A Model for the Analysis of Security Policies in Industrial Networks

Ivan Cibrario Bertolotti[1], Luca Durante[1], Tingting Hu[1,2], Adriano Valenzano[1]
[1] National Research Council of Italy – IEIIT
[2] Politecnico di Torino – DAUIN
c.so Duca degli Abruzzi 24
I-10129 Torino
Italy
www.ieiit.cnr.it
{*ivan.cibrario, luca.durante, tingting.hu, adriano.valenzano*} *@ieiit.cnr.it*

The analysis of security policies designed for ICS and SCADA can benefit significantly from the adoption of automatic/semi-automatic software tools that are able to work at a global (system) level. This implies the availability of a suitable model of the system, which is able to combine the abstractions used in the definition of policies with the access control and right management mechanisms usually present in the real system implementation. This paper introduces a modeling framework based on the Role Based Access Control (RBAC) technique that includes all the elements needed to support different kinds of automatic security analyses such as policy coherence checks and verifications of correct implementation of policies.

Role Based Access Control, Security policy analysis, Security of industrial networks

1. INTRODUCTION

Cyber security of industrial control systems (ICS) and supervision, control and data acquisition systems (SCADA) is a topic whose importance and popularity have been growing rapidly in the recent past, though with a delay of several years with respect to the awareness and sensitivity for security issues in business and office networks. This difference in time mainly depends on both the isolation of ICS and SCADA networks and the widespread use of proprietary technologies in the past, until evident advantages obtainable through remote access, control and management, started pushing towards the full integration of these systems with the office and business networks.

Besides benefits, however, the adoption of standard protocols and open technologies also brought significant disadvantages such as the exposure to cyber threats, which is now affecting the industrial control and SCADA domains too. Unfortunately, as shown by Cheminod et al. (2013), the main functional needs of ICS and SCADA, e.g. the typical real-time communication constraints and maximum availability requirements, prevent the immediate adoption of those security countermeasures already developed and available for the business and office networks.

Most solutions, in fact, rely on a *detect and patch approach*, which is often incompatible with availability, and in many cases they also require a lot of computational resources and communication bandwidth (e.g. to support cryptography), in a context where several devices with reduced computing power have to deal with real-time constraints.

From a different point of view, fortunately, although industrial control and SCADA systems can be very large and complex, they usually exhibit reduced network dynamics, when compared to traditional IT networks. In fact, the set of involved users and protocols is smaller and almost fixed, while the system topology is generally simpler, as pointed out by Cárdenas et al. (2008). This aspect enables the off-line use of security analysis tools, whose efficacy is strongly based on the coherency between the system model, used in the analysis, and the running real system.

This requirement, which is not trivial for rapidly changing systems, looks reasonable and affordable in this context. Tools with these characteristics are not commercially available yet, but some research results are encouraging, and have raised confidence about the feasibility of (semi)automatic security analyses for industrial networked systems.

In this paper we propose a framework conceived to enable the automatic conformance check of security policies against the system configuration. On the one hand, we have security policies described by means of the Role Based Access Control (RBAC) formalism specified by Sandhu et al. (1996); ANSI INCITS 359-2004 (2004); Ferraiolo et al. (2007). On the other hand, we consider a formal description of an ICS or SCADA system, whose conformance against the RBAC security policies has to be checked. In particular, we consider hierarchical RBAC, where the role hierarchy enables the inheritance of permissions. Administrative features, i.e. roles able to grant and revoke membership of users to roles and permissions to roles etc., have not been taken into account since the reduced network dynamics of industrial control and SCADA systems does not justify this additional burden.

RBAC is a well-known formal methodology for the abstract (i.e. implementation independent) description of access control policies, whose goal is providing a comprehensive methodology for the management of users' access rights to the devices and resources of the system. As all access control management systems, tools and formalisms, RBAC neither forbids nor prevents the users from misusing their rights, but it allows to clearly state and know "who can do what", possibly following the principles of both separation of duties and giving each user the minimum power required to perform her/his task.

A lot of scientific work has been done about RBAC the past and its theoretical aspects and extensions (Jayaraman et al. (2011); Sun et al. (2011); Masood et al. (2010); Alberti et al. (2011); Sandhu et al. (1999), and many others), whereas few contributions deal with implementation issues (Bertino et al. (1999); Park et al. (2001); Faden (1999); Karjoth (2000)). In any case, however, the underlying physical system is assumed to be able to *enforce* (i.e. to guarantee) the high-level policies. This is usually obtained by proposing and deploying suitable system software layers or operating system modules carrying out the job as expected, and heavily relying on specific features of the involved system, application or server. For this reason, the scope of proposals appeared in the literature is always narrow: (Workflow Management Systems - Bertino et al. (1999), Web Servers - Park et al. (2001), Solaris UNIX System - Faden (1999), and Java - Karjoth (2000)), and no scalability advantages are leveraged in heterogeneous environments, consisting of many different devices and resources.

Despite the above limitations, this approach is possible only when the system components, e.g. processing nodes, are powerful enough, their operating systems provide suitable features exploitable to this purpose, and no critical functional/performance requirement is violated. This is not the case of many industrial systems that 1) have to satisfy severe requirements, 2) include devices with low computational power, and 3) run special purpose and real-time system software.

A main consequence is that neither dependencies nor relationships can be enforced, between the set of security policies and the running system. In such a situation we regard the abstract set of policies (called *specification* in the following) and the formal model of the running system (*implementation*) without assuming any predefined relationship between them. Relationships are then computed to check whether or not the *implementation* is a refinement of the *specification*. As mentioned before, ICS and SCADA seldom provide embedded security features. Hence, the implementation of access control policies relies on both the physical confinement of devices and proper configuration of active network devices such as switches, firewalls and routers. For this reason, they have to be explicitly taken into account in the modeling framework.

A theoretical framework has been developed both for modeling the RBAC policies and the system and for the consequent consistency analysis, i.e. comparison between what is explicitly allowed on the RBAC side, and what the system permits to do. Here, we are mainly interested in presenting the modeling and analysis tool and its architecture, instead of the theoretical development, thus all concepts and examples will be introduced through the Prolog syntax.

The paper is organized as follows: Section 2 introduces models for both specification and implementation by means of suitable examples, Section 3 briefly summarizes the class model, fully presented in Cibrario Bertolotti et al. (2012). Then, Sections 4 and 5 describe the class system used to specify the RBAC policies and the system implementation, respectively. Last, Section 6 provides the Prolog models of the example introduced in Section 2, and the analysis results. Section 7 contains some concluding remarks.

2. MODELS

In this section both the specification and implementation models are briefly introduced by means of an example we will use throughout the paper.

2.1. RBAC specification

An RBAC specification revolves around the concept of *role*. *Permissions*, i.e. pairs made by an *object*

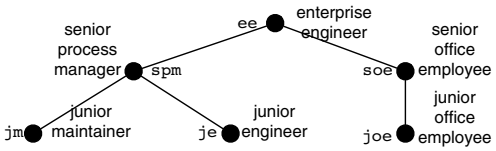

Figure 1: *Example of RBAC roles hierarchy.*

users	roles	permissions	
u_ee	ee		
u_spm	spm	(reach,b_plc)	perm4
u_jm	jm	(reach,b_enterprise)	perm1
u_jm_je		(reach,b_field)	perm3
u_je	je	(reach,b_enterprise)	perm1
u_jm_je		(reach,b_field)	perm3
u_soe	soe	(reach,b_dmz)	perm2
u_joe	joe	(reach,b_enterprise)	perm1

Table 1: *RBAC users, roles and permissions.*

and an *operation* allowed on it, are given to roles, and *users* are assigned to roles. The rationale of this approach is that few roles are usually enough to describe all kinds of jobs in a company, since many employees do the same job. The *role* acts as an intermediary between *users* and *permissions*, thus leading to a more compact descriptions of tasks and people allowed to carry them out. Roles can be further organized in a hierarchical fashion, allowing role inheritance to users and permission inheritance to roles. Figure 1 shows an example of roles hierarchy, whereas Table 1 lists users, roles and permissions respectively. In each row of the table, the two leftmost elements are assignments of users to roles, while the two rightmost cells are permission to role assignments. By convention, users names begin with 'u_', followed by the sequence of role names each user belongs to: in this way, user `u_je` belongs to role `je`, and user `u_jm_je` belongs to both roles `jm` and `je`.

Formally permissions are sets of pairs (operation,object), and each pair is a basic permission. Permissions of Table 1 enable the operation `reach` on rooms, called `b_enterprise`, `b_dmz`, `b_field`, and `b_plc` respectively. In the following, each basic permission will be referred through its id: `perm1`, `perm2`, `perm3`, and `perm4` respectively. Figure 1 and Table 1 provide a complete RBAC specification, and the inheritance semantics propagate role memberships downwards in the tree, e.g. `u_ee` also belongs to all underlying roles in the tree, whereas permissions propagate upwards, e.g. role `spm` also acquires `perm1` and `perm3`, and role `ee` owns `perm1`, `perm2`, `perm3`, and `perm4`.

2.2. System model

Figure 2 depicts the basic structure of an exemplar industrial system. It consists of a *field* network,

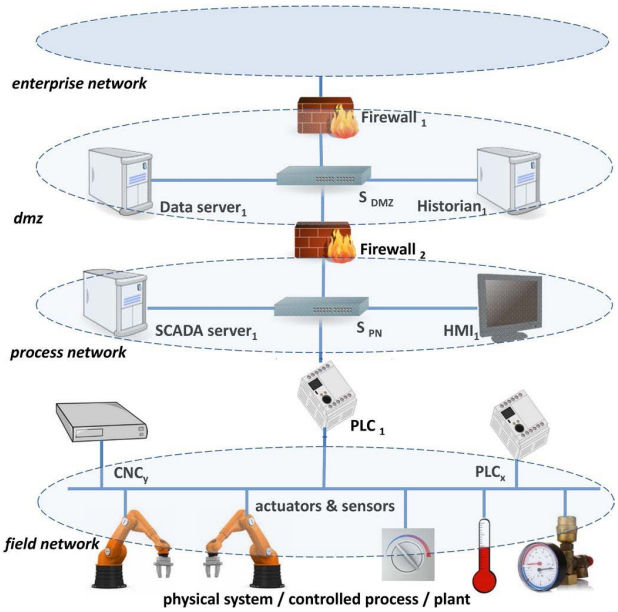

Figure 2: *Exemplar system: network architecture.*

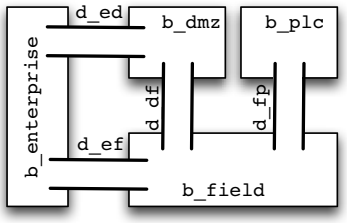

Figure 3: *Exemplar system: physical topology.*

comprising sensors, actuators and secondary PLCs. A primary PLC connects the field network to the Ethernet-based *process* network, which includes a SCADA server and an HMI device. A *demilitarized zone* (DMZ) isolates the process network from the wider *enterprise* network. A data server and a historian are hosted within the DMZ.

Figure 2 does not include some useful information, such as the physical confinement of devices, shown in Figure 3. In the picture, `b_enterprise` is the room (or building) where the nodes belonging to the enterprise network of Figure 2 are confined, `b_dmz` hosts nodes of the dmz network, `b_field` contains elements of both the process and field networks, and `b_plc` is a cabinet to physically protect PLC_1, since it lacks any software-based access control mechanism. To keep the example compact, only aspects concerning the physical access to rooms/buildings were considered. All the other modeling elements, e.g. hosts, accounts etc. will be anyway introduced in Sections 4 and 5.

The system model also contains information not shown in Figures 2 and 3 such as, for example, credentials (e.g. passwords, smart cards, physical

room/building	door	credential
b_enterprise	d_ed	c_d_ed
	d_ef	c_d_ef
b_dmz	d_ed	c_d_ed
	d_df	c_d_df
b_field	d_ef	c_d_ef
	d_df	c_d_df
	d_fp	c_d_fp
b_plc	d_fp	c_d_fp

Table 2: *Exemplar system: physical access controls.*

key, etc.) needed to control the accesses to rooms/buildings and to implement the abstract permissions of Table 1. Table 2 describes each room/building in terms of its doors and credential required to cross a door. For instance, door `d_ed` connects `b_enterprise` to `b_dmz` and vice-versa, and the credential needed to open it is `c_d_ed`.

3. CLASS MODEL

The class model adopted in this paper has been thoroughly discussed in Cibrario Bertolotti et al. (2012) and is informally summarized here. The main goal of the class model is to provide a unified framework, in which both the RBAC policy *specification* and the system *implementation* class hierarchies can be defined. Moreover, the class model is simple enough to be expressed in a first-order logic and ensure that it can be implemented conveniently by means of a logic programming language. In the following, the ISO Prolog language specified in ISO/IEC (1995) and implemented by the SWI Prolog system from University of Amsterdam (2012) has been used as a reference.

3.1. Class Hierarchy

Classes and their fields are declared by means of `class` and `field` facts. For instance, the following facts assert the existence of a class `c` with a field `a`.

```
class(c).
field(c, a).                                    (1)
```

The class–superclass relationship is specified by means of `super` facts. The following example indicates that class `c` has `s` as a superclass:

```
super(c, s).                                    (2)
```

The class system supports two kinds of aggregation: by either *composition* or *containment*. Aggregation by composition allows a class to be defined in terms of other classes. For instance, the following fact denotes that class `d` includes class `c` as a component. Within class `d`, the component is identified by its name `f`:

```
component(d, f, c).                             (3)
```

According to this definition, `f` has the same role as a field name within a structure type definition in other programming languages.

Aggregation by containment generalizes the previously-mentioned aggregation by composition and allows a class to contain a collection of zero or more instances of the component class, instead of exactly one. Aggregation by containment is conveyed by means of the `array` fact, with a similar syntax. This fact denotes that class `d` includes zero or more instances of class `c` and they are collectively denoted as `g`:

```
array(d, g, c).                                 (4)
```

When aggregation (either nested or not) is used, it is no longer possible to refer to fields by a single name. Rather, a *field selector*, consisting of a well-formed sequence of component and field names, must be used. In Prolog, field selectors are represented by lists. For example, `[f, a]` and `[g, 3, a]` are field selectors for class `d`, given the definitions (1)–(4). Namely, the second selector refers to field `a` of instance `3` of aggregation `g`.

At any level of the class hierarchy it is also possible to define default field values, by means of the `default` fact. For instance, the facts:

```
default(c, [a], v).
default(d, [f, a], w).                          (5)
```

assert that the default value of field `a` in class `c` (and all its subclasses) is `v`. This default value is overridden by a finer-grained default defined at the level of class `d`. The second fact asserts that the default value of field `a` in class `c`, when that class is a component of class `d`, is `w` (instead of `v`). In general, a default value overrides another if it is defined deeper in the class hierarchy, regardless of the order in which defaults values are defined.

Multiple `default` facts can be asserted at the same level of the class hierarchy to take advantage of implicit enumeration, performed during the inference process by Prolog, as shown by Nilsson and Małuszyński (1995). For example, if the additional fact

```
default(c, [a], x).                             (6)
```

is asserted, it denotes that the value of field `a` in class `c` can be either `v` (from (5)) or `x`. This capability can profitably be used to easily implement sets as field values, as discussed by Munakata (1992).

This kind of enumeration is called implicit because, even though multiple field values are considered during the inference process and the analysis, the whole set of values that a field may possibly assume is never computed and stored all at once. When desired, set elements can still be enumerated

explicitly and stored into a list, by means of the built-in meta-predicates `bagof` and `setof`.

Starting from these facts, a set of predicates can conveniently assess useful properties of the class hierarchy. Most notably, the goals

```
has_field(C, FS).
has_default(C, FS, D).
```
(7)

succeed when `FS` is a valid field selector for class `C` and when the default value of field `FS` in `C` is `D`, respectively. As usual in Prolog, symbols beginning with a capital letter denote variables, whereas lowercase symbols are constant.

3.2. Object Instantiation

Object instantiation builds on the class model discussed in the previous section. It consists of three steps, two mandatory and one optional.

1. The `object` fact asserts that a certain object exists in the system.
2. The `belongs_to` fact associates an object to a class within the hierarchy defined as shown in Section 3.1.
3. Zero or more `value` facts give a value to the object fields, overriding any default value defined in the class hierarchy.

As for `default` facts, multiple `value` facts can be asserted for the same field selector, with analogous effects.

For example, the following assertions put object `o`, belonging to class `c`, into existence. The value of its field `a` will be `y`, overriding both the default values given in (5) and (6).

```
object(o).
belongs_to(o, c).
value(o, [a], y).
```
(8)

This way of instantiating object is a stark departure from more traditional, imperative programming languages like, for instance, C++. In fact, object instantiation in those languages basically consists of allocating a memory image of the object comprising all its fields and components—unless they are indirectly referenced through pointers—and then running a set of constructors to initialize it.

On the contrary, with the approach proposed in this paper, the instantiation of an object consists of the assertion of a usually small number of facts. These facts put the object into existence, position it within the class hierarchy, and assign non-default values to its fields, while default values are still held in `default` facts at the class level. Although this method is less general than the previous one, because it does not support the execution of generic constructor code, it is still adequate to represent classes and objects involved in an RBAC specification and its implementation. In addition, it provides two practical advantages when there are lots of identical or very similar objects:

1. When most fields of an object retain their default values, the object representation is very compact from the storage *space* point of view, because only non-default values are stored explicitly for individual objects.
2. Under the same conditions, any inference process involving the objects can be made very efficient from the execution *time* point of view, too, as explained in the following.

In order to justify the second advantage, it must be remarked that predicate evaluation performed at the object level involves default values, and hence, shifts from the object to the class level, with high probability. As a consequence, the same predicates will be evaluated a large number of times on the same classes, that is, Prolog entities. As shown, for instance, by Johnson (1993) in the general context of constraint logic programming, even relatively simple and well-established optimizations like *memoization* are very effective under these conditions.

In a class/object hierarchy, the predicate `instance_of` walks through the `belongs_to` and `super` facts to establish whether or not an object is an instance of a certain class or its superclasses. The `has_value` predicate extends `has_default` to work on objects and take `value` facts into consideration. The definition of these predicates is straightforward and is not shown for conciseness.

4. RBAC CLASS HIERARCHY

The RBAC role hierarchy, e.g. the one shown in Figure 1, is well suited to be hosted in an object-oriented hierarchy based on a unified class model specified by means of a formal language. Provided that the class model is general enough to support also a formal description of the system that implements a given set of RBAC policies, this greatly facilitates the analysis, because it provides a unified framework for it.

4.1. RBAC Specification Classes

As shown in Table 3, the class model described in Section 3 readily supports a set of basic RBAC classes derived directly from ANSI INCITS 359-2004 (2004). All of them are subclasses of the `rbac` class and, besides their unique identifier (a Prolog atom), they are also characterized by a textual description,

```
class(rbac).
field(rbac, desc).
default(rbac, [desc], '').

class(role).  super(role, rbac).
field(role, down).

class(user).

class(ua).  super(ua, rbac).
field(ua, user).
field(ua, role).

class(op).
class(obj).

class(op_reach).
super(op_reach, op).

class(perm).  super(perm, rbac).
field(perm, a_perm).

class(pa).  super(pa, rbac).
field(pa, perm).
field(pa, role).
```

Table 3: *Basic RBAC Specification Classes.*

represented by the `desc` field, which is an empty string by default. It is not used during the analysis, but it may assist in results interpretation. RBAC defines a partial order relation on roles, conveyed by the `down` field of the `role` class, which indicates the sub-roles of a given role. Since fields are allowed to hold multiple values, it is possible to specify a set of sub-roles rather than just one.

Users are assigned to roles by means of *user assignment* objects, belonging to the `ua` class. The fields `user` and `role` contain the Prolog identifiers of the user and role to be bound together, respectively. Multiple `ua` objects can refer to the same user or role, in order to support an arbitrary many-to-many relationship. The set of users assigned to a certain role can be easily enumerated by means of the following inference rules, which provide implicit and explicit enumeration, respectively.

```
assigned_user(R, U) :-
  instance_of(UA, ua),
  has_value(UA, [role], R),
  has_value(UA, [user], U).

assigned_users(R, SET_U) :-
  setof(U, assigned_user(R, U), SET_U).
```
(9)

RBAC policies refer to abstract operations (`op`) on abstract objects (`obj`), whose relationship with concrete operations and objects described in the implementation (Section 5) may or may not be fully known in advance. For this reason, the definition of the `op` and `obj` classes shown in Table 3 is incomplete. This approach does not imply any side effect, because the class model supports the incremental definition of classes. A better description on how the relationship between abstract and concrete concepts can be brought into the model will be given in Section 5.3. All abstract operations are subclasses of `op`. In the table, just one of them is shown, namely, the `op_reach` class. Instances of this class denote generic *reachability* operations, conveying the meaning that it is possible to reach a certain `obj` either physically or logically, for instance, to perform an operation on it.

The definition of RBAC *permissions* (`perm`) is more involved, because a single permission is a set of pairs (π, β), where π is an `op` and β is an `obj`. Each value of field `a_perm` in a `perm` is a pair, represented directly as a two-element Prolog list. A `perm` holds the whole set of pairs as multiple values of its `a_perm` field. A *permission assignment* (`pa`) object binds a permission to a role by means of its `perm` and `role` fields. As for user assignments, multiple `pa` objects may refer to the same permission or role to build a many-to-many relationship. The `assigned_perm` and `assigned_perms` predicates are defined in the same way as their counterparts for user assignments. On the other hand, the following two predicates further expand permissions and return individual (π, β) pairs:

```
assigned_a_perm(R, A_PERM) :-
  assigned_perm(R, P),
  has_value(P, [a_perm], A_PERM).

assigned_a_perms(R, SET_A_PERM) :-
  setof(A_PERM,
    assigned_a_perm(R, A_PERM),
    SET_A_PERM).
```
(10)

The operations `OP` enabled by a certain permission `P` and the objects `OBJ` affected by it can be enumerated by means of:

```
op_enabled(P, OP) :-
  instance_of(P, perm),
  has_value(P, [a_perm], [OP, _]).

obj_affected(P, OBJ) :-
  instance_of(P, perm),
  has_value(P, [a_perm], [_, OBJ]).
```
(11)

The partial order relation on roles conveyed by the `down` field can be used to further generalize the `assigned_user`, `assigned_perm`, and `assigned_a_perm` predicates. For instance, assuming that a role inherits user assignments from its (grand)parents, the set of *authorized* users for a certain role can be determined by means of:

```
authorized_user(R, U) :-
  instance_of(R, role),
  role_ge(RP, R),
  assigned_user(RP, U).
```
(12)

where `role_ge(RP, R)` determines the partial order relationship between `RP` and `R` based on the transitive closure of the `down` field. In other words, `role_ge(RP, R)` succeeds iff role `R` can be reached from role `RP` by following zero or more `down` links. The `authorized_perm` and `authorized_a_perm` predicates can be derived in a similar way from

```
class(impl).
field(impl, desc).
default(impl, [desc], '').

class(host).
super(host, impl).

class(account).
super(account, impl).
field(account, user).
field(account, host).

class(group).
super(group, impl).
field(group, role).
field(group, host).
```

Table 4: *Basic Implementation Classes.*

their simpler counterparts. The procedure to develop predicates for explicit enumeration from their implicit counterparts involves the use of `setof` and is the same as it was shown in (9).

5. IMPLEMENTATION MODEL

The complement of the system *specification*, expressed by a set of RBAC policies defined as discussed in Section 4, is its *implementation*. In order to perform the analysis within a unified framework, the same class model has been used for both. All implementation-related classes are derived from the base class `impl`.

5.1. Hosts, Accounts, and Groups

Table 4 shows the classes used to describe the most basic elements of the implementation, that is, hosts, accounts, and groups. Accounts, represented by `account` objects, allow access by a certain `user` to a certain `host`. Similarly, groups are represented by `group` objects and allow a `role` to be carried out on a certain `host`. After those basic implementation elements are known for a given system, it becomes possible to verify that the mapping between the RBAC policy specification and its implementation is appropriate. For instance, it is possible to check the implementation does not allow users to perform any operation that is not allowed by RBAC policies pertaining to their role.

This can be done, first of all, by defining the notion of accounts assignment to groups—denoted as `aa`—as the natural join (Abiteboul et al. (1995)) between pairs (u, h) coming from `account` objects and pairs (r, h) coming from `group` objects. The result is a set of tuples (u, r, h) that can be enumerated by the following predicate:

```
aa(U, R, H) :-
  instance_of(A, account),
  has_value(A, [user], U),
  has_value(A, [host], H),
  instance_of(G, group),
  has_value(G, [role], R),
  has_value(G, [host], H).
```
(13)

```
class(obj_room).  super(obj_room, obj).
field(obj_room, op).

class(door).  super(door, impl).
field(door, room_1).
field(door, room_2).

class(op_cross).  super(op_cross, op).
field(op_cross, door).
field(op_cross, cred).

class(cred).  super(cred, impl).
```

Table 5: *Room, Door, and Credential Classes.*

The inference rule enumerates `account` and `group` objects (by means of `instance_of`) and then unifies them based on their `host` field, held in variable `H`. Those definitions allow a first, very basic consistency check to be made. Namely, it is possible to verify if the system implementation is compatible with the RBAC specification. If AA is the set of account assignments to groups, enumerated by `aa`, a necessary compatibility condition is:

$$\forall (u, r, h) \in \mathrm{AA} \Rightarrow u \in \mathrm{authorized\_users}(r) \ . \quad (14)$$

The condition can be translated into Prolog in a straightforward way as:

```
validate_aa :-
  forall(aa(U, R, H),
    authorized_user(R, U)).
```
(15)

The predicate shown above has been kept as simple as possible for clarity and to better highlight the convenient, strong relationship between the abstract formulation of the check (14) and its programming language notation (15). It simply fails if the compatibility condition is not met. Alternate, more complex, implementations can provide informative error messages about the reasons of incompatibility.

5.2. Rooms, Doors, and Credentials

A second set of implementation-related classes concerns *rooms*, *doors*, and *credentials*. They are shown in Table 5. The concept of room, represented by the `obj_room` class, goes beyond a physical place and, more generically, denotes any physical or logical location in the system when one or more operations can be performed by an agent, for instance, a user. The set of operations allowed in a room are stored in its multi-valued field `op`. One of these operations is the `op_cross` operation, which makes the agent enter that room from another one.

This operations involves crossing a door, provided the agent holds an appropriate credential. Those items of information are stored in fields `door` and `cred`, respectively. As before, both concepts may be correlated to some physical counterpart, but this is not required. Hence, for instance, the credential required to cross a physical door can be a physical

key, a password, or a smartcard, depending on the kind of lock installed on the door.

From this ground, it is possible to define the concrete, implementation-related counterpart of the abstract reachability relation discussed in Section 4. In its simplest form, it is conveyed by the `visit_with_cred` predicate shown in Table 6.

The goal `visit_with_cred(A, B, CredBAG, Path)` succeeds if it is possible to go from room `A` to room `B` by making use of the credential contained in the list `CredBAG`. The path to be followed is stored in `Path`. `Path` is a list in which each element but the last is a triple `[Room, Cred, Door]` that indicates a movement from room `Room` through door `Door` using credential `Cred`. The last element of the list, the final destination room, just contains the room identifier.

In the first inference rule, `visit_with_cred` is defined in terms of `visit_with_cred_c`, by adding an extra argument in third position. This argument is propagated and extended through the recursive definition of `visit_with_cred_c` by means of the variable `Visited`. As its name suggests, it holds the set of visited rooms. It is consulted before extending the current path in order to avoid non-termination due to cycles in the room/door topology.

The first inference rule defined for `visit_with_cred_c` covers the base case of the recursion, in which the source and destination door are the same. Informally speaking, the second inference rule calculates a path as follows:

- It looks for a door `D` that connects the current room `A` with another room `C`, which has not already been visited (according to the contents of the `Visited` variable).
- For all operations `OP` defined in room `C`, it selects the operations that are instances of `op_cross`, referring to door `D` and for which the agent holds the appropriate credential `Cred`.
- If all the previous conditions are met, `C` is a candidate to be part of the path from room `A` to room `B`. In this case, `visit_with_cred_c` is evaluated recursively to calculate the path from `C` to `B`. The other arguments are udpated as appropriate.
- If the sub-goal succeeds, the path from `A` to `B` is defined as `[A, Cred, D]` (leaving room `A` through door `D` using credential `Cred`) followed by the `Path` determined by the sub-goal.

5.3. Bridge Classes

One important issue to be addressed in a unified class system, supporting both an RBAC specification and the corresponding system implementation, is how to correlate entities defined and referenced by them. In the subset of the class system presented in this paper, the three entities shared between specification and implementation are users (class `user`), operations (`op`), and objects (`obj`). For what concerns objects, the discussion will focus on rooms. On the one hand, a main requirement is the possibility of establishing an *a priori* association between entities, when it is known in advance. On the other hand, establishing an association either *during* or *after* the analysis, based on (partial) analysis results, must also be possible.

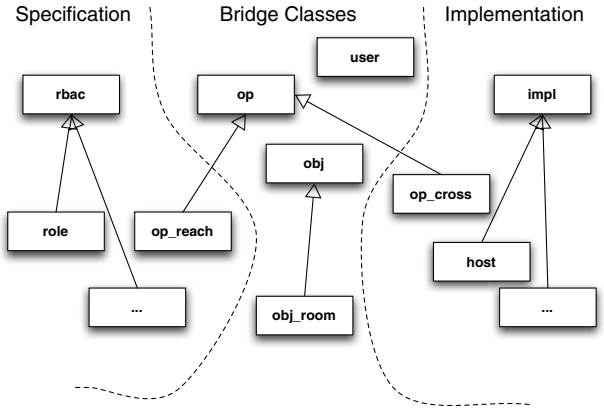

Figure 4: *Bridge Classes between Specification and Implementation.*

As shown in Figure 4, this issue has been solved by introducing so-called *bridge* classes, which belong to neither the `rbac` nor the `impl` class hierarchy. Rather, they are direct descendants of the class system's root class. In this way, when the association is likely known in advance, as it happens for instance for rooms and users, a single class (`obj_room` and `user`, respectively) is defined. Then, its elements are referenced from both "sides" of the model and the mapping is established by name.

If the association must be determined during or after the analysis instead, a variant of the same approach can be followed. First, an abstract bridge class for the objects to be correlated is defined. Then, two separate subclasses are defined, one to be used in the RBAC specification and the other in the implementation. With this approach, the specification and the implementation will use and refer to different objects, belonging to those two subclasses, with no correlation implied in advance. At the same time, the presence of the common superclass is useful to efficiently identify which objects may be correlated at a later time.

In the class system described in this paper, the `op` class is a bridge class, which is the common ancestor of all operations introduced in the system.

```
visit_with_cred(A, B, CredBAG, Path) :-
        visit_with_cred_c(A, B, [A], CredBAG, Path).

visit_with_cred_c(A, A, _, _, [[A]]).
visit_with_cred_c(A, B, Visited, CredBAG, [[A, Cred, D] | Path]) :-
        instance_of(D, door),
        ((has_value(D, [room_1], A), has_value(D, [room_2], C));
        (has_value(D, [room_2], A), has_value(D, [room_1], C))),
        \+ member(C, Visited),
        has_value(C, [op], OP),      % Scan C's operations
            instance_of(OP, op_cross),   % Select op_cross operations
        has_value(OP, [door], D),    % Check D is the right door for OP
        has_value(OP, [cred], Cred), % Cred is the credential we need
        member(Cred, CredBAG),       % Check that Cred is in our CredBAG
        % C is a candidate for the next hop.  CredBAG falls through
        visit_with_cred_c(C, B, [C | Visited], CredBAG, Path).
```

Table 6: *Concrete Counterpart of RBAC Reachability.*

For what concerns reachability, the specification and the implementation operate at two different levels of abstraction, and hence, distinct subclasses (`op_reach` and `op_cross`) have been defined.

- As discussed in Section 4.1, the RBAC specification abstractly specifies reachability in a permission as the possibility of *reaching* a certain `obj`, most notably a room. The objects used to express this concept of reachability are instances of the `op_reach` class.
- On the other hand, as shown in Section 5.2, the system implementation introduces a more concrete concept of reachability. It is conveyed by stating how doors connect rooms together and under which conditions (related to the availability of a certain credential) they can be crossed by an agent. The class `op_cross` has been defined accordingly.

6. EXAMPLE

In this section the example shown in Figures 1, 2, and 3, and Tables 1 and 2 is formalized.

6.1. RBAC Specification

Table 7 contains the RBAC hierarchical role specification for the example system that can be derived, for instance, from company policies shown in Figure 1. The table also contains assignments of users to roles of Table 1.

The ability of a user to reach certain places (rooms) of the system depends on its roles in the system. Table 8 shows the permissions assignment of Table 1, here implemented by `pa1` through `pa7`: both junior maintainers and junior engineers are allowed to reach the enterprise and the field networks. The junior office employee is allowed to reach the enterprise network only, whereas the senior office employee is granted access to the DMZ as well. The senior process manager is allowed to physically reach PLC_1. In addition to the explicit permission assignments mentioned here, roles also inherit permissions from other roles, according to the role hierarchy shown in Figure 1, and specified in Table 7.

```
object(jm).  belongs_to(jm, role).
value(jm, [desc], 'Junior maintainer').

object(je).  belongs_to(je, role).
value(je, [desc], 'Junior engineer').

object(spm).  belongs_to(spm, role).
value(spm, [desc], 'Senior process manager').
value(spm, [down], jm).
value(spm, [down], je).

object(joe).  belongs_to(joe, role).
value(joe, [desc], 'Junior office employee').

object(soe).  belongs_to(soe, role).
value(soe, [desc], 'Senior office employee').
value(soe, [down], joe).

object(ee).  belongs_to(ee, role).
value(ee, [desc], 'Enterprise engineer').
value(ee, [down], spm).
value(ee, [down], soe).

object(u_jm).  belongs_to(u_jm, user).
object(ua1).  belongs_to(ua1, ua).
value(ua1, [user], u_jm).
value(ua1, [role], jm).

object(u_jm_je).  belongs_to(u_jm_je, user).
object(ua2).  belongs_to(ua2, ua).
value(ua2, [user], u_jm_je).
value(ua2, [role], jm).
value(ua2, [role], je).
```

Table 7: *RBAC User and Role Specification.*

6.2. Implementation

The implementation model of the exemplar system of Figure 3 and Table 2 is partially shown in Table 9, for what concerns rooms and doors only. As discussed in Section 5.2, in the implementation model reachability is expressed as the ability to cross a door, connecting two rooms, under the constraint of holding an appropriate credential. For instance, door `d_ed` connects the two rooms `b_enterprise` and `b_dmz` (corresponding to the enterprise network and DMZ). It is possible to cross this door using

```
object(reach).  belongs_to(reach, op_reach).

object(perm1).  belongs_to(perm1, perm).
value(perm1, [a_perm], [reach, b_enterprise]).

object(perm2).  belongs_to(perm2, perm).
value(perm2, [a_perm], [reach, b_dmz]).

object(perm3).  belongs_to(perm3, perm).
value(perm3, [a_perm], [reach, b_field]).

object(perm4).  belongs_to(perm4, perm).
value(perm4, [a_perm], [reach, b_plc]).

object(pa1).  belongs_to(pa1, pa).
value(pa1, [perm], perm1).
value(pa1, [role], jm).

object(pa2).  belongs_to(pa2, pa).
value(pa2, [perm], perm3).
value(pa2, [role], jm).

object(pa3).  belongs_to(pa3, pa).
value(pa3, [perm], perm1).
value(pa3, [role], je).

object(pa4).  belongs_to(pa4, pa).
value(pa4, [perm], perm3).
value(pa4, [role], je).

object(pa5).  belongs_to(pa5, pa).
value(pa5, [perm], perm1).
value(pa5, [role], joe).

object(pa6).  belongs_to(pa6, pa).
value(pa6, [perm], perm2).
value(pa6, [role], soe).

object(pa7).  belongs_to(pa7, pa).
value(pa7, [perm], perm4).
value(pa7, [role], spm).
```

Table 8: *RBAC Permission Assignment.*

the `cross_1` operation, if the agent holds credential `c_d_ed`.

Table 10 summarizes the initial user states. As for the other aspects of the implementation, the information shown in this table is expressed by means of Prolog facts in the real model. However, it is presented here in tabular form for brevity. For each user, the initial location in the system and the set of available credentials are specified.

6.3. Preliminary Results

A first set of analysis results concerns the mutual *consistency* between the security policies given in the system specification and the real system behavior, as derived from its implementation model. Starting from the (partial) specification and implementation models discussed in this paper it is possible, for instance, to determine whether or not the abstract room reachability properties contained in the specification have been implemented correctly.

By means of the `authorized_users` predicate we can, first of all, determine the users associated with the `spm` (senior process manager) role, and then the

```
object(b_enterprise).  belongs_to(b_enterprise, obj_room).
value(b_enterprise, [desc], 'Enterprise').
value(b_enterprise, [op], cross_1).
value(b_enterprise, [op], cross_2).

object(cross_1).  belongs_to(cross_1, op_cross).
value(cross_1, [door], d_ed).
value(cross_1, [cred], c_d_ed).

object(c_d_ed).  belongs_to(c_d_ed, cred).

object(cross_2).  belongs_to(cross_2, op_cross).
value(cross_2, [door], d_ef).
value(cross_2, [cred], c_d_ef).

object(c_d_ef).  belongs_to(c_d_ef, cred).

object(b_dmz).  belongs_to(b_dmz, obj_room).
value(b_dmz, [desc], 'DMZ').
value(b_dmz, [op], cross_1).
value(b_dmz, [op], cross_3).

object(cross_3).  belongs_to(cross_3, op_cross).
value(cross_3, [door], d_df).
value(cross_3, [cred], c_d_df).

object(c_d_df).  belongs_to(c_d_df, cred).

object(b_field).  belongs_to(b_field, obj_room).
value(b_field, [desc], 'Field').
value(b_field, [op], cross_2).
value(b_field, [op], cross_3).
value(b_field, [op], cross_4).

object(cross_4).  belongs_to(cross_4, op_cross).
value(cross_4, [door], d_fp).
value(cross_4, [cred], c_d_fp).

object(c_d_fp).  belongs_to(c_d_fp, cred).

object(b_plc).  belongs_to(b_plc, obj_room).
value(b_plc, [desc], 'PLC').
value(b_plc, [op], cross_4).

object(d_ed).  belongs_to(d_ed, door).
value(d_ed, [room_1], b_enterprise).
value(d_ed, [room_2], b_dmz).

object(d_ef).  belongs_to(d_ef, door).
value(d_ef, [room_1], b_enterprise).
value(d_ef, [room_2], b_field).

object(d_df).  belongs_to(d_df, door).
value(d_df, [room_1], b_dmz).
value(d_df, [room_2], b_field).

object(d_fp).  belongs_to(d_fp, door).
value(d_fp, [room_1], b_field).
value(d_fp, [room_2], b_plc).
```

Table 9: *System Implementation (partial).*

set of rooms reachable by them, with the queries:

```
?- authorized_users(spm, U).
U = [u_ee, u_spm] ;

?- authorized_a_perms(spm, AP).                (16)
AP = [[reach, b_enterprise],
      [reach, b_field],
      [reach, b_plc]].
```

The result of the queries is that users belonging to the `spm` role, according to the policy specification, should be able to reach all rooms of the system, except the DMZ (`b_dmz`). On the other hand, it is possible to determine what the `u_spm` user is really allowed to do

user	room	credentials
u_jm	b_enterprise	c_d_ef
u_je	b_enterprise	c_d_ef
u_spm	b_enterprise	c_d_ef, c_d_fp, c_d_df
u_jm_je	b_enterprise	c_d_ef
u_joe	b_enterprise	—
u_soe	b_enterprise	c_d_ed
u_ee	b_enterprise	c_d_ed, c_d_ef, c_d_df, d_d_fp

Table 10: *Initial user state (in tabular form) with an extra, incorrect credential (underlined).*

in the system by querying the implementation model:

```
?- visit_with_cred(b_enterprise,
 ROOM, [c_d_ef, c_d_fp, c_d_df], PATH).


ROOM = b_enterprise
PATH = [[b_enterprise]] ;

ROOM = b_field
PATH = [[b_enterprise, c_d_ef, d_ef],
        [b_field]] ;                          (17)

ROOM = b_dmz
PATH = [[b_enterprise, c_d_ef, d_ef],
        [b_field, c_d_df, d_df],
        [b_dmz]] ;

ROOM = b_plc
PATH = [[b_enterprise, c_d_ef, d_ef],
        [b_field, c_d_fp, d_fp],
        [b_plc]] ;
```

In the previous query, the initial location of user u_spm (b_enterprise) and his set of credentials (c_d_ef, c_d_df, and c_d_fp) have been derived from Table 10.

In this simple example, the conformance of the implementation (17) with respect to the specification (16) is *not* verified, because user u_spm can indeed reach the room b_dmz. It is also worth mentioning that the analysis performed on the implementation (17) provides additional information about implementation-specific aspects such as, for instance, the path followed to reach a certain room and the credentials used to that purpose.

In particular, from (17) it is easy to deduce that the issue likely comes from credential c_d_df, since u_spm is allowed to enter b_dmz because of it. By repeating the analysis after removing that credential, the result is:

```
?- visit_with_cred(b_enterprise,
 ROOM, [c_d_ef, c_d_fp], PATH).


ROOM = b_enterprise,
PATH = [[b_enterprise]] ;

ROOM = b_field,                               (18)
PATH = [[b_enterprise, c_d_ef, d_ef],
        [b_field]] ;

ROOM = b_plc,
PATH = [[b_enterprise, c_d_ef, d_ef],
        [b_field, c_d_fp, d_fp],
        [b_plc]] ;
```

The conformance of the implementation (18) with respect to the specification (16) has now been verified.

7. CONCLUSIONS

The automatic analysis of security policies in industrial distributed systems has to be based on a suitable modeling framework, which should be able to conjugate the high-level description of policy characteristics with the low-level access control mechanisms adopted in the system processing nodes and communication networks. The Role Based Access Control (RBAC) modeling technique is well suited to cope with the description of security policies with a high degree of abstraction but was not initially conceived to deal with many aspects and details of real system implementations.

This paper has introduced a modeling framework that, in our views, is able to bridge the semantic gap between the RBAC formal description of policies and a fine-grained model of the actual system implementation with particular attention to its raw access control mechanisms. The model contains all the elements that we think are needed to provide an exhaustive picture of the real system from the security point of view and is able to allow different kinds of automatic analysis by means of suitable software tools.

Our work is far from being concluded and, from this point of view, the definition of this modeling framework has to be considered as a preliminary but necessary step. Lessons learned from our past experience confirm that, on the one hand, there is a growing need in the ICS and SCADA areas for tools that are able to carry out automatic security analyses at a global (system) level. On the other hand, such a kind of solutions have to be based on abstract models which are able to capture several fine-grained characteristics of the system implementation

in order to produce useful results from the analysis itself.

ACKNOWLEDGMENT

This work was partially supported by the National Research Council of Italy in the framework project “Factory of the Future”, GECKO “Generic Evolutionary Control Knowledge-based mOdule” project.

REFERENCES

Abiteboul, S., R. Hull, and V. Vianu (1995). *Foundations of Databases*. Addison-Wesley.

Alberti, F., A. Armando, and S. Ranise (2011). Efficient symbolic automated analysis of administrative attribute-based rbac-policies. In *Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security*, ASIACCS ’11, New York, NY, USA, pp. 165–175. ACM.

ANSI INCITS 359-2004 (2004). *Role Based Access Control*. ANSI INCITS 359-2004.

Bertino, E., E. Ferrari, and V. Atluri (1999, February). The specification and enforcement of authorization constraints in workflow management systems. *ACM Trans. Inf. Syst. Secur. 2*(1), 65–104.

Cárdenas, A. A., S. Amin, and S. Sastry (2008). Research Challenges for the Security of Control Systems. In *Proc. 3rd USENIX Workshop on Hot Topics in Security*, pp. 6:1–6:6.

Cheminod, M., L. Durante, and A. Valenzano (2013). Review of security issues in industrial networks. *IEEE Transactions on Industrial Informatics 9*(1), 277–293.

Cibrario Bertolotti, I., L. Durante, T. Hu, and A. Valenzano (2012, October). A unified class model for checking security policies in ICT infrastructures. In *Proc. 1st IEEE AESS European Conference on Satellite Telecommunications (ESTEL)*, pp. 1–6.

Faden, G. (1999). Rbac in unix administration. In *Proceedings of the fourth ACM workshop on Role-based access control*, RBAC ’99, New York, NY, USA, pp. 95–101. ACM.

Ferraiolo, D. F., D. R. Kuhn, and R. Chandramouli (2007). *Role-Based Access Control* (2nd ed.). Information Security and Privacy Series. Norwood, MA 02062, USA: Artech House.

ISO/IEC (1995). *ISO/IEC 13211-1 – Information technology – Programming languages – Prolog – Part 1: General core*. ISO/IEC.

Jayaraman, K., V. Ganesh, M. Tripunitara, M. Rinard, and S. Chapin (2011). Automatic error finding in access-control policies. In *Proceedings of the 18th ACM conference on Computer and communications security*, CCS ’11, New York, NY, USA, pp. 163–174. ACM.

Johnson, M. (1993). Memoization in constraint logic programming. In *Proc. 1st International Workshop on Principles and Practice of Constraint Programming*.

Karjoth, G. (2000). An operational semantics of java 2 access control. *Computer Security Foundations Workshop, IEEE 0*, 224.

Masood, A., A. Ghafoor, and A. Mathur (2010). Conformance testing of temporal role-based access control systems. *IEEE Transactions on Dependable and Secure Computing 7*(2), 144–158.

Munakata, T. (1992, March). Notes on implementing sets in Prolog. *Commun. ACM 35*(3), 112–120.

Nilsson, U. and J. Małuszyński (1995). *Logic, Programming and Prolog* (2nd ed.). John Wiley & Sons Ltd.

Park, J. S., R. Sandhu, and G.-J. Ahn (2001, February). Role-based access control on the web. *ACM Trans. Inf. Syst. Secur. 4*(1), 37–71.

Sandhu, R., V. Bhamidipati, and Q. Munawer (1999, February). The arbac97 model for role-based administration of roles. *ACM Trans. Inf. Syst. Secur. 2*(1), 105–135.

Sandhu, R. S., E. J. Coyne, H. L. Feinstein, and C. E. Youman (1996). Role-Based Access Control Models. *IEEE Computer 29*(2), 39–47.

Sun, Y., Q. Wang, N. Li, E. Bertino, and M. Atallah (2011, November). On the complexity of authorization in rbac under qualification and security constraints. *IEEE Trans. Dependable Secur. Comput. 8*(6), 883–897.

University of Amsterdam (2012). *SWI-Prolog 6.0 Reference Manual*. University of Amsterdam.