all the way through bearing the system architecture details (interconnections and capabilities) [19].

The correctness of these algorithms is a crucial aspect since the system configuration must reflect the abstract policy. One can simply design such algorithms using imperative languages and then validate them via specific tools [18]. Imperative program verification is performed in three steps: (1) the specification of a program is first formalized by some properties based on a first order logic; (2) an automatic process for analyzing the program extracts its semantics, i.e., a set of equations that define the program theory in the first order logic; (3) a proof system is finally used to prove that the program has the properties of step (1) given the extracted data in step (2). The main concerns with such approaches are: (a) the verification of these properties is realized at the end and not during the algorithm implementation; (b) it is difficult to express the *interesting* properties: not only those concerning the design of operations which may be seen as generic (e.g., termination of loops or lack of side effects), but also those reflecting, in our case, some network security aspects (e.g., no security *anomalies* are introduced during the deployment of the policies [13]). To cope with these issues, we claim that the algorithms intended for enforcing deployment of policies have to be designed with proof-based development methods, like the B-Method, to allow the expression and verification of important properties (e.g., security properties).

Formal validation or security policy deployment has already been addressed in the literature. The approaches in [15] and [16] seem to be the closest to ours. Jürjens et al. propose in [15] to apply UMLsec [14] to analyze some security mechanisms enforced with respect to a security policy. UMLsec is an extension of UML which allows the expression of some security-related information in UML diagrams. Stereotypes and tags are used to formulate the security requirements. Analyzing the mechanisms means verifying whether the requirements are met by the system design. For this purpose two models are proposed: (1) a Security Requirements Model which includes architectural or behavioral system details in a prescriptive manner and (2) a Concretized Model summarizing a concrete architecture which should satisfy the security requirements. Both models appear as UMLsec diagrams. The verification is realized using the UMLsec tools which includes several plugins that uses (1) SPIN (*Simple Promela Interpreter*) for model-checking and (2) SPAAS (an *automated theorem prover for first-order logic with equality*) or Prolog as theorem provers given that some UMLsec sequence diagrams are automatically translated to first-order formulas. Applying the right plugin depends on the scenario, i.e., the architecture and the security requirements. However, the approach in [15] presents some drawbacks. First, no abstract model is employed for modeling the policy. Second, the Concretized Model must already exist in order to automatically derive the first-order formulas to be automatically proved by SPAAS. And finally, the expressions of security properties are application-dependent: there are no generic properties dealing with the anomalies that could exist within single- or multi-component network security policies.

Laborde et al. present in [16] a different solution to the problem of deploying security policies: the use of (1) Petri Nets as the language to specify the system and (2) CTL (*Computational Tree Logic*) as the language to express the security properties. Four generic system *functionalities* are identified and modeled as different Petri Nets which are then interconnected in order to specify the system (i.e., each security device is modeled by a Petri Net): channel (e.g., network links), transformation (IPsec and NAT), filtering (to include the firewalls) and the *end-flow* functionality for the hosts (the active and passive entities in the network). The Petri Nets transitions for each PEPs (here, firewalls and IPsec tunnels) are guards which actually represent the security rules to be enforced by the PEP. The policy model is RBAC-based. However, no clear downward (i.e., refinement) approach is defined and no algorithms for selecting the right PEP are described either. The model-checking verification is realized after manually deploying the policy and consequently this approach can be applied to relatively simple architectures. Besides, it is not clear whether other functionalities (e.g., intrusion detection performed by IDSs) can be taken into account with one of the four functionalities.

The research presented in [22] proposes the use of Event B specifications to provide a link between the two levels of abstractions provided by the OrBAC model. As further research perspectives, the authors mentioned that their models can be reused for further developments of real infrastructures with respect to their security policy. We consider our work as a natural continuation of such a research line. As we did, the authors in [22] chose not to address the proof of an OrBAC policy in terms of conflicts. The use of automatic tools (e.g., MotOrBAC [4]), allows us to assume that the policy is consistent and free of anomalies. Similarly, Coq was proposed in [7] to derive OCaml algorithms for conflict detection in firewall policies. The use of Coq as a theorem prover to derive refinement algorithms can also be found in [21]. The authors provide a solution to detect and remove conflicts in policies defined as tuples <permission/prohibition, subject, read/write/execute, object>. Finally, and regarding the security policy deployment domain, there exist in the literature several proposals. Some are more or less RBAC-based (e.g., proposal presented in [5]), others propose different languages for the high level policy definition. Although the efforts are significant [10], often such languages are not generic enough [17], covering only some specific security applications (e.g., host firewalls, system calls management); or they do not address some key policy matters like the conflict management or the dynamic and contextual security requirements [8].

3 Model and Notation

We propose a refinement process that guarantees anomaly-free configurations ([13]). The process derives a global policy into specific configurations for each security component in the system. Our proposal provides the set of algorithms for such a refinement process, and proves the correctness of the outgoing algorithms. We briefly justify in the sequel the choice of our formalisms. We also describe the necessary concepts to establish the link between policies and architectures.

3.1 Choice of OrBAC and B-Method

The OrBAC [1] (Organization-Based Access Control) model is an extended RBAC [20] access control model which provides means to specify contextual security requirements. It allows the expression of a wide range of different requirements both static and contextual. OrBAC is well-known for being a robust language for defining complex administration tasks at the organizational level. Existing automatic tools (e.g., MotOrBAC [4]) ease, moreover, the administration of tasks using this model.

In contrast to model checking, we choose the B-Method – a theorem proving approach – for various reasons. First, the performances of theorem-proving tools are not influenced by system complexity. For example, we do not make any assumption concerning the number of nodes in the architecture. Second, the B Method eases the use of refinement paradigms. Even if *B refinement* does not necessarily mean an enhancement of system specifications (i.e., here it denotes the weakening of the preconditions and of the operation indeterminism towards the implementation level), there is always the possibility of keeping a B specification up to date. The B refinement allows the decomposition of system specification in layers and particular B clauses (e.g., use of SEES and IMPORTS clauses). This allows a modular system proof. New modules can be added to a B specification and existent ones may be assumed as being already proved; there is no need to totally reprove (i.e., recompute) the new specification. We therefore reuse an already proved specification. This aspect is very important as the security functionalities may be changed in a given system. Moreover, since specifying the system is an important step in our proposal, the link with the security policy must be established in a specific way. In this sense, the OrBAC philosophy considers that the security policy must be detached by functionality and by technology details at network level; and that changing the system architecture (hereinafter system or network with the same meaning) has no impact in policy definition. The same OrBAC policy may be specified for two different systems. Finally, the specification language must ensure the previous constraint: changing the security policy or the architecture should not trigger a new call for a total system proof.

We consider that the B-Method is suitable to achieve these purposes: in our approach the policy SEES the system it was designed for. The SEES clause in the B-Method makes the assumption that the *seen* system is already proved (i.e., the INVARIANTS of the SEEING module are being proved with the assumption that the INVARIANTS of the SEEN module are already proved). Let us notice that in our approach, we use the terminology *security property* as a synonym of *correct* deployment of a security policy in a system, meaning that it is achievable whenever certain specific security properties are verified. For instance, all network traffic between two network zones is protected if all traffic passes through an IPsec tunnel with certain parameters. If an IPsec tunnel is established and there is no IPsec tunnel anomaly [12] related to the current tunnel, the *integrity* security property is consequently verified. Hence, in a B specification there is the possibility of capturing such details at the INVARIANT clause level.

Some security requirements are dynamic or contextual. It is sometimes necessary to add new security functionalities to the given system. For example, some firewalls are upgraded with new functionalities (e.g., temporal functionality). Taking into account all security functionalities is out of our scope. We, therefore, address only some basic functionalities such as packet filtering, IPsec tunneling and signature-based Intrusion Detection. If further functionalities are added to the specification, their semantics must be reflected in particular SETS, CONSTANTS and consequently PROPERTIES clauses; but the main deployment algorithms should remain unaffected.

3.2 Policy and System Modeling

The starting point in deploying a security policy is a set of OrBAC abstract rules. A first assumption is that the abstract policy is consistent: no OrBAC conflicting rules. This is ensured by some pertaining tools like MotOrBAC which implements the conflict resolution described in [9].

The *context* definition may be related to a specific subject, action and object. Consequently, it is necessary to instantiate the corresponding subjects, actions and objects for each such contexts before deploying the OrBAC rules over each PEP. The abstract OrBAC rules, Permission(org, r, a, ν, c), must be brought to a concrete OrBAC expression [1], Is_permitted(org, s, α, o, c). Even if a large set of concrete security rules will have to be deployed, this may be the only option if the security requirements imply only such context definitions. At this point, we refer to the works in [22] that addressed the refinement problem: the OrBAC abstract expression towards a concrete one and using the B-Method (cf. Section 2). The works in [22] stopped at our stage, i.e., the link with the system. Therefore we will make a second assumption: the OrBAC concrete rules (i.e., Is_permitted(org, s, α, o, c)) are already available and they represent the input to our deployment process. The main entities to implement our approach are described as follows. The "security policy" is defined as the set of rules over the domain (Subjects $\times$ Actions $\times$ Objects $\times$ Contexts). "Subjects" and "objects" represent active and respectively passive entities in the network. A host in a subnetwork may be modeled as a subject in contrast to a web-server which may be seen as an object; not only the hosts/network components but also the clients and servers applications may be seen as subject-object entities. The "actions" are defined as network services (e.g., http and https are actions of the same abstract activity, *web*). We should also include the "contexts" in which some rules are activated. These may be bound in *hard* with some functionalities. For example, the *protected* context relies on IPsec functionalities and the *warning* context on IDS functionality. *Is_permitted(s, α, o, default)* is activated in the *default* context only if a path from *s* to *o* exists, so the firewalls on this path have to open some ports corresponding to the action α. Finally, the approach also includes all those interesting "nodes" in the network (subjects and objects) will appear as nodes in a connected graph. Those having security functionalities are the PEPs. A PEP may also be a subject or an object.

Figure 1 depicts a sample network in which an access control policy must be deployed. The system is modeled as a graph (cf. Figure 2). In real networks, each link may have a real cost or weight (e.g., an overhead required to send IP packets over the link and inversely proportional to the bandwidth of the link). A routing protocol like the OSPF, *Open Shortest Path First* [rfc 2328], establishes routes in choosing, for example, the shortest paths (i.e., the less expensive). We choose positive integers for the link costs and we assume that IP datagrams always follow the shortest path between two points. In such a system, the PEPs must be enforced with the right decisions corresponding to each security rule. For didactical reasons we take into account only permissions. One may choose to enforce the same rule in all PEPs having the same functionalities: this is what we call a redundancy anomaly and this is what we try to avoid. Our configuration approach is the following: for each permission, only the *interesting* PEPs must be identified and enforced. This leads us to consider an algorithm for selecting

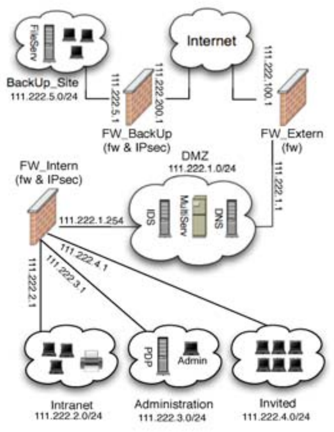

Fig. 1. The Real Architecture

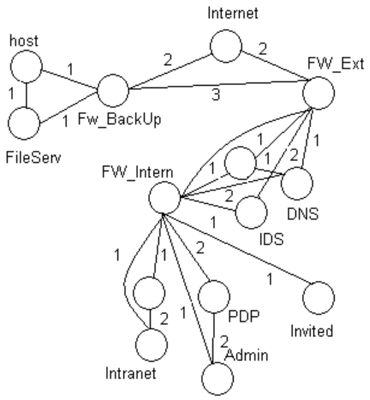

Fig. 2. The Corresponding Graph

the interesting PEPs: the well placed PEPs for each security rule. For example, a rule in a default context takes into account only the firewalls on a certain path. In a protected context, two PEPs must configure an IPsec tunnel and all firewalls on the tunnel path must permit the establishment of the tunnel. Regarding the warning context, the most down-stream IDS (i.e., the closest IDS to the destination) is enough and more efficient to spoofing attacks than the most up-stream IDS (i.e., the closest to the source).

We aim to formally implement this approach in B and go as far as possible towards the IMPLEMENTATION level. In this manner we will capture all interesting details for the security property enouncement. In the following sections we consider the policy at concrete OrBAC level as described above. We deal with a system where some distributed nodes have security functionalities (PEPs) and some are either active entities (subjects) or resources (objects) or both.

4 Policy Deployment Process: Formal Specification

Not all B machines will be carried out to an IMPLEMENTATION level: the *Policy* and *Network* (cf. Figure 4) machines should be instantiated for each scenario. The data these machines manipulate does not require highly specialized mathematical objects: only lists/sequences must be provided. But the other machines we present will have an IMPLEMENTATION structure. We describe in detail the Policy and Network (system) machines; they will be incorporated to our model development as a result of an IMPORTING machine: *Deployment* machine which also imports other machines necessary to our process. The *Path* machine will implement a tracing path algorithm necessary to select the well placed PEPs for each security rule which are then updated by the *UpdatePep* machine (cf. Figure 3).

4.1 Policy and Network Machines

The SUBJECTS, ACTIONS and OBJECTS will represent deferred sets but we prefer for the moment a concrete/enumerated set of CONTEXTS: *default*, *protected*, *user-defined* and *other*. The set of permissions to be deployed as well as the matching *nodes-subjects*, *nodes-objects* may be considered as CONSTANTS. They will be defined via some relations, more precisely functions. As already mentioned we prefer to bind in *hard* a context to a security functionality; we model this by the *matching* relation. Moreover, the user will be given the possibility of defining other types of context activation. For instance, a certain user-defined context may impose a hub-and-spoke tunnel configuration so the user must be able to manually indicate the hub and the spokes (nodes in the network). We model this by the *context* constant relation. Besides, the permissions are progressively read in the deployment process. An abstract variable is consequently necessary, the Read_Permissions. All these semantics will be summed up at PROPERTIES level.

Regarding the dynamic part of the Policy machine: the Read_Permissions is initialized with the empty set and some simple operations are necessary (1) to read and return a permission (read_permission) and (2) to read the attributes (subject, object, action and context) of a permission (read_data_in_permission). The INVARIANT is a simple one, it checks the variable type. Other inquiry operations (no_read_permission, no_more_permissions) simply return *true* or *false*.

Concerning the Network machine, we can envision the following two options. We can use an abstract machine encapsulating a node, say the Node machine. It should contain at least the node functionalities as a deferred SET. Node could be imported in our project by renaming: the project will therefore contain as many renamed Node machines as the existent ones in the real network. In the same project, a different SEEN machine will define, via a constant, the network

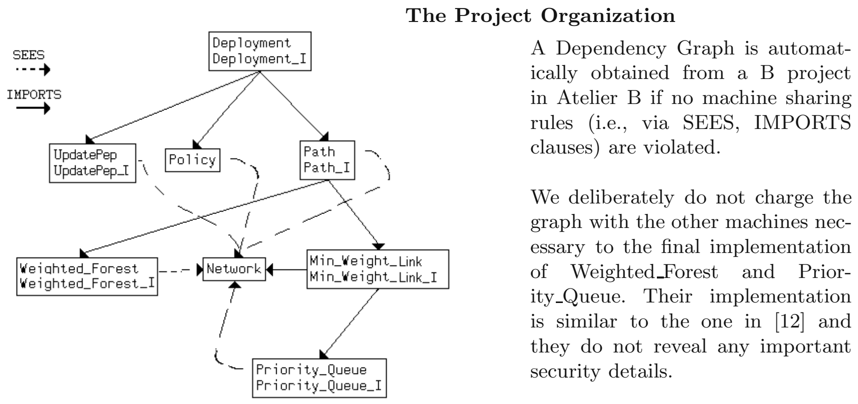
The Project Organization
SEES
IMPORTS
Deployment
Deployment_I
UpdatePep
UpdatePep_I
Policy
Path
Path_I
Weighted_Forest
Weighted_Forest_I
Network
Min_Weight_Link
Min_Weight_Link_I
Priority_Queue
Priority_Queue_I
A Dependency Graph is automatically obtained from a B project in Atelier B if no machine sharing rules (i.e., via SEES, IMPORTS clauses) are violated.
We deliberately do not charge the graph with the other machines necessary to the final implementation of Weighted_Forest and Priority_Queue. Their implementation is similar to the one in [12] and they do not reveal any important security details.

Fig. 3. Dependency Graph

topology (e.g., the nodes connections). The second option is to define a machine Network containing the topology description from the beginning. This way, we do not require other machines to carry out the network topology. Network will simply be imported only once in the project.

We choose the second solution. The Network machine models a graph: a non-empty set of *Nodes*, a set of $Links \in Nodes \times Nodes$ and a *weight* function binding a link to a natural number. We choose to identify each node by a natural number in the sequence $1..nn,\ where\ nn = \mathrm{card}(Nodes)$. These are constants that require a refinement (i.e., valuation) at the IMPLEMENTATION level with concrete data for each different topology. The graph is undirected, so $Links \cap Links^{-1} = \emptyset$. The connectedness assumption is caught by All_Links*= $(Nodes) \times (Nodes)$ (where * stands for the transitive closure of relations). Each node may have security functionalities: $functionality \in Nodes \leftrightarrow FUNCTIONALITIES$ (a constant relation). Some definitions are also necessary when a path is traced in the network: all links having a common node and the cost/weight of a set of links. These are the λ-functions *cost* and *neigh_nodes*.

When choosing the variables of the Network machine we take into account not only the network parsing aspect but also the nodes involved in the construction of paths (shortest paths) given a source node and a destination one. We, therefore, introduce some *processed* nodes (*PPnodes*) and links variables necessary in our shortest path algorithm. The INVARIANT of the Network machine acts on the variables type. We follow a generous style in specifying the operations: almost each operation has some preconditions. The generous style, in contrast to the *defensive* style which considers some internal operation tests (e.g., IF, SELECT substitutions), is more in the spirit of a B specification ([2]) as it demands prior design and specification. Such operations may be called from somewhere else (i.e., operations of other machines) and their preconditions must be verified; otherwise, they may not *terminate*.

Policy SEES Network (cf. Figure 4). Although the constants of Network may appear in the operations of a SEEING machine (e.g., Policy) its variables may be read-only. No operation of a SEEN machine can be called from within a SEEING machine. Therefore, Network is not aware of the fact that it is seen. Even if there are specific proof-obligations generated as a result of a SEES clause ([2]), the invariants of the SEEN machine are considered already proved. If we change the Policy for the same Network, the latter is once and independently proved. A SEEN machine, and consequently Network, may be imported only once somewhere in the project. We pay attention and we import Network only in a machine that really necessitates more than read-only variable references. This is illustrated in Figure 3.

4.2 A Tracing Path Algorithm

The role of a tracing path algorithm in our development is to find the security devices that must enforce each policy rule. These security devices must have the right functionalities and must be well placed in the network. If the right functionalities are indirectly designated by the OrBAC contexts (i.e., *default* - firewall, *protected* - IPsec and firewalls, *warning* - IDS), finding the well placed device in the network is not obvious. Nevertheless things are getting simpler if we consider that IP packets follow the shortest path in the network. One could say this is a severe assumption, but conciliating a given routing policy with our deployment process is a simpler matter: it suffices to take into account the few hops a route may involve.

Therefore, we use Dijkstra's shortest path algorithm. Implementing such an algorithm in imperative languages is not too difficult but it is not obvious using the B-Method. There are already B algorithms for deriving spanning trees, [11], [3], but none for shortest-path trees. We base our path derivation algorithm on the works in [11]; we could not totally reuse their method: the shortest path may not go along the minimum spanning tree. Moreover, shortest-path tree changes as a result of choosing different source and destination nodes. Some implementation details in [11] concerning the priority-queues turned out to be extremely useful: we reused them although the lists would have been much easier to manipulate. However, we mention that the project in [11] violates a sharing rule: a SEEN machine must be imported once but IMPORTS must not introduce loops in the project. We believe their error is due to their prover which did not check on machine sharing rules.

Path Machine, Weighted Forest, Min Weight Link: Path machine SEES Network whose constants (Nodes, Links and weight) are used. We need a definition of a path in the network. But as a pre-requisite we have to formalize the notion of a tree, more precisely of a spanning-tree: a forest with (n - 1) links, where n = card(Nodes) and a forest is a cycle-free set of links. We also need a definition of the set of paths from a source node to a destination node: all (i.e., the union of) adjacent links with the source and destination as extremity nodes. We are therefore interested in selecting the less-expensive path in this set of paths. This will represent the shortest path which will be simply selected once the shortest-path tree is generated (shortest_path_tree). In what follows we introduce only some specific B details we faced when designing Path machine and its implementation (the termination of the shortest_path_tree operation is ensured by our assumption of a connected graph).

Implementing the Path machine with the previous specifications would be difficult. The IMPLEMENTATION will therefore import two machines: *Min_Weight_Link* machine, to find the minimum link weight in a set of links adjacent to some processed nodes (Dijsktra's algorithm) and *Weighted_Forest* machine, to build the tree as a union of links. The tree is noted LL which is an abstract variable of Weighted_Forest. LL finally represents the shortest-path-tree. Due to space limitations we do not go further with our algorithm. The complete implementation of Weighted_Forest and Min_Weight_Link has several hundreds of B code lines. We mention only that we used the Abstract_Constants clause in order to avoid the error in [11] regarding the SEES and IMPORTS clauses: if an abstraction SEES a machine, all the further refinements must also

```
MACHINE Policy
SEES Network
SETS
    SUBJECTS;
    ACTIONS;
    OBJECTS;
    CONTEXTS={default, prot, user_def,
other_ctx}
DEFINITIONS
    Nodes==1 .. nn
CONSTANTS
    context, Permissions, Subject, Object, match-
ing
PROPERTIES
    context ∈ CONTEXTS ↔ seq(Nodes) ∧
    Permissions ∈ P(SUBJECTS ×
ACTIONS × OBJECTS × CONTEXTS) ∧
    matching ∈ CONTEXTS ↔ FUNCTIONAL-
ITIES ∧ Subject ∈ SUBJECTS → Nodes ∧
    Object ∈ OBJECTS → Nodes

VARIABLES
    Read_Permissions
INVARIANT
    Read_Permissions ⊆ Permissions
INITIALISATION
    Read_Permissions:= ∅

OPERATIONS
    no_read_permission =
      Read_Permissions:= ∅ ;

    permission ← read_permission =
      PRE Read_Permissions ≠ Permissions
      THEN ANY per WHERE
per ∈ Permissions-Read_Permissions THEN
        permission, Read_Permissions :=
per, Read_Permissions ∪ {per}
        END        END;

    bb ← no_more_permissions =
bb:=bool(Read_Permissions=Permissions);

ss,aa,oo,cc ← read_data_in_permission(pp) =
PRE Read_Permissions ≠ ∅ ∧
      pp ∈ Read_Permissions THEN
ANY sub, act, obj, ctx WHERE sub ∈ SUB-
JECTS ∧ act ∈ ACTIONS ∧ obj ∈ OBJECTS
∧ ctx ∈ CONTEXTS ∧ {sub ↦ act ↦ obj ↦
ctx}={pp} THEN
            ss, aa, oo, cc:=sub, act, obj, ctx
        END
    END

END /*Policy.mch*/
```

```
MACHINE Network
SETS
    FUNCTIONALITIES = {fw, ipsec, ids, other}
CONSTANTS
    nn, Links, weight, functionality
DEFINITIONS
    Nodes==1 .. nn;
    All_Links==Links ∪ Links ⁻¹ ;
    cost== λ LL.(LL ∈ P (Links) | ∑ link.(link
∈ LL | weight(link)));
    neigh_nodes== λ LL.(LL ∈ Nodes | ( ⋃ ii.(ii
∈ Nodes ∧ LL ↦ ii ∈ Links | {LL ↦ ii})) ∪ ( ⋃
jj.(jj ∈ Nodes ∧ jj ↦ LL ∈ Links | {jj ↦ LL})))
PROPERTIES
     nn ∈ NAT1 ∧ Links ∈ Nodes ↔ Nodes ∧
weight ∈ Links → NAT ∧ Links ∩ Links ⁻¹ = ∅
∧ All_Links*=(Nodes) × (Nodes) ∧
    card(Links) ∈ NAT1 ∧ functionality ∈ Nodes
↔ FUNCTIONALITIES ∧ (cost)(Links) ∈ NAT

VARIABLES
    Read_Links, Read_Nodes, Neighbors, PPnodes,
Read_Neighbors
INVARIANT
    Read_Links ∈ P (Links) ∧ Read_Nodes ∈ P
(Nodes) ∧ Neighbors ∈ P (Links) ∧ PPnodes ∈ P
(Nodes) ∧ Read_Neighbors ∈ P (Links)
INITIALISATION
    Read_Links, Read_Nodes, Neighbors, PPn-
odess, Read_Neighbors:= ∅ , ∅ , ∅ , ∅ , ∅

OPERATIONS /* for space limitation reasons,
not all operations are detailed */
    node, func ← read_node =
      PRE Read_Nodes ≠ Nodes THEN
        ANY nod WHERE nod ∈ Nodes ∧ nod
∉ Read_Nodes THEN
        node, func, Read_Nodes:=nod, functional-
ity[{nod}], Read_Nodes ∪ {nod}
        END        END;

    uu, vv, ww ← read_link =
      PRE Read_Links ≠ Links THEN
        ANY ii, jj WHERE ii ∈ Nodes ∧ jj ∈
Nodes ∧ (ii,jj) ∈ Links-Read_Links THEN
        uu, vv, ww, Read_Links:=ii, jj, weight(ii,
jj), Read_Links ∪ {ii ↦ jj}
        END        END;

    new_neighbors(uu) = PRE uu ∈ Nodes ∧
(Neighbors ∩ (neigh_nodes)(uu)) ≠ ∅ ∧ uu ∉ PPn-
odes THEN ... /*set new Neighbor Links*/
... /*other operations*/

END /*Network.mch*/
```

Fig. 4. Policy and Network Machines

SEES this machine and the final IMPLEMENTATION cannot IMPORTS the seen machine.

4.3 Deployment Implementation and Security Properties

The root machine of our model is the Deployment machine (cf. Figure 5). Its abstract specification is quite simple: there is only a Boolean concrete variable, *deployment_ok* modified by an operation, *deploy*. The refinement of this operation is based on other operations of the IMPORTED machines Policy, Path and UpdatePep. Network is imported in our model indirectly, via the Path machine. By using the IMPORTS clause, allowed only from within an IMPLEMENTATION there are specific obligation-proofs generated for the IMPORTING machine. We deliberately leave the IF substitution unfinished: there are tests concerning the existence of a path in the network according to the type of context. Therefore, *exists_path*, a boolean variable of the Path machine, is valued in function of several other variables: the security functionalities of the source and destination nodes (e.g., the IPv6 protocol incorporates the IPsec suite but this functionality may not be considered in the IPsec tunnel extremities), the security functionalities in their neighborhood (e.g., for IDS rules) or the security functionalities of the whole path (*path_set*) between the source and the destination. These tests are simple and rely on the definition and implementation of the Path machine. Finally, the *UpdatePep* machine stores, for each PEP, the security configuration as a set of rules $\{sub \mapsto act \mapsto obj \mapsto ctx\}$ modified via the *update_pep* operation. The concrete *deployment_ok* variable respects the data types required in an IMPLEMENTATION. It needed no further refinement and we defined it as a CONCRETE_VARIABLE in the abstraction.

4.4 Security Properties

A security property is generally expressed at a more abstract level than the security requirements. A security property may rely on the correct enforcement of several security requirements. Moreover, a property may still not be verified after the deployment of all security rules. Often, a property violation is the result of anomalies in deploying the policy. Our refinement approach is a property-aware one: the target properties determine the enforcement of the security devices. In the implementation of the Deployment machine (cf. Figure 5), we denote by P_1-P_8 some of the most interesting application-independent properties the policy deployment process should verify. We do not claim to achieve a thorough analysis of security properties: some may be enounced at higher levels ([6]) and some may be identified from specific security requirements ([15]).

- **Completeness:** Captured by INVARIANT P_1, this property states that if the network path from a subject to an object is correctly computed (i.e., it exists and the security devices belonging to this path have the right functionalities with respect to the context) the security rule may and will be deployed.

```
IMPLEMENTATION Deployment_I
REFINES Deployment
IMPORTS Path, Policy, UpdatePep
INVARIANT

P_1 { $(deployment\_ok = \text{true}) \Leftrightarrow (exists\_path{=}\text{true}) \wedge$

P_2 { $\forall(sub, act, obj).(sub \in SUBJECTS \wedge act \in ACTIONS \wedge obj \in OBJECTS \wedge$
    { $(sub \mapsto act \mapsto obj \mapsto prot) \in PERMISSIONS \Rightarrow \neg\exists(n1, n2).((n1 \mapsto n2) \in Links \wedge$
    { $\{n1, n2\} \subseteq shortest\_path(Subject(sub), Object(obj)) \wedge$
    { $(sub \mapsto act \mapsto obj \mapsto default) \in config[\{n1\}] \cap config[\{n2\}])) \wedge$

P_3 { $\forall(sub, act, obj).(sub \in SUBJECTS \wedge act \in ACTIONS \wedge obj \in OBJECTS \wedge$
    { $(sub \mapsto act \mapsto obj \mapsto default) \in PERMISSIONS \Rightarrow$
    { $\forall(node).(node \in Nodes \wedge fw \in functionality\ [\{node\}] \wedge$
    { $node \in shortest\_path(Subject(sub), Object(obj)) \Rightarrow$
    { $(sub \mapsto act \mapsto obj \mapsto default) \in config[\{node\}])) \wedge$

P_4 { $\forall(sub, act, obj).(sub \in SUBJECTS \wedge act \in ACTIONS \wedge obj \in OBJECTS \wedge$
    { $(sub \mapsto act \mapsto obj \mapsto default) \in PROHIBITIONS \Rightarrow$
    { $\forall(node).(node \in Nodes \wedge fw \in functionality\ [\{node\}] \wedge$
    { $node \in path(Subject(sub), Object(obj)) \Rightarrow$
    { $(sub \mapsto act \mapsto obj \mapsto default) \in config[\{node\}])) \wedge$

P_5 { $\forall(sub, act, obj).(sub \in SUBJECTS \wedge act \in ACTIONS \wedge obj \in OBJECTS \wedge$
    { $(sub \mapsto act \mapsto obj \mapsto authent) \in PERMISSIONS \Rightarrow (authentication \in history[\{sub\}]))$

INITIALISATION
  deployment_ok:=true
OPERATIONS
  deploy = VAR permis, bb, kp IN no_read_permission;
      bb ← no_more_permissions; kp:=0;
    WHILE bb=false DO
      permis ← read_permission; kp:=kp+1;
   VAR suj, act, obj, cont, src, dest, path_set IN
suj, act, obj, cont ← read_data_in_permission(permis);
      src:=Subject(suj); dest:=Object(obj);
      IF $dest \neq src$ THEN
        shortest_path_tree(src); path_set ←shortest_path(src, dest);
        ...
        update_pep(src, dest, path_set); deployment_ok:=true
     ELSE          bb:=true /*loop exit*/   END    END
    INVARIANT          $bb \in$ BOOL $\wedge$

P_6 { $(cont = default) \Rightarrow (exists\_path = true) \wedge$

P_7 { $(cont = default \vee cont = logging) \Rightarrow (exists\_path{=}true \wedge$
    { $\exists\ node.(node \in Nodes \wedge node \in shortest\_path(src, dest) \wedge$
    { $fw \in functionality[\{node\}])) \wedge$

P_8 { $(cont = prot) \Rightarrow (exists\_path = true) \wedge$
    { $ipsec \in (functionality[\{src\}] \cap functionality[\{dest\}])$

    VARIANT card(Permissions)-kp      END    END /*deploy*/
END /*Deployment_I*/
```

Fig. 5. Deployment Implementation

- **Accessibility (and Inaccessibility):** Property P_6 states that for each permission rule, a subject is able to access an object with respect to the policy. Thus, there must be a path between the subject and the object network entities and if this path involves some firewalls, they must all permit the action the subject is supposed to realize on the object. In this manner, the *default* context is activated (this may be seen as a *minimal* context). However, P_6 must be seen as a *partial* accessibility property: it is verified at each WHILE loop iteration, i.e., it does not take into account all deployed rules. In order to guarantee the *global* accessibility, P_3 relies on the correct deployment of all permissions. We simplified the notation: given that the operation $shortest\_path(src,\ dest)$ returns a set of links called $path\_set$, we should have written: $node \in path\_set$. In P_3 we use *config* which is defined in the UpdatePep machine: $config \in Nodes \leftrightarrow PERMISSIONS$ and $config[\{ni\}]$, the image of the $\{ni\}$ set under *config*, is the set of rules already deployed over the node *ni*.

  Following the same reasoning we can also enounce the global inaccessibility property, P_4: there should be no open *path* from the subject to the object. The *path* variable is given in the Path machine and regroups a set of links from a source node to a destination one.
- **All traffics are regulated by firewalls:** Property P_7 is also interesting when there is a new context called *logging*: this context is managed by those devices with a logging functionality as today's most popular firewalls.
- **Integrity and confidentiality property:** This property is related to the establishment of IPsec tunnels. It ensures the extremities of the IPsec tunnel. Moreover, particular IPsec configurations may include recursive encapsulation of traffic on a path. Verifying this property begins at higher levels: if no OrBAC security rule is enounced with a protected (*prot*) context, no further verification is necessary. To ensure the protected context activation, a configuration of an IPsec tunnel is necessary. If no specific information concerning the IPsec tunnel establishment is provided, we may suppose the following two cases: (1) the subject/source and the object/destination are IPsec enabled (e.g., IPv6 nodes and end-to-end tunnel) or (2) at least one node in their neighborhood (e.g., site-to-site tunnel) has IPsec functionalities. For the first case, it suffices to check on the IPsec functionalities on both the subject and the object nodes and this is captured by the P_8 property of the WHILE loop. The second case is handled as follows: in one of the IMPORTED machines on the Weighted_Forest development branch, we provide an operation $predec(node)$ which returns the *precedent* node in the current shortest-path from the source (src) node to the destination (dest) node. Via PROMOTE clauses, the operation may be called by *higher* IMPORTING machines, including the Deployment machine. We consequently check on the predec(dest) and predec^{-1}(src) nodes as in the P_8 formula.

  There is a further case that cannot be addressed at the WHILE loop INVARIANT level (i.e., after each iteration) because it relies on the correct enforcement of all IPsec tunnels in the network. Figure 6 shows an example of successive encapsulations on a site-to-site topology. It may result in violating the confidentiality property. It also shows an example where the source-destination traffic is twice encapsulated: by n1 (IPsec tunnel mode between n1 and n2) and by n3 (between n3 and n4). The configurations of n2 and n3 neighbor nodes include a security rule in a default context (i.e., $\{sub \mapsto act \mapsto obj \mapsto default\}$) allowing the IP traffic to pass the section n2-n3 with no encapsulation: the confidentiality is not preserved in this topology (i.e., between n1 and n4). Such an anomaly is the result of deploying separately the IPsec tunnels and consequently it cannot be controlled by the WHILE INVARIANT after each iteration. Nevertheless, the main INVARIANT of the Deployment machine is not reproved during these iterations but after the loop termination and consequently the IPsec anomaly can be dealt with only at this level. The P_2 formula, in logical conjunction with the *completeness* property, accomplishes the integrity and confidentiality property.

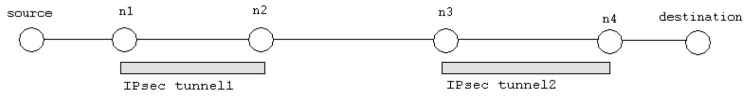

Fig. 6. Chained IPsec Tunnels

– **Authentication:** P_5 is interesting if we deal with an *authentication* context: an action that cannot be applied in a certain context unless an authentication process is achieved. We can verify these cases by providing a variable that records the actions realized by the subject concerned with the authentication. We, therefore, impose a workflow constraint ($history[\{sub\}]$ is the set of actions that *sub* realized, with $history \in SUBJECTS \leftrightarrow ACTIONS$).

5 Discussion

The B project depicted in this paper was realized using Atelier B v4.0. Validating a B project consists in proving the Proof Obligations (POs) automatically generated after analyzing and type-checking the entire project. The functional correctness of each machine is validated separately with respect to the specific B inter-machine clauses (SEES, IMPORTS, etc.). The current project was validated with the assumption of a conflict-free OrBAC policy and of a correct system architecture: no lack of security functionalities in the security components placed on the shortest-paths. This leads us to conclude that the outgoing algorithms are correct with respect to the security properties we considered. The number of POs automatically generated for each machine varied based on the operational complexity: from 2 for the Policy machine which involved very simple operations to 272 for the Min_Weight_Link implementation; for the latter one, 110 POs were automatically discharged, the rest being interactively proved.

The choice of the OrBAC model and of the B-Method was motivated by the type of applications that we address in this paper: the deployment of access control security policies. However, our work shows some limitations. On the one hand, our approach focuses on the deployment of policies in systems with an already existing set of security devices. The appropriate deployment of the access control policy is closely related to the interconnections and the capabilities of these security devices. As we avoid the intra- and inter-component anomalies ([13]), there may be unaccomplished security requirements because of a deficient security device capability. Thus, an improvement to our approach would be to find, for a given system and a given security policy, the best security architecture so that all security requirements be met. On the other hand, the type of security requirements may also induce a limitation to our approach. As long as we consider only access control requirements, the B-Method is very efficient. However, temporal logic specifications cannot be addressed with the B Method. Therefore, except for the specific case of authentication, the security requirements involving a trace-like modeling (an ordered set of actions to be realized by a subject on an object) cannot be addressed with the B Method. The authentication can be dealt with since the subject needs to accomplish a single (previous) action (modeled by a *provisional* — *history* — OrBAC context) before gaining the access.

6 Conclusions

The configuration of security devices is a complex task. A wrong configuration leads to weak security policies, easy to bypass by unauthorized entities. The existence of reliable automatic tools can assist security officers to manage such a cumbersome task. In this paper, we established a formal frame for developing such tools. Our proposal allows the administrator to formally specify security requirements by using an expressive access control model based on OrBAC [1]. A tool which is proved using the B-Method may therefore implement the so-called *downward* process: the set of algorithms realizing the translation of an OrBAC set of rules into specific devices configurations. Not only the job of administrators is simplified, but they know for certain what security properties are verified at the end.

Acknowledgments. This work has been supported by a grant from the Brittany region of France and by the following projects: POLUX ANR-06-SETIN-012, SEC6 Project, TSI2007-65406-C03-03 E-AEGIS, and CONSOLIDER CSD2007-00004 "ARES".

References

1. Abou el Kalam, A., Baida, R.E., Balbiani, P., Benferhat, S., Cuppens, F., Deswarte, Y., Miège, A., Saurel, C., Trouessin, G.: Organization Based Access Control. In: IEEE 4th Intl. Workshop on Policies for Distributed Systems and Networks, Lake Come, Italy, pp. 120–131 (2003)
2. Abrial, J.R.: The B-Book — Assigning Programs to Meanings. Cambridge University Press, Cambridge (1996)
3. Abrial, J.R., Cansell, D., Méry, D.: Formal Derivation of Spanning Trees Algorithms. In: Bert, D., Bowen, J.P., King, S. (eds.) ZB 2003. LNCS, vol. 2651, pp. 457–476. Springer, Heidelberg (2003)

4. Autrel, F., Cuppens, F., Cuppens-Boulahia, N., Coma, C.: MotOrBAC 2: A security policy tool. In: SAR-SSI 2008, Loctudy, France (2008)
5. Bartal, Y., Mayer, A., Nissim, K., Wool, A.: Firmato: A novel firewall management toolkit. In: IEEE Symposium on Security and Privacy, pp. 17–31 (1999)
6. Benaissa, N., Cansell, D., Méry, D.: Integration of Security Policy into System Modeling. In: Julliand, J., Kouchnarenko, O. (eds.) B 2007. LNCS, vol. 4355, pp. 232–247. Springer, Heidelberg (2006)
7. Capretta, V., Stepien, B., Felty, A., Matwin, S.: Formal correctness of conflict detection for firewalls. In: ACM workshop on Formal methods in security engineering, FMSE 2007, Virginia, USA, pp. 22–30 (2007)
8. Casassa Mont, M., Baldwin, A., Goh, C.: POWER prototype: towards integrated policy-based management. In: Network Operations and Management Symposium, USA, pp. 789–802 (2000)
9. Cuppens, F., Cuppens-Boulahia, N., Ben Ghorbel, M.: High level conflict management strategies in advanced access control models. Electronic Notes in Theoretical Computer Science (ENTCS) 186, 3–26 (2007)
10. Damianou, N., Dulay, N., Lupu, E., Sloman, M.: The ponder policy specification language. In: Sloman, M., Lobo, J., Lupu, E.C. (eds.) POLICY 2001. LNCS, vol. 1995, pp. 18–38. Springer, Heidelberg (2001)
11. Fraer, R.: Minimum Spanning Tree. In: FACIT 1999, pp. 79–114. Springer, Heidelberg (1999)
12. Fu, Z., Wu, S.F., Huang, H., Loh, K., Gong, F., Baldine, I., Xu, C.: IPSec/VPN Security Policy: Correctness, Conflict Detection and Resolution. In: Policy 2001 Workshop, Bristol, UK, pp. 39–56 (2001)
13. Garcia-Alfaro, J., Cuppens, N., Cuppens, F.: Complete Analysis of Configuration Rules to Guarantee Reliable Network Security Policies. International Journal of Information Security 7(2), 103–122 (2008)
14. Jürjens, J.: Secure Systems Development with UML. Springer, New York (2004)
15. Jürjens, J., Schreck, J., Bartmann, P.: Model-based security analysis for mobile communications. In: 30th international conference on Software engineering, Leipzig, Germany, pp. 683–692 (2008)
16. Laborde, R., Kamel, M., Barrere, F., Benzekri, A.: Implementation of a Formal Security Policy Refinement Process in WBEM Architecture. Journal of Network and Systems Management 15(2) (2007)
17. Ioannidis, S., Bellovin, S.M., Ioannidis, J., Keromitis, A.D., Anagnostakis, K., Smith, J.M.: Virtual Private Services: Coordinated Policy Enforcement for Distributed Applications. International Journal of Network Security 4(1), 69–80 (2007)
18. Ponsini, O., Fédèle, C., Kounalis, E.: Rewriting of imperative programs into logical equations. In: Sci. Comput. Program., vol. 54, pp. 363–401. Elsevier North-Holland, Inc., Amsterdam (2005)
19. Preda, S., Cuppens, F., Cuppens-Boulahia, N., Alfaro, J.G., Toutain, L., Elrakaiby, Y.: A Semantic Context Aware Security Policy Deployment. In: ACM Symposium on Information, Computer and Communication Security (ASIACCS 2009), Sydney, Australia (March 2009)
20. Sandhu, R., Coyne, E.J., Feinstein, H.L., Youman, C.E.: Role-Based Access Control Models. IEEE Computer 29(2), 38–47 (1996)
21. Unal, D., Ufuk Çaglayan, M.: Theorem proving for modeling and conflict checking of authorization policies. In: Proceedings of the International Symposium on Computer Networks, ISCN, Istanbul, Turkey (2006)
22. ACI DESIRS project: DÉveloppement de Systèmes Informatiques par Raffinement des contraintes Sécuritaires