URL: http://www.elsevier.nl/locate/entcs/volume51.html 11 pages

Graph Transformations for the Specification of Access Control Policies

Manuel Koch[1], Luigi V. Mancini[2] and Francesco Parisi-Presicce[2,3]

[1] *PSI AG, Berlin, Germany*
[2] *Univ. di Roma La Sapienza, Roma, Italy*
[3] *George Mason Univ., Fairfax, VA, USA*

Abstract

Graph Transformations provide a uniform and precise framework for the specification of access control policies. States are represented by graphs and state transitions by graph transformations. A policy is formalized by four components: a type graph, positive and negative constraints and a set of rules. This formalism allows the detailed comparison of different policy models and the precise description of the evolution of a policy and of the integration of policies.

Key words: security, access control, rule-base specifications, evolution.

1 Introduction

Access Control (AC) protects the computing system against unauthorized access to or modification of information, whether in storage, processing or transit. In this respect, AC is concerned with determining the legitimate activities of users of a computing system [San94]. Several models have been proposed to address the AC requirements of computing applications. Traditional AC models are broadly classified as Discretionary Access Control (DAC) [San94], and Mandatory Access Control (MAC) [San93] models.

In DAC models, the access authorization rules are specified for each pair (subject, object) in the computing system. A subject can be a user, a group or a process that acts on behalf of other subjects. If a subject is the owner of object O, the subject is authorized to grant or revoke access rights on O to other subjects at his discretion. The DAC models have an inherent flexibility

[1] Email: mkoch@psi.de
[2] Email: lv.mancini@dsi.uniroma1.it
[3] Email: fparisi@ise.gmu.edu

and are the most widely used in commercial systems, e.g. in the Unix operating system. However, these models do not provide high security assurance and suffer from Trojan Hoarse attacks. For example, if the authorized user access object O executing a program containing a Trojan Hoarse, O could be copied maliciously into another object with a lower autorization, which results in allowing access to a copy of O to a user who does not have access to the original O. Most of the existing implementation of DAC follow the HRU model [HRU]. This model is based on an access control matrix where each element specifies the access rights of a particular subject for accessing a particular object in the system. Safety in HRU is in general undecidable. The basic safety problem is to determine whether there exists a reachable state in which a given subject possesses a given authorization that it did not previously possess.

The MAC models appear more rigid, all subjects and objects are classified based on predefined security levels that are used during the authorization decision. The MAC models focus on the problem of controlling and enforcing information flow, which is not addressed by DAC models. In MAC the access control policy is enforced by the system and cannot be overridden by a user or a compromised application. For example, to ensure information secrecy in military applications, the MAC model is implemented using a multilevel security restriction that employs no read-up and no write-down rules. These rules are designed to ensure that information does not flow from a higher security level to a lower security level even if a compromised application at higher security level contains a malicious Trojan Hoarse. Despite offering these attractive properties of containment, MAC models have not been used outside of the classified information processing area. Systems with MAC features have no longer supported standard applications or management tools, and they became much more complicated to manage and use than those based on DAC. Moreover, MAC systems implement a form of containment that is too strong and restrictive, that is, the standard applications often can no longer function normally.

New models such as Role-Based Access Control (RBAC) [San98,SFK00] have been proposed to address the security requirements of a wider range of applications. In particular, RBAC models introduce the new concept of role to organize users and access privileges, where roles represent organizational responsibilities of the users of the information system. RBAC models provide several well-recognized advantages. They have been shown to be "policy neutral", meaning that using the RBAC features a wide range of security models can be expressed, including traditonal DAC and MAC. Security administration can also be simplified by the use of role. For example, if a user is assigned a new task within the organization, the user is revoked from the old role and assigned to the new one, whereas in other models, the users's old permissions would have to be individually removed, and new permissions would have to be granted.

Given the variety of different AC models defined in the literature, an im-

portant issue is how to evaluate and compare the features of all these models in terms of expressive power and complexity of their implementation. Hence there seems to be a need for the development of a formal framework to specify and compare the features of different AC models.

In particular, this paper addresses the specification of evolution, transition and integration of any AC models using the formal framework of graph trasformation. The evolution of an AC model consists of changing the AC rules and constraints of an existing model. The changes have to preserve coherence in the sense that the constraints are not violated by the modified functionality. Similar problems occur for the transition from one AC policy to a new one. Besides the evolution and the transition of AC models, the integration of existing AC policies forms the third mechanism for producing a new policy. The integration of policies is relatively simple as long as there are no conflicts in the access allowed by different policies. If there are conflicts, they can be studied and reconciled more easily if the different policies are described within a uniform formal framework.

The major goal of our ongoing work is to describe in a systematic way the evolution of different AC policies, the integration of diverse AC policies and the transition from one policy to another one. A unique formal framework is proposed to tackle these two problems. This paper uses graph transformations to specify a RBAC model.

We are aware of two recent articles ([BDS00,BCFP01]) on putting together AC policies. In the first one, an algebra of models is defined using a standard language of expressions, whose semantics is based on triples <subject, object, rights> and the fact that rules are Horn clauses. We focus instead on graph rules and on the formal tools available to assist in the process of evolving and integrating policies to guarantee a coherent model.

2 Security Policy Framework

This section introduces the framework for the specification of AC policies based on graph transformations. The framework is called *security policy framework* and consists of four components: The first component is a type graph that provides the type information of the AC policy [CELP96]. The second component is a set of graph rules specifying the policy rules that generate the graphs representing the states of the system accepted by the AC policy. For some AC policies, it is meaningful to restrict the set of system graphs constructed by the graph rules, since not all of them represent valid states. Therefore, a security policy framework contains also two sets of *constraints* that specify graphs that shall not be contained in any system graph (*negative constraints*) and graphs that must be explicitly constructed as parts of a system graph (*positive constraints*). In the actual implementation of an AC policy, the constraints are redundant since the only acceptable states are those explicitly built by the implemented rules. But when developing an

AC policy through successive refinement steps, or when comparing different policies, or when trying to predict the behavior of the policy obtained by integrating two different ones, it is useful to have the additional information provided by the constraints. Furthermore, it is usually difficult to extract negative information from "constructive" rules. Positive and negative constraints can be considered as formal documentation of the initial requirements and the development process of rules.

In the sequel, we fix a type graph TG and all graphs and graph morphisms are supposed to be typed in TG.

Graph rules $p : (L \stackrel{r}{\to} R, A(p))$ are specified by a rule name p, a partial graph morphism r and a set of negative application conditions (NAC) $A(p) = \{L \to N_i : i \in I\}$ containing total graph morphisms with source L [HW95]. For the derivation of graphs by graph rules, the Single-Pushout approach is chosen because of its (in the field of AC policies) desired property of automatic deletion of dangling edges.

Both positive and negative constraints are formally specified by morphisms. Only their semantics distinguishes them.

Definition 2.1 [Negative and positive constraints] A *constraint* (positive or negative) is given by a total graph morphism $c : X \to Y$. A graph G *satisfies* a positive (negative) constraint c if for each total injective graph morphism $p : X \to G$ there exists (does not exist) a total injective graph morphism $q : Y \to G$ such that $X \stackrel{c}{\to} Y \stackrel{q}{\to} G = X \stackrel{p}{\to} G$.

Definition 2.2 [Security Policy Framework] A *security policy framework*, or just *framework*, is a tuple $SP = (TG, (P, r_P), Pos, Neg)$, where TG is a type graph, the pair (P, r_P) consists of a set of rule names and a total mapping $r_P : P \to |\mathbf{Rule}(TG)|$ [4], Pos is a set of positive and Neg is a set of negative constraints.

Example 2.3 [Role-Based Access Control] The example presents the security policy framework for a variant on the Role-based Access Control (RBAC) model [San98,OSM00]. This model considers several user roles, or just roles, and several administrator roles responsible for the user-role assignment. Both roles and administrative roles are ordered in a hierarchy given by a partial order. In Fig. 1 an example of an administrative role hierarchy (on the left-hand side) and a role hierarchy (on the right-hand side) is shown, where the hierarchies are given by graphs. Roles as well as administrative roles are given by nodes of type r and ar, respectively, and edges between roles show the inheritance relation.

Typically, a role *Head of a Division* higher in the hierarchy than the role *Head of a Department* inherits all the permissions associated with a *Head of a*

[4] The category **Rule**(TG) has as objects all pairs (r, A), where $r : L \to R$ is a partial graph morphism and A is a set of total graph morphisms with source L.

Department. A user *Smith* assigned to a role *Head of a Department* is granted all the permissions associated with the role.

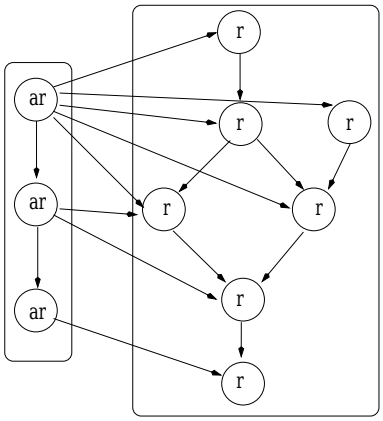

Fig. 1. *Administrator role hierarchy (left) and role hierarchy (right).*

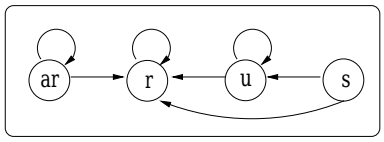

Fig. 2. *The type graph for the RBAC model.*

Graphs generalize the notion of a partial order, and by defining specific rules one could restrict the graph structure to any role hierarchy. Additionally, Fig. 1 shows edges between administrative roles and user roles, representing the authorization to modify the user role by the administrative role. The set of roles reachable by such edges from an administrative role is the *range* of the administrative role. For instance, the range of the upper administrative role is given by the upper five roles, whereas the lower administrator has the authorization only for the lowest role.

A user can be assigned to or revoked from a role by an administrator. A user is a member of a role if she/he is directly assigned to a role. She/he is authorized for a role, if the role is inherited from a role to which the user is assigned. A user can establish a session during which the user activates a subset of roles of which she/he is authorized.

Considering this RBAC model, the type graph for the SPF (see Fig. 2) consists of a node type ar for administrator roles, a node type r for roles, a node type u for user and a node type s for user sessions. The loops for ar and r nodes, respectively, are used to specify the hierarchy. The edge from node u to a node r is called *assignment edge* and assigns a user to a role, the loop at node u specifies the non-existence of an assignment edge. The edge between node s and a node u assigns a session to a user.

The creation and deletion of roles is not considered in this paper, so that the basic operations of the RBAC model are *add user*, *remove user*, *add assignment*, *remove assignment*, *add session*, *remove session*, *activate role* and *deactivate role*. All these operations are modeled by the graph transformation rules in Fig. 3.

The rule `add user` introduces a new user to the system. The newly created user gets a loop to indicate that the user is not yet assigned to a role. The rule `remove user` removes a user and all his/her sessions as well to ensure

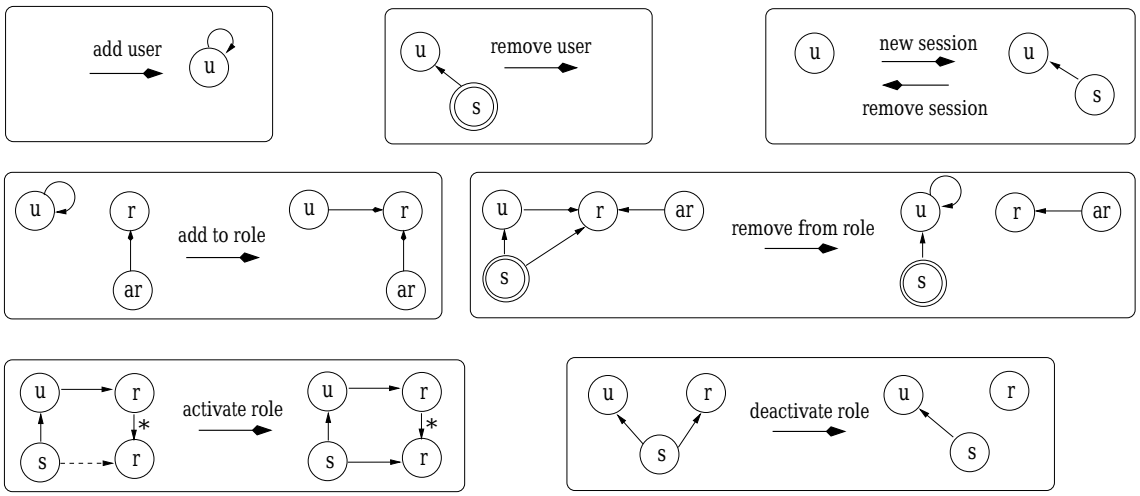

Fig. 3. *Graph rules for the centralized RBAC model.*

that there are no active sessions of the removed user. This is indicated by the double circled session node in the left-hand side of rule `remove user`. The interpretation is that *all* sessions connected to the user are deleted. The role is an abbreviation for a set of roles consisting a rule for each number of possible sessions. Negative application conditions can ensure that a rule is applicable if and only if the number of occurring session nodes is the same for which that the rule is specified.

The rules for the creation and deletion of sessions are `new session` and `remove session`. A session node s is immediately connected by an edge to the user who is using s. A session can be deleted at any time regardless of the presence of active roles of the session.

The rule `add to role` assigns a user to a role by connecting both by the assignment edge. The idea is to have at most one assignment per user, i.e. a user can be in at most one role. Membership is indicated by the non-existence of a loop at the user node u. The rule `add to role` requires the loop and deletes it when the rule sets the assignment. The rule `remove from role` removes the assignment edge and deactivates all sessions of the user. All sessions can be deactivated since all activated roles for a session are authorized by the one assignment edge. There cannot be roles activated that are not authorized by this assignment. The user node is equipped with a loop again.

A user can activate any role r for which she/he is authorized. A user is authorized for r, if there is a path starting with an assignment edge and ending in r. The corresponding graph rule is `activate role`. This edge is created by the user of the session. The star $*$ at the edge between the roles indicate a (possibly empty) path through the role hierarchy. An empty path indicates that a user can also activate a role to which she/he is directly assigned. Role r can only join a session if r is not already a member of that session, as indicated by the dashed edge for the NAC between the session node and the role node. The deactivation of a role from a session is specified by deleting

the edge between the session and the role node.

Examples of negative and positive constraints are given in Fig. 4. The negative constraints require that the loop at a user node u cannot simultaneously occur with an assignment edge (upper constraint) and that a user cannot be in two or more roles (lower constraint). This ensures that a user is assigned to at most one role. The positive constraint requires that whenever a user session is activated for a role, the user of the session is authorized for the session.

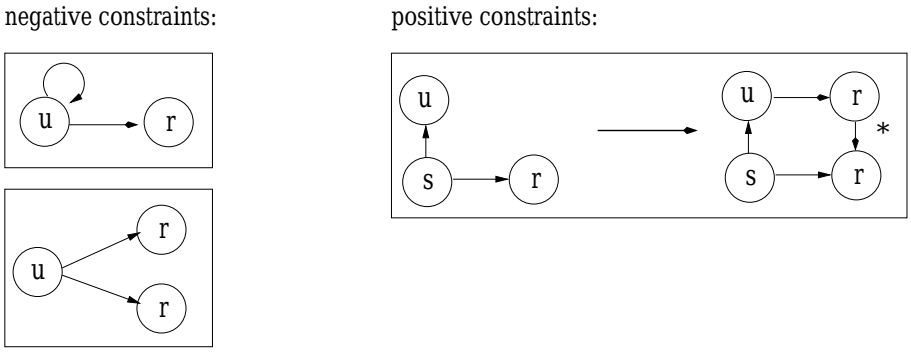

Fig. 4. *Constraints for the RBAC model.*

The graphs that can be constructed by the rules of a framework represent the system states possible within the policy model. These graphs are called *system graphs* in the sequel. A security policy framework is *coherent* if all system graphs satisfy the positive and negative constraints in the framework.

A *security policy framework morphism* $f : SP_1 \to SP_2$, or just *framework morphism*, relates security policy frameworks by a total graph morphism $f_{TG} : TG_1 \to TG_2$ between the type graphs and a mapping $f_P : P_1 \to P_2$ between the sets of rule names. The mapping f_P must preserve the behavior of rules in the sense that a rule name x can be mapped to a rule name $f_P(x)$ only if $f_P(x)$ does on the renamed types everything which x does and possibly more. The set of positive constraints in SP_2 can contain, in addition to Pos_1, new positive constraints and positive constraints of SP_1 extended w.r.t. new types. The set of negative constraints in SP_2 may contain additional negative constraints on new types, but must not impose new negative constraints on old types.

Definition 2.4 [Framework Morphism] A *framework morphism* between security policy frameworks $SP_i = (TG_i, (P_i, r_{P_i}), Pos_i, Neg_i)$ for $i = 1, 2$ is a pair $f = (f_{TG}, f_P) : SP_1 \to SP_2$, where $f_{TG} : TG_1 \to TG_2$ is a total graph morphism and $f_P : P_1 \to P_2$ is a total mapping, so that $V_{f_{TG}}(r_{P_2}(f_P(p))) = r_{P_1}(p)$ for all $p \in P_1$, $Pos_1 \subseteq V_{f_{TG}}(Pos_2)$ and $V_{f_{TG}}(Neg_2) \subseteq Neg_1$. [5]

The *category of security policy frameworks*, denoted by **SP**, has as objects all security policy frameworks and as morphisms all framework morphisms.

[5] $V_{f_{TG}}$ is the forgetful functor induced by f_{TG}.

For each framework SP, $id_{SP} = (id_{TG}, id_P)$ is the identity and composition is defined componentwise. The category **SP** is finitely cocomplete [KMPP01b].

3 Evolving policies

As in practically any development activity, the specification of a security policy is an incremental process: an initial description of the wanted (positive constraints) and of the unwanted (negative constraints) states must evolve as the requirements are better understood. The process of modifying an AC policy may deal with *local changes* (a policy evolves through the addition/removal of individual constraints and rules), with *modular changes* (two policies are combined to form a larger policy of which the original ones are subcomponents) and with *global changes* (the replacement of one policy by another one).

Local changes are needed when a policy must be modified to take into account a better understanding of, or a change in, the security requirements.

A security policy framework $SP = (TG, (P, r_P), Pos, Neg)$ can be changed by modifying its components, that is, the extension or reduction of the type graph, the addition/removal of a graph rule to/from P, the addition/removal of a positive constraint to/from Pos and the addition/removal of a negative constraint to/from Neg. A framework morphism $f : SP_1 \to SP_2$ describes the change of the framework SP_1 to the framework SP_2, but also from SP_2 to SP_1. We define an evolution as a sequence of framework morphisms in the category **SP** that can be travelled in both directions.

Definition 3.1 [evolution] An *evolution* of a framework SP to a framework SP' is a sequence $e = (SP_0 SP_1 ... SP_{n-1} SP_n)$ of frameworks such that $SP_0 = SP$, $SP_n = SP'$ and, for each $i = 0, ..., n-1$, there is a framework morphism $m_i^f : SP_i \to SP_{i+1}$ or $m_i^b : SP_{i+1} \to SP_i$.

The evolution of a security policy framework yields a new security policy framework that reflects the desired changes. The changes, however, do not ensure generally that the new security policy framework is coherent. From a semantical point of view, this problem can be solved by considering the full sub-category $\mathbf{SP}^c$ of **SP** that contains only coherent security policy frameworks. Evolution is possible only in this sub-category. From an operational point of view, we can solve the problem by using a mechanical construction originally introduced in [HW95]. The construction manipulates the rules of a framework by adding application conditions to ensure that the rules do not create graphs that do not satisfy the constraints. A methodology for generating a coherent security policy framework is presented in [KMPP00,KMPP01a].

Modular changes can occur when two companies merge while keeping the main part of their rights and their behavior so that the security information of the two companies must be merged as well. Merging takes place at the syntactical level with the security policy frameworks, and at the semantical level with the system graphs representing the state of the companies at the point

of the merge. The merge on the semantical level is what distinguishes an evolution from an integration: evolution is expressed syntactically by considering only the security policy frameworks, integration includes semantical changes, too. The integration of two AC policies on the syntactical level yields a new security policy framework via a pushout of the security policy frameworks in the category **SP**. Two security policy frameworks SP_1 and SP_2 are related by an auxiliary framework SP_0 that identifies the common parts (types and rules) in both frameworks; the actual integration is expressed by framework morphisms $f_1 : SP_0 \to SP_1$ and $f_2 : SP_0 \to SP_2$. The pushout of f_1 and f_2 in **SP** integrates the frameworks SP_1 and SP_2 in a new security policy framework SP called the *integrated framework*. Note that the integration concepts of the paper can be easily generalized to an integration of several frameworks because of the existence of (finite) colimits in **SP**.

An important integration aspect is the preservation of coherence: if the frameworks SP_1 and SP_2 are coherent, is SP? Coherence w.r.t. negative constraints is always preserved by a pushout [KMPP01b]. Coherence w.r.t. positive constraints is generally not preserved by the pushout construction. The reason for incoherence w.r.t. positive constraints, however, can be reduced to the parts of positive constraints referring to common types. Coherence of positive constraints referring to types occuring only in SP_1 or only in SP_2 is preserved.

Another important issue in integrating two policies is the overlapping of rules. If two similar rules can be applied at a given moment and, say, they are not parallel independent (i.e., the application of one may prevent the application of the other one), an (outside) decision must be made to determine the "strategy". Different strategies are possible, ranging from *Priority for one policy* to *Priority for rules*. In the first case, one of the two policies is chosen and the rules of the other one are either discarded altogether (radical solution) or equipped with a NAC to eliminate the conflict; to in the second case, each pair of rules is analyzed and one (not always from the same policy) is preferred modifying the other one (static) or the choice is made at run time (dynamic) on which rule to apply.

A global change for a policy occurrs when AC policy is deemed inadequate for a particular company that wants to replace the current one with a new one and the new one is the only one to be used henceforth. The transition from policy A to policy B can be viewed as a two-step evolution, where the first step is the embedding of A into A+B and then the inverse morphism to B, after adopting the strategy giving priority for policy A with a radical solution.

4 Concluding Remarks

We have presented a formalism to specify AC policies. States are represented by graphs and their evolution by graph transformations. A policy is formalized by four components: a type graph, positive and negative constraints (a

declarative way of describing what is wanted and what is forbidden) and a set of rules (an operational way of describing what can be constructed). The change over time of a policy can described in terms of sequences of framework morphisms, corresponding to step-by-step addition/deletion of rules and constraints. We have also discussed the effect of integrating two policies using a pushout in the category of policy frameworks and framework morphisms.

Among the remaining problems under investigation is that of comparing not only the expressive power of the different access control models (DAC, MAC, RBAC) but also their relative complexity. This could be approached by expressing the graph rules in the different models as built using expressions [GRPPS00] from a small number of basic operations easily implementable in common security systems.

References

[Bal90] R.W. Baldwin. Naming & Grouping Privileges to Simplify Security Management in Large Databases. In *Proc. of 1990 IEEE Symposium on Research in Security and Privacy*, pp. 116–132. IEEE Computer Society Press, May 1990.

[BCFP01] E. Bertino, B. Catania, E. Ferrari and P. Perlasca. A logical framework for reasoning about access control modelss. in *Proc. 6th ACM Symp. on Access Control Models and Technologies* pp. 41–52. ACM Press 2001.

[BDS00] P. Bonatti, S. D. C. di Vimercati, and P. Samarati. A Modular Approach to composing Access Control Policies. In S.Jajodia and P.Samarati, editors, *Proc. of the ACM Computers and Communication Security Conference*, pages 164–173. ACM, November 2000.

[CELP96] A. Corradini, H. Ehrig, M. Löwe, and J. Padberg. The category of typed graph grammars and their adjunction with categories of derivations. In *5th Int. Workshop on Graph Grammars and their Application to Computer Science*, number 1073 in LNCS, pages 56–74. Springer, 1996.

[GRPPS00] M. Große-Rhode, F. Parisi-Presicce, and M. Simeoni. Refinements of Graph Transformation Systems via Rule Expressions. In H. Ehrig, G. Engels, H.-J. Kreowski, and G. Rozenberg, editors, *Proc. of TAGT'98*, number 1764 in LNCS, pages 368–382. Springer, 2000.

[HRU] Harrison, M.H., Ruzzo, W.L., and Ullman, J.D., Protection in Operating Systems, Comm. of the ACM, 19, 8, pp. 461-471, Oct. 1976.

[HW95] Reiko Heckel and Annika Wagner. Ensuring consistency of conditional graph grammars - a constructive approach. In *Proc. of SEGRAGRA'95 Graph Rewriting and Computation*, number 2. Electronic Notes of TCS, 1995. http://www.elsevier.nl/locate/entcs/volume2.html.

[KMPP00] M. Koch, L.V. Mancini, and F. Parisi-Presicce. A Formal Model for Role-Based Access Control using Graph Transformation. In *Proc. of the 6th*

European Symposium on Research in Computer Security (ESORICS 2000) number 1895 in LNCS, pages 122–139. Springer, 2000

[KMPP01a] M. Koch, L. V. Mancini, and F. Parisi-Presicce. On the specification and evolution of access control policies. in *Proc. 6th ACM Symp. on Access Control Models and Technologies* pages 121–130. ACM Press 2001.

[KMPP01b] M. Koch, L.V. Mancini, and F. Parisi-Presicce. Foundations for a Graph-Based approach to the Specification of Access Control Policies. In *Proc. FoSSaCS 2001* number 2030 in LNCS, pages 287–302. Springer, 2001

[OSM00] S. Osborn, R. Sandhu, and Q. Munawer. Configuring role-based access control to enforce mandatory and descritionary access control policies. *ACM Transactions on Information and System Security*, 3(2), May 2000.

[Roz97] Grzegorz Rozenberg, editor. *Handbook of Graph Grammars and Computing by Graph Transformations. Vol. I: Foundations.* World Scientific, 1997.

[San93] R. S. Sandhu. Lattice-based access control models. *IEEE Computer*, 26(11):9–19, 1993.

[San94] R. Sandu and P. Samarati. Access Control: Principles and Practice. *IEEE Communication Magazine*, pages 40–48, 1994.

[San98] Ravi S. Sandhu. Role-Based Access Control. In *Advances in Computers*, volume 46. Academic Press, 1998.

[SFK00] R. Sandhu, D. Ferraiolo, and R. Kuhn. The NIST Model for Role-Based Access Control: Towards A Unified Standard. In *Proc. of the 5th ACM Workshop on Role-Based Access Control.* ACM, July 2000.