Foundations for a Graph–Based Approach to the Specification of Access Control Policies*

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

[1] PSI AG, Berlin (DE)
[2] Univ. di Roma La Sapienza, Rome (IT)
[3] George Mason Univ., Fairfax VA (USA)
mkoch@psi.de lv.mancini@dsi.uniroma1.it parisi@dsi.uniroma1.it ;
fparisi@ise.gmu.edu

Abstract. Graph Transformations provide a uniform and precise framework for the specification of access control policies allowing the detailed comparison of different policy models and the precise description of the evolution of a policy. Furthermore, the framework is used for an accurate analysis of the interaction between policies and of the behavior of their integration with respect to the problem of conflicting rules. The integration of policies is illustrated using the Discretionary Access Control and the Lattice Based Access Control policies.

1 Introduction

A considerable amount of work has been carried out recently on models and languages for Access Control. Access Control (AC) is concerned with determining the activities of legitimate users [SS94]. Usually AC is enforced by a reference monitor which mediates every attempted access by a *subject* (a program executing on behalf of a user) to *objects* in the system. In [KMPP00b,KMPP00a] we have proposed graph transformations as a uniform conceptual framework for the specification of access control policies. In this paper we discuss the formal properties of this framework and their applications to problems not addressed anywhere else in a formal way.

The three main AC policies commonly used in computer systems are discretionary policies [SS94], lattice-based policies (also called mandatory policies) [San93] and role-based policies [San98]. As illustrative examples, we use here the lattice-based access control (LBAC) and the access control list (ACL) that is an implementation of a discretionary policy. Role-based access control is not considered in this article, but is the main focus of [KMPP00b].

Lattice-based access control: Classic LBAC enforces unidirectional information flow in a lattice of security levels [1]. The diagram on the left-hand side of Fig. 1 shows a partial order security lattice.

* partially supported by the EC under TMR Network GETGRATS and under Esprit WG APPLIGRAPH, and by the Italian MURST.

[1] In [San93], security levels are called security labels. We use ‘security level’ here to avoid confusion with the notion of a label for a node or an edge in a graph.

F. Honsell and M. Miculan (Eds.): FOSSACS 2001, LNCS 2030, pp. 287–302, 2001.

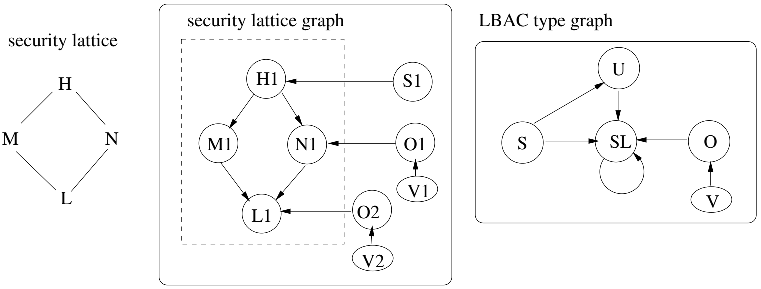

Fig. 1. A security lattice (left-hand side) and its presentation by a graph (middle). The type graph for the LBAC model (right-hand side).

The LBAC policy is expressed in terms of security levels attached to *subjects* and *objects*. A subject is a process in the system and each subject is associated to a single user, where one user may have several subjects concurrently running in the system. An object is a container of information, e.g. files or directories in an operating system. Usually the security levels on subjects and objects, once assigned, do not change. If $\lambda(x)$ denotes the security level of x (subject or object) then the specific LBAC rules for a lattice allow a subject S to read object O if $\lambda(S) \geq \lambda(O)$ and to write object O if $\lambda(S) = \lambda(O)$.

Access Control List: The ACL policy is an implementation of a discretionary AC policy. We consider an ACL policy similar, but simpler, to that one used in the UNIX operating system. Our model distinguishes only between the owner of an object and the rest of the world and for simplicity groups are not considered. The owner of the object has read, write and execution access and can change the access permissions of the object with respect to the world.

The different models for Access Control have been specified with expressive but ad hoc languages requiring ad hoc conversions to compare the relative strengths and weaknesses. Not much work [BdVS00] has been performed on the evolution of a policy, to construct in an incremental way complex policies, and on the integration of policies, to obtain the global policy of an organization by combining the security policies of different departments.

The main goal of this paper is to present some basic properties of a formal model for Access Control policies based on graphs and graph transformations and to address the problems of evolution and integration in a categorical setting. A system state is represented by a graph and graph transformation rules describe how a system state evolves. The specification ("framework") of an AC policy contains also declarative information ("invariants") on what a system graph must contain (positive) and what it cannot contain (negative). A crucial property of a framework is that it specifies a coherent policy (one without internal contradictions). As with any soft system, policies can evolve and can be combined: we have formalized here in a categorical setting the notions of evolution and of integration and their main properties. To help with the issue of conflicting rules in the integrated framework obtained by combining two distinct ones, the notion of metapolicy is proposed.

The paper is organized as follows: the next section reviews graph transformations and introduces the LBAC and the ACL policies; Sect. 3 defines the formal framework to specify AC policies and presents its main properties; Sect. 4 discusses the notion of integration of policies and Sect. 5 introduces metapolicies to resolve conflicts; the last section mentions related and future work.

2 Graph Transformations

This section introduces graph transformations [Roz97]. The LBAC model is used throughout the section to support the explanations by an example.

A *graph* $G = (G_V, G_E, s_G, t_G, l_G)$ consists of disjoint sets of nodes G_V and edges G_E, two total functions $s_G, t_G : G_E \to G_V$ mapping each edge to its source and target node, respectively, and a function $l_G : G_V \cup G_E \to L$ assigning to each node/edge a label. Labels are elements of a set $L = X \cup C$, where X is a set of *variables* and C is a set of *constants*. A binary relation $\prec \subseteq L \times L$ is defined on L as $\alpha \prec \beta$ if and only if $\alpha \in X$. A path between nodes a and b is indicated by an edge $a \overset{*}{\to} b$ and can be seen as an abbreviation for a set of paths each representing a possible sequence of edges between a and b.

A *total graph morphism* $f : G \to H$ between graphs $G = (G_V, G_E, s_G, t_G, l_G)$ and $H = (H_V, H_E, s_H, t_H, l_H)$ is a pair (f_V, f_E) of total mappings $f_V : G_V \to H_V$ and $f_E : G_E \to H_E$ that respect the graph structure, i.e. $f_V \circ s_G = s_H \circ f_E$ and $f_V \circ t_G = t_H \circ f_E$, as well as the label order, i.e. $l_G(v) \prec l_H(f_V(v))$ for each $v \in G_V$ and $l_G(e) \prec l_H(f_E(e))$ for each $e \in G_E$. A *partial graph morphism* $f : G \rightharpoonup H$ is a total graph morphism $\bar{f} : dom(f) \to H$ from a subgraph $dom(f) \subseteq G$ to H. Graphs and partial graph morphisms form a category $\mathbf{Graph^P}$. The subcategory of graphs and total graph morphisms is denoted by **Graph**. The category $\mathbf{Graph^P}$ is in general not co-complete, but has pushouts for morphisms $f_p : G \rightharpoonup H$ and $f_c : G \rightharpoonup K$ where f_p is *label preserving*, i.e. $l_G(x) = l_H(f_p(x))$ for all $x \in G$ (node or edge) [PPEM87].

A *type graph* TG represents the type information in a graph transformation system [CELP96], specifying the node and edge types which may occur in the instance graphs modeling system states. For instance, the type graph in Fig. 1 on the right shows the possible types for the LBAC graph model. There is a type U for user nodes, a type S for subjects, a type O for objects, a type V for the value of objects and a type SL for the security levels. This type graph indicates also that there cannot be an edge from a node of type U to a node of type O.

A pair $\langle G, t_G \rangle$, where G is a graph and $t_G : G \to TG$ is a total graph morphism, is called a *graph typed over* TG. If the type graph is fixed, we denote the pair simply as G. The total graph morphism t_G is called *typing morphism* and is indicated in the examples by the symbols used for nodes and edges. In the middle of Fig. 1 a graph typed over the LBAC type graph is shown. The graph consists of a node with label $H1$ and type SL, a node with label $S1$ and type S, a node with label $O1$ and type O, etc. The security lattice on the left-hand side is modeled by a security lattice graph. From now on, the typing morphism maps a node with label Tx to the type T.

A *morphism* between typed graphs $\langle G, t_G \rangle$ and $\langle H, t_H \rangle$ is given by a partial graph morphism $f : G \hookleftarrow dom(f) \to H$ that preserves types, that is, the diagram

$$\begin{tikzcd}
G \arrow[dr, "t_G"'] & dom(f) \arrow[l, hook'] \arrow[r] \arrow[d] & H \arrow[dl, "t_H"] \\
& TG &
\end{tikzcd}$$

in **Graph** commutes. The morphism is total if the underlying graph morphism is total. Graphs typed over a fixed type graph TG and morphisms between them form a category **TG** [CELP96]. The existence of pushouts is inherited from the category $\mathbf{Graph^P}$.

A graph typed over a type graph TG can be *re-typed* over TG' if there is a total morphism $f : TG \to TG'$. The re-typing of a graph $\langle G, t_G \rangle$ typed over TG to a graph typed over TG' is a renaming of types in G. Re-typing from TG' to TG is a renaming of types and a forgetting of nodes and edges. Formally, the re-typing w.r.t. a morphism $f : TG \to TG'$ is specified by functors $F_f : \mathbf{TG} \to \mathbf{TG'}$ and $V_f : \mathbf{TG'} \to \mathbf{TG}$, called *forward typing* and *backward typing functor* [CELP96,GRPPS98].
A *graph rule* $p : r$, or just *rule*, is given by a rule name p, from a set $RNames$, and a label preserving *morphism* $r : L \rightharpoonup R$. The graph L, *left-hand side*, describes the elements a graph must contain for p to be applicable. The partial morphism is undefined on nodes/edges that are intended to be deleted, defined on nodes/edges that are intended to be preserved. Nodes and edges of R, *right-hand side*, without a pre-image are newly created.

Example 1 (Graph rules for the LBAC graph model). Figure 2 shows the rules for the LBAC policy. The labels for the nodes ($Ux, Sx, SLx, SLy, ...$) of the rules are variables taken from the set of variables in L. The rule `new object` creates a new object Ox connected to a node Vx (the initial value of the object). The object

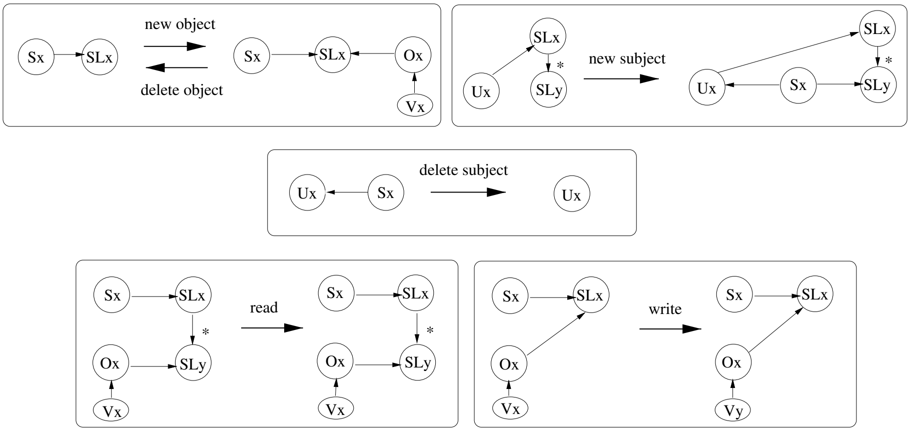

Fig. 2. Graph rules for the LBAC policy.

Ox gets the security level SLx. The variable SLx is generic: it is substituted by the actual security level of the subject when the rule is applied. The rule for `delete object` for the deletion of objects is represented by reversing the partial morphism of the rule `new object`. The rule `new subject` creates a subject Sx on behalf of a user Ux. The new subject is attached to a security level SLy that is lower in the hierarchy graph than the security level SLx of the user Ux. This requirement is specified by the path from SLx to SLy, since edges in the security lattice graph point from higher to lower levels.

For the application of rules we use the Single Pushout (SPO) approach to graph transformations [EHK+97]. Formally, the application of a graph rule $p : L \overset{r}{\rightharpoonup} R$ to a graph G is given by a total graph morphism $m : L \to G$, called *match* for p in G. The direct derivation $G \overset{p,m}{\Rightarrow} H$ from G to the derived graph H is given by the pushout of r and m in **TG** (see the diagram below). Note that the pushout exists, since the rule morphism is label preserving [PPEM87].

$$\begin{tikzcd}
L \arrow[r, harpoon, "r"] \arrow[d, "m"'] & R \arrow[d, "m^*"] \\
G \arrow[r, harpoon, "r^*"'] & H
\end{tikzcd}$$

Example 2 (Application of a graph rule). In Fig. 3, the left-hand side L of the rule `new subject` occurs several times in G. In one possible match the node Ux in L is associated to the node U_2 in G and the nodes SLx and SLy to the specific security level H.

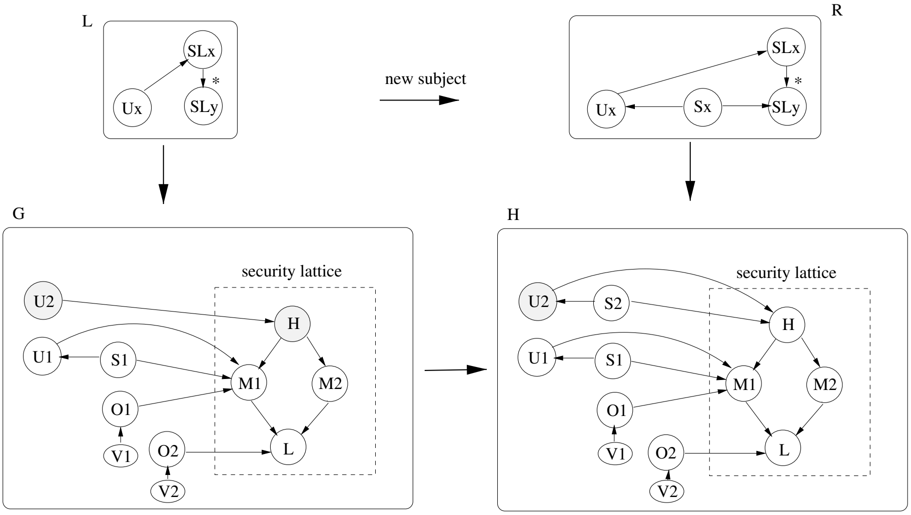

Fig. 3. Application of rule `new subject`.

For the specification of the ACL by graph transformations, *negative application conditions* for rules are needed. A negative application condition (NAC) for a rule $p : L \overset{r}{\rightharpoonup} R$ consists of a set $A(p)$ of pairs (L, X), where the graph L is a subgraph of X. The part $X \setminus L$ represents a structure that must not occur in a graph G for the rule to be applicable. In the figures, we depict (L, X) by the graph X, where the subgraph L is drawn with solid and $X \setminus L$ with dashed lines. A rule $p : L \overset{r}{\rightharpoonup} R$ with a NAC (L, X) is applicable to G if L occurs in G and it is not possible to extend L to X. Examples of rules with a NAC are the ACL rules `connect` and `give read` in Fig. 5.

Example 3 (Graph rules for the ACL). The type graph TG_{ACL} in Fig. 4 provides the node types U, O and P. Just as in the LBAC model, a node of type U represents a user and a node of type O an object. An edge between a user node U and an object node O specifies that U is the owner of the object O. A node of type P represents a process. An edge of type R, W or X represents the read, write or execute permission of an object to the world. The owner of the object has always all the permissions for his/her objects and does not need the loops. Some of the ACL graph rules are shown in Fig. 5. The rule `new process` starts a new process on behalf of a user. To kill a process, the rule `remove process` deletes the process node and its connection to the user. The rule `create object` adds a new node Ox to the system, connecting it to the process node Px that

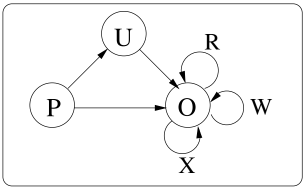

Fig. 4. The type graph for the ACL.

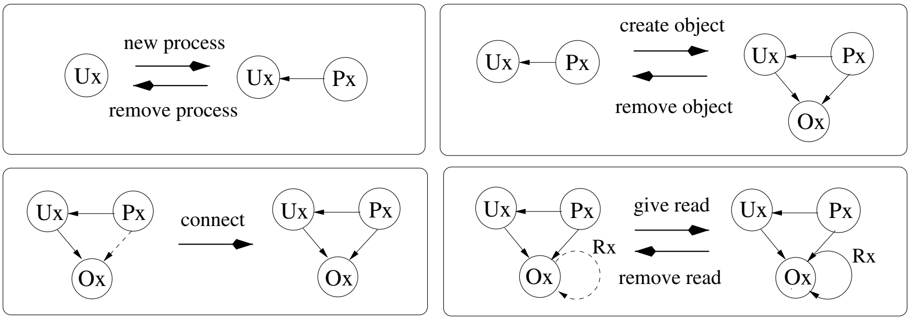

Fig. 5. Graph rules for the ACL model.

has created the object and to the user node Ux to which the process belongs. The rule `connect` connects a process of a user to an object of the user. The rule has a NAC (indicated by the dashed edge between Px and Ox on the left-hand side of the rule) that forbids the application of the rule to processes and objects of the user already connected. The rule `give read` gives the read permission provided that it has not been granted already. Other rules such as `give write` and `give execution` are not shown.

3 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. 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 informations from "constructive" rules. Positive and negative constraints can be considered as formal documentation of the initial requirements and the development process of rules. Both positive and negative constraints are formally specified by morphisms. Only their semantics distinguishes them.

Definition 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 graph morphism $p : X \to G$ there exists (does not exist) a total graph morphism $q : Y \to G$ such that $X \xrightarrow{c} Y \xrightarrow{q} G = X \xrightarrow{p} G$.*

Example 4 (Constraints for LBAC and ACL). Figure 6 shows two positive constraints and two negative constraints for the LBAC model. The morphisms for the negative constraints are the identity on the graphs shown. The positive and the negative constraint on the left-hand side require that objects always have a security level (the positive constraint) and that there does not exist more than one security level for an object (negative constraint). The right-hand side of the figure specifies the same existence and uniqueness requirements for subjects.

The constraints for the ACL framework in Fig. 7 require that a process belong to a unique user (the positive and the first negative constraint), that an object

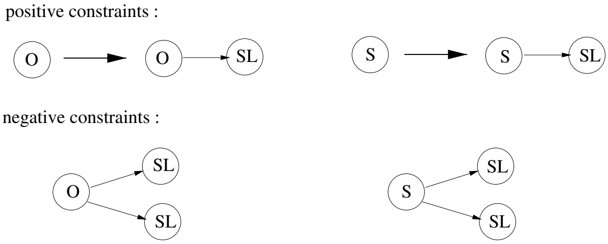

Fig. 6. Positive and negative constraints for LBAC.

does not belong to more than one user (the second negative one) and there is at most one permission loop with the same permission attached to the same object (the third negative one). Note that the third negative constraint represents three negative constraints, one for R, one for W and one for X.

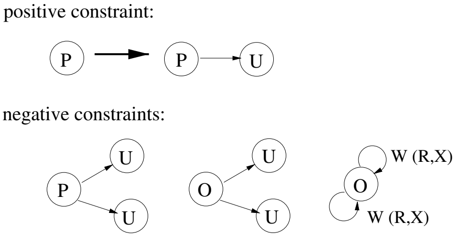

Fig. 7. Positive and negative constraints for the ACL.

Definition 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 $r_P : P \to |\mathbf{Rule}(TG)|$ mapping each rule name to a rule $L \xrightarrow{r} R$ of TG–typed graphs, Pos is a set of positive and Neg is a set of negative constraints.*

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 *positive* (resp. *negative*) *coherent* if all system graphs satisfy the constraints in Pos (resp. Neg). It is *coherent* if it is both positive and negative coherent.

The security policy framework for the LBAC policy consists of the type graph in Fig. 1 and the negative and positive constraints in Fig. 6. The rule names $p_1, ..., p_6$ are mapped to the rules in Example 1.

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 as 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 3 (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 $V_{f_{TG}}(Neg_2) \subseteq Neg_1$.*

We provide now the categorical formalization by defining the category of security policy frameworks and framework morphisms.

Definition 4 (Category of Security Policy Frameworks). *The* category of security policy frameworks*, denoted by* **SP***, has as objects all security policy frameworks and as morphisms all framework morphisms. For each framework SP, $id_{SP} = (id_{TG}, id_P)$ is the identity and composition is defined componentwise.*

Proposition 1 (Initial Security Policy Framework). *The initial object in* **SP** *is given by the security policy framework $SP_I = (\emptyset, (\emptyset, r), \emptyset, \emptyset)$.*

Security policy frameworks can be glued together using the standard categorical constructions.

Proposition 2 (Pushouts). *The category* **SP** *has all pushouts.*

By combining the previous two, we obtain the main result of this section.

Theorem 1 (Colimits). *The category* **SP** *is finitely cocomplete.*

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 5 (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 $\mathbf{SP}$ 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 [KMPP00b,KMPP00a].

4 Integration of Security Policy Frameworks by Pushouts

Integration is concerned with the merging of AC policies. A merge is necessary on the syntactical level, i.e. a merge of the security policy frameworks, and on the semantical level, i.e. the merge of the system graphs representing the state at merge time. 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 is a pushout of the security policy frameworks in the category **SP**[2]. 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*.

Throughout this section, the integration of the LBAC framework with the ACL framework (both introduced in Section 3) is used as an example.

Example 5 (Pushout integration of the ACL and LBAC frameworks). The type graph in the middle of Fig. 8 shows the types common to ACL and $LBAC$. The U and the O types are in common and the type P (processes in ACL) and the type S (subjects in LBAC) coincide. The edge between the U and the P (resp. S) node is a common part as well. The pushout of the two type graphs is the type graph at the bottom of Fig. 8. The pushout identifies the P and S node to a common type P. All rules are kept in the integrated security policy framework, where their graphs are now typed over the integrated type graph. The integrated policy framework contains the two positive constraints of the LBAC model (now typed over the integrated type graph) and the positive ACL constraint. The integrated framework has no negative constraints, since there are (after re-typing) no common negative constraints in ACL and $LBAC$.

An important integration aspect is the preservation of coherence: if the frameworks SP_1 and SP_2 are coherent, is SP? Generally, this is not the case. The

[2] 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**.

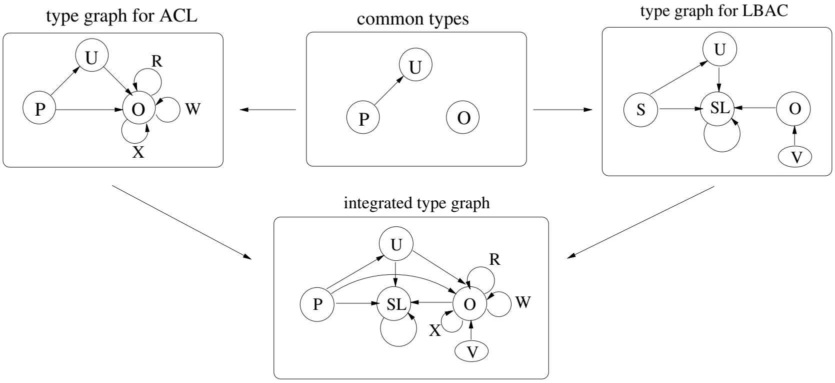

Fig. 8. Integrated type graph for the combined LBAC and ACL security model.

integrated framework for the ACL and the LBAC frameworks of the previous example contains a positive constraint that requires a security level for each node of type P (stemming from the identification of the LBAC type S and the ACL type P). This requirement can be destroyed by the ACL rule `new process`. Negative constraints, however, are preserved by the pushout.

Proposition 3 (Preservation of negative coherence). *Given framework morphisms $f_1 : SP_0 \rightarrow SP_1$ and $f_2 : SP_0 \rightarrow SP_2$ so that SP_1 and SP_2 are coherent, the pushout object $SP = (TG^{int}, (P^{int}, r_{P^{int}}), Pos^{int}, Neg^{int})$ of f_1 and f_2 in* **SP** *is coherent w.r.t. the set of negative constraints Neg^{int}.*

Coherence w.r.t. positive constraints is generally not preserved by the pushout construction as the counterexample above shows. 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.

Proposition 4 (Preservation of positive coherence). *Given framework morphisms $f_1 : SP_0 \rightarrow SP_1$ and $f_2 : SP_0 \rightarrow SP_2$ with SP_1 and SP_2 coherent, the pushout $SP = (TG^{int}, (P^{int}, r_{P^{int}}), Pos^{int}, Neg^{int})$ of f_1 and f_2 in* **SP** *is incoherent if and only if SP is incoherent w.r.t. positive constraints of Pos^{int} containing types in TG_0.*

After merging the AC policies on the syntactical level to an integrated SP, the AC policies are merged on the semantical level by a pushout of the system graphs G_1 of SP_1 and G_2 of SP_2 representing the system states at merge time. The merge of the system graphs must yield a graph typed over the integrated type graph TG^{int} of SP.

Example 6 (Integration of LBAC and ACL System Graphs). Figure 9 shows a system graph for the ACL framework and the LBAC framework. The auxiliary graph G_0 contains the user $U1$ (may be working for both companies) and the objects $O1$ and $O2$ (may be files already shared before the merge). The integrated system graph at the bottom of Fig. 9 contains features of both frameworks.

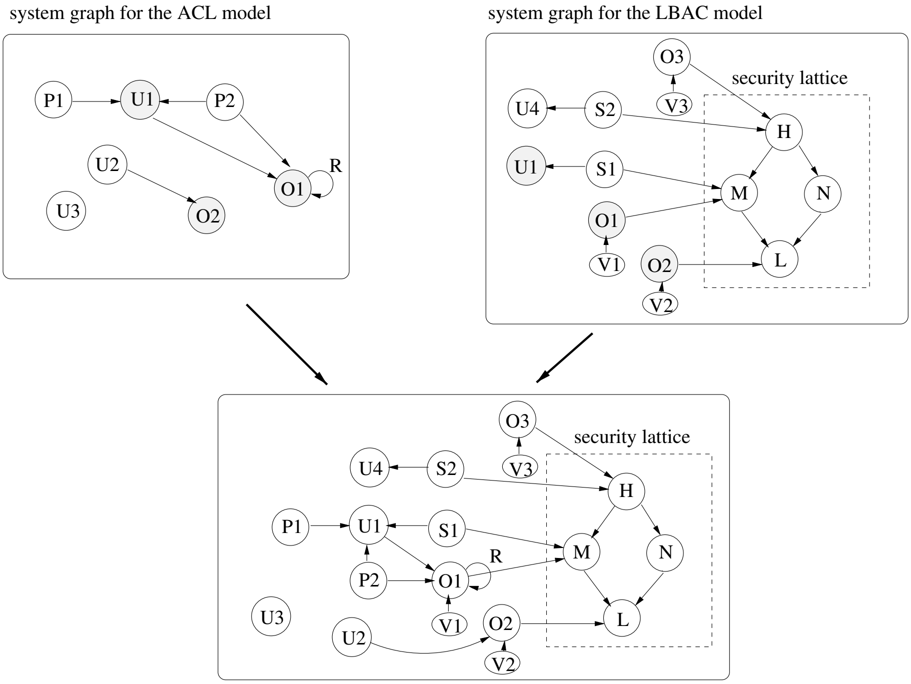

Fig. 9. Integrated system graph for the combined LBAC and ACL framework.

The integration of the system graphs does not ensure that the integrated system graph satisfies the constraints of the integrated framework SP. There are two possible solutions: first, the integrated system graph is adapted to satisfy the constraints. In the example, the integrated system graph in Fig. 9 could be modified by connecting objects and processes without security level with the security lattice graph. Second, the set of constraints can be changed by, for instance, removing the unsatisfied constraints. This is meaningful, if one policy is preferred and the constraint is from the other policy. In the example, the LBAC constraint for security levels could be removed if the ACL policy is preferred.

5 Solving Rule Conflicts by Meta Policies

In this section, we assume that two security policies are integrated by a pushout in **SP** for the frameworks and a pushout in $\mathbf{TG}^{int}$ for the system graphs. Even if the integrated system graph is modified to be coherent w.r.t. the constraints of the integrated framework, there may be rules in the integrated framework, coming from SP_1 and SP_2, respectively, that are applicable to the same part of the integrated system graph. If the rules specify two conflicting actions specific to the policy, we call such a situation a *rule conflict* and denote by $CR(SP)$ the set of conflicting rule pairs. For instance, the LBAC rule `new object` and the ACL rule `create object` are in conflict, since both are applicable to the user

$U1$ in the integrated system graph in Fig. 9. The rule `new object` would create an object with a security level, the rule `create object` an object without one. Which rule shall be applied in this case? Several strategies, called *meta policies*, are possible to resolve conflicts between rules. A meta policy is specified by mappings, on the rule names, that define a set of rules intended for addition or deletion from the integrated framework SP (the "gluing" of SP_1 and SP_2).

Definition 6 (meta policy). *A* meta policy $MP = (strategy_i)_{i \in I}$ *for* SP *is an* I*-indexed set of partial mappings* $strategy_i : D_i \to |\mathbf{Rules}|$ *from a set* $D_i \subseteq P_1 \cup P_2$ *to the set of rules* $|\mathbf{Rules}|$.

A mapping $strategy : D \to |\mathbf{Rules}|$ of a meta policy changes the framework SP with respect to the rules by deleting all rules from SP that occur in D and by adding the rules in the image $strategy(D)$. After a framework is modified for all mappings in a meta policy MP, it is called *closed under* MP.

Definition 7 (closure under meta policy). *Let* SP *be the integrated framework for* SP_1 *and* SP_2 *and* $MP = (strategy_i : D_i \to |\mathbf{Rules}|)_{i \in I}$ *a meta policy for* SP*. The* framework $SP^{MP} = (TG^{int}, (P^{MP}, r_{P^{MP}}), Pos^{int}, Neg^{int})$ closed under MP *consists of the set of rules names*

$$P^{MP} = P^{int} \setminus \bigcup_{i \in I} D_i \cup \bigcup_{i \in I} dom(strategy_i).$$

The mapping $r_{P^{MP}} : P^{MP} \to |\mathbf{Rules}|$ *is defined for* $p \in P^{int}$ *as* $r_{P^{MP}} = r_{P^{int}}(p)$ *and for* $p \notin P^{int}$ *as* $r_{P^{MP}}(p) = strategy_i(p)$*, where* $p \in dom(strategy_i)$*.*

For the following examples we introduce the *weakrule* $p_2(p_1)$ of a rule p_2 w.r.t. a rule p_1. The weakrule is derived from rule p_2 by adding a negative application condition $WAP(p_2,p_1)$, that allows the application of the rule p_2 only if the rule p_1 is not applicable. The application condition $WAP(p_2,p_1)$ is constructed by extending the left-hand side L_2 of p_2 to $L_1 \cup L_2$.

Definition 8 (weak rule). *Given two rules* $p_1 : L_1 \xrightarrow{r_1} R_1$ *and* $p_2 : L_2 \xrightarrow{r_2} R_2$*, the* weak condition *for* p_2 *w.r.t.* p_1*, denoted by* $WAP(p_2,p_1)$*, is the pair* $(L_2, L_1 \cup L_2)$*. The* weakrule $p_2(p_1)$ *of* p_2 *w.r.t.* p_1 *is the rule* p_2 *extended by the negative application condition* $WAP(p_2,p_1)$*.*

Example 7 (weak rule). Figure 10 shows the example of the ACL rule `create object` and the LBAC rule `new object` (note that the rule LBAC rule is typed over the integrated type graph, with a Px node instead of a Sx node). The weak rule for `create object` w.r.t. `new object` has a NAC that forbids the application to a process with a security level. Therefore, the weak rule for `create object` is only applicable to processes coming from the ACL model and without a counterpart in the LBAC model. The weak rule for `new object` w.r.t. `create object` has a NAC that forbids the application if a user is connected to the process. Since each user is connected to a process, the rule is not applicable.

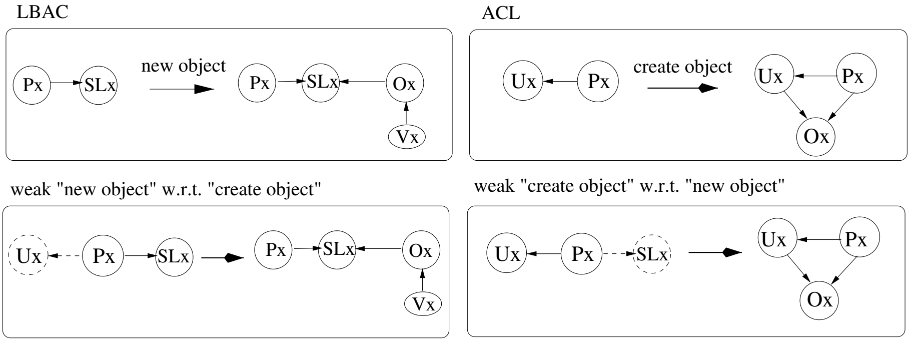

Fig. 10. The weak rule for `new object` and `create object`.

The weak rule ensures that the less important (minor) rule of SP_2 is applied only if the more important (major) rule of SP_1 is not applicable.

Lemma 1. *The weak rule $p_2(p_1)$ is applicable at a match $m : L_2 \to G$ if and only if the morphism m cannot be extended to a match for p_1.*

We introduce three possible meta policies, called *radical*, *weakRadical* and *static*. The meta policies *radical* and *weakRadical* choose a major SP_1 and a minor policy SP_2. The meta policy *radical* solves the problem of conflicting rules globally by selecting the rules of SP_1 and deleting the rules of SP_2. It consists of the mapping $killMinor : P_2 \to |\mathbf{Rules}|$ defined as identity for the rules of SP_2 not in a conflict with a rule in SP_1.

$$killMinor(p_2) = \begin{cases} \text{undefined} & \text{, if there is } (p_1, p_2) \in CR(SP) \\ p_2 & \text{, otherwise} \end{cases}$$

The effect of the meta policy *radical* is that SP_1 will "survive" during the subsequent evolution of the system, whereas the framework SP_2 "dies". The meta policy *weakRadical* keeps the conflicting rule of SP_2, extended by all NACs of the weak rule w.r.t. conflicting rules, and consists of the mapping $weakMinor : P_2 \to |\mathbf{Rules}|$ defined by

$$weakMinor(p_2) = p_2 \text{ with weak condition } WAP(p_1, p_2)\ \forall (p_1, p_2) \in CR(SP)$$

To non-conflict points, the weakened rule p_2 is still applicable so that the part of the system graph where the rule p_2 is applicable is only reduced.

Whereas the meta policies *radical* and *weakRadical* favor the major framwork SP_1, the choice in the meta policy *static* depends on the conflicting rule pair. The meta policy *static* consists of the two mappings $weakMinor : D_2 \to |\mathbf{Rules}|$ as above with a domain $D_2 \subseteq P_2$ and $weakMajor : D_1 \to |\mathbf{Rules}|$ with $D_1 \subseteq P_1$ similar to the mapping *weakMinor*, defined by

$$weakMajor(p_1) = p_1 \text{ with weak condition } WAP(p_2, p_1)\ \forall (p_1, p_2) \in CR(SP)$$

The meta policy *weakRadical* is a special case of the meta policy *static*, where $D_2 = P_2$ and $D_1 = \emptyset$.

Proposition 5 (conflictfreenes). *The closure of the framework SP under the metapolicies radical or weakRadical is conflictfree. If $D_1 \cap D_2 = \emptyset$ and $D_1 \cup D_2 = \{p, p' | (p, p') \in CR(SP)\}$, then the closure under static is conflictfree.*

6 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. The problem of dealing with inconsistencies caused by conflicts between a rule of one policy and a constraint of the other policy has been addressed in part elsewhere [KMPP00b, KMPP00a], where it is also shown the adequacy of this framework to represent different Access Control policies.

Besides the new results in Sections 3 and 4, we have introduced the notion of metapolicy and shown that three natural ones transform a policy resulting from an integration into a conflict-free policy. The choice of the appropriate meta policy may depend on the specific application domain of the particular AC model. Among the problems still under investigation are the transition from a system using one policy to a system using another policy.

References

[BdVS00] P. Bonatti, S. De Capitani di Vimercati, and P. Samarati. A modular approach to composing access control policies. In *Proc. of the 7th ACM Conference on Computer and Communication Security.* 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.

[EHK$^+$97] H. Ehrig, R. Heckel, M. Korff, M. Löwe, L. Ribeiro, A. Wagner, and A. Corradini. *Handbook of Graph Grammars and Computing by Graph Transformations. Vol. I: Foundations*, chapter Algebraic Approaches to Graph Transformation Part II: Single Pushout Approach and Comparison with Double Pushout Approach. In Rozenberg [Roz97], 1997.

[GRPPS98] M. Große-Rhode, F. Parisi-Presicce, and M. Simeoni. Spatial and Temporal Refinement of Typed Graph Transformation Systems. In *Proc. of MFCS'98*, number 1450 in LNCS, pages 553–561. Springer, 1998.

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

[KMPP00a] M. Koch, L. V. Mancini, and F. Parisi-Presicce. On the specification and evolution of access control policies. Technical Report SI-2000/05, Dip.Scienze dell'Informazione, Uni. Roma La Sapienza, May 2000.

[KMPP00b] M. Koch, L.V. Mancini, and F. Parisi-Presicce. A Formal Model for Role-Based Access Control using Graph Transformation. In F.Cuppens, Y.Deswarte, D.Gollmann, and M.Waidner, editors, *Proc. of the 6th European Symposium on Research in Computer Security (ESORICS 2000)*, number 1895 in LNCS, pages 122–139. Springer, 2000.

[PPEM87] F. Parisi-Presicce, H. Ehrig, and U. Montanari. Graph Rewriting with unification and composition. In H. Ehrig, M. Nagl, G. Rozenberg, and A. Rosenfeld, editors, *Int. Workshop on Graph Grammars and their Application to Computer Science*, number 291 in LNCS, pages 496–524. Springer, 1987.

[Roz97] G. 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.

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

[SS94] R.S. Sandhu and P. Samarati. Access Control: Principles and Practice. *IEEE Communication Magazine*, pages 40–48, 1994.