Conflict Detection and Resolution in Access Control Policy Specifications*

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

[1] Freie Universität Berlin, Berlin (DE)
mkoch@inf.fu-berlin.de
[2] Univ. di Roma La Sapienza, Rome (IT)
lv.mancini@dsi.uniroma1.it
[3] George Mason University, Fairfax VA (USA)
parisi@dsi.uniroma1.it, fparisi@ise.gmu.edu

Abstract. Graph-based specification formalisms for Access Control (AC) policies combine the advantages of an intuitive visual framework with a rigorous semantical foundation. A security policy framework specifies a set of (constructive) rules to build the system states and sets of positive and negative (declarative) constraints to specify wanted and unwanted substates. Models for AC (e.g. role-based, lattice-based or an access control list) have been specified in this framework elsewhere. Here we address the problem of inconsistent policies within this framework. Using formal properties of graph transformations, we can systematically detect inconsistencies between constraints, between rules and between a rule and a constraint and lay the foundation for their resolutions.

1 Introduction

Access Control (AC) deals with decisions involving the legitimacy of requests to access files and resources on the part of users and processes. One of the main advantages of separating the logical structure from the implementation of a system is the possibility to reason about its properties. In [KMPP00,KMPP01a] we have proposed a formalism based on graphs and graph transformations for the specification of AC policies. This conceptual framework, that we have used in [KMPP00,KMPP01a] to specify well-known security models such as role-based policies [San98], lattice-based access control (LBAC) policies (examples of mandatory policies) [San93] and access control lists (ACL) (examples of discretionary policies) [SS94], allows for the uniform comparison of these different models, often specified in ad hoc languages and requiring ad hoc conversions to compare their relative strength and weaknesses.

Our graph-based specification formalism for AC policies combines the advantages of an intuitive visual framework with a rigor and precision of a semantics

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

founded on category theory. In addition, tools developed for generic graph transformation engines can be adapted to or can form the basis for applications that can assist in the development of a specific policy.

We use in this paper examples from the LBAC and the ACL models to illustrate the different concepts, with no pretense of giving complete or unique solutions by these examples.

The main goal of this paper is to present some basic properties of a formal model for AC policies based on graphs and graph transformations and to address the problem of detecting and resolving conflicts 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, that is one without internal contradictions. Formal results are presented to help in recognizing when the positive and the negative constraints of a framework cannot be simultaneously satisfied, when two rules, possibly coming from previously distinct subframeworks, do (partly) the same things but under different conditions, and when the application of a rule produces a system graph that violates one of the constraints (after one or the other has been added to a framework during the evolution of a policy). The solutions proposed on a formal level can be made part of a methodology and incorporated into an Access Control Policy Assistant.

The paper is organized as follows: the next section reviews the basic notations of graph transformations and recalls the formal framework to specify AC policies [KMPP01a]; Sect.3 discusses the notion of a conflict of constraints, Sect.4 introduces conflicts between rules and mentions strategies to resolve conflicts; Sect.5 discusses how to modify a rule so that its application does not contradict one of the constraints; the last section mentions related and future work.

2 Graph-Based Security Policy Frameworks

We assume that the reader is familiar with the basic notation for graph transformations as in [Roz97] and in [KMPP01a]. Parts of a LBAC model are used throughout the section to illustrate the explanations by examples.

Graphs $G = (G_V, G_E, s_G, t_G, l_G)$ carry labels taken from a set X of *variables* and a set C of *constants*. A path of unspecified length between nodes a and b is indicated by an edge $a \xrightarrow{*} b$ as an abbreviation for a set containing all possible paths from a to b through the graph.

A *total morphism* $f : G \to H$ is a pair $(f_V : G_V \to H_V,\ f_E : G_E \to H_E)$ of total mappings that respect the graph structure and may replace a variable with other variables or constants. 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 can be typed by defining a total morphism $t_G : G \to TG$ to a fixed *type graph* TG that represents the type information in a graph transformation system [CELP96] and specifies the node and edge types which may occur in the

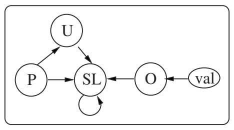

Fig. 1. *The type graph for the LBAC model.*

instance graphs modeling system states. For example, the type graph in Figure 1 shows the possible types for the LBAC graph model. The node U is the type of nodes representing users, the node O the objects, the node val the actual information of objects and the node P the processes that run on behalf of users. The node SL with its loop represents a whole security lattice, and there is an edge from security level SL_1 to SL_2 if $SL_1 > SL_2$. The attachment of security levels to objects, users and processes is modeled by an edge to a security level of the security lattice. The typing morphism t_G maps a node with label Tx to the type T, and morphisms must preserve the typing.

A *rule* $p : r$ consists of a name p, and a label preserving injective morphism $r : L \rightharpoonup R$. The *left-hand side* L describes the elements a graph must contain for p to be applicable. The partial morphism r 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. Note that the actual deletions/additions are performed on the graphs to which the rule is applied. The application of a rule $p : r$ to a graph G requires a total graph morphism $m : L \to G$, called *match*, and the direct derivation $G \overset{p,m}{\Rightarrow} H$ is given by the pushout of r and m in the category of graphs typed over TG [EHK+97].

Example 1 (LBAC graph rules). Figure 2 shows the rules for the LBAC policy. The labels for the nodes $(Ux, Px, SLx, SLy, ...)$ of the rules are variables.

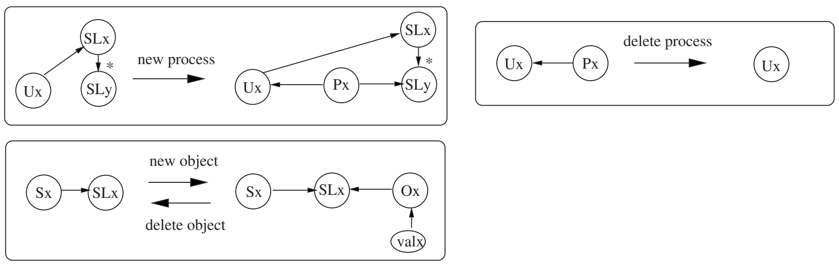

Fig. 2. *Graph rules for the LBAC policy.*

The rule `new object` creates a new object Ox connected to a node $valx$ (the initial value of the object) and assigned to the security level SLx. The label SLx is generic and is substituted by the actual security level of the process when the rule is applied. The rule `delete object` for the deletion of objects is represented by reversing the partial morphism of the rule `new object`. The rule `new process` creates a process Px on behalf of a user Ux. The new process is attached to a security level SLy that is no higher in the security lattice graph than the security level SLx of the user Ux. This requirement is specified by the path from SLx to SLy. Processes are removed by the rule `delete process`.

For the specification of AC policies by graph transformations, *negative application conditions* for rules are needed. A negative application condition (NAC) for a rule $p : L \xrightarrow{r} R$ consists of a set $A(p)$ of pairs (L, N), where the graph L is a subgraph of N. The part $N \setminus L$ represents a structure that must not occur in a graph G for the rule to be applicable. In the figures, we represent (L, N) with N, where the subgraph L is drawn with solid and $N \setminus L$ with dashed lines. A rule $p : L \xrightarrow{r} R$ with a NAC $A(p)$ is applicable to G if L occurs in G via m and it is not possible to extend m to N for each (L, N) in $A(p)$.

Example 2 (NAC). Figure 3 shows the rules for modifying the security lattice. New security levels can be inserted above an existing security level (rule `new level 1`), below (`new level 2`) or between existing levels (`new level 3`). (Notice that the lattice structure is not preserved by these rules.) The rule `delete level` removes a security level. Since users, processes and objects need a security level, security levels cannot be removed if a user, process or object possesses this level. Thus, the NAC of the rule `delete level`, whose left-hand side contains the node SLx, has three pairs (L, N): the first one prevents the deletion of security levels that are assigned to a process, the second one concerns users and the last one objects. Only if the NAC is satisfied, a security level can be removed.

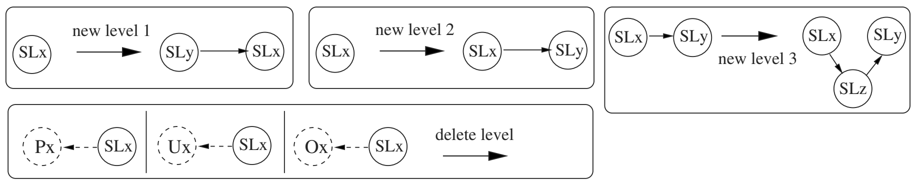

Fig. 3. *LBAC rules for modifying the security lattice.*

Negative application conditions are a form of constraint on the applicability of a rule. Constraints can be defined independently of rules.

Definition 1 (Constraints). *A* constraint *(positive or negative) is given by a total morphism $c : X \to Y$. A total morphism $p : X \to G$* satisfies *a positive (negative) constraint c if there exists (does not exist) a total morphism $q : Y \to G$ such that $X \xrightarrow{c} Y \xrightarrow{q} G = X \xrightarrow{p} G$.*
A graph G satisfies *a constraint c if each total morphism $p : X \to G$ satisfies c. A graph G* vacuously *satisfies c if there is no total morphism $p : X \to G$; G* properly *satisfies c otherwise.*

Example 3 (Constraints for LBAC). Figure 4 shows a positive and a negative constraint for the LBAC model. The morphism for the negative constraint is the identity on the graph shown (to simplify the presentation, we depict only the graph). The constraints require that objects always have one (the positive constraint) and only one (negative constraint) security level.

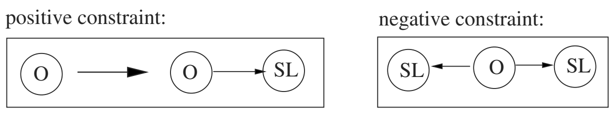

Fig. 4. *Positive and negative constraints for LBAC.*

We now review the specification of AC policies based on graph transformations [KMPP00]. The framework is called *security policy framework* and consists of a type graph that provides the type information of the AC policy, a set of rules (specifying the policy rules) that generate the graphs representing the states accepted by the AC policy, a set of *negative constraints* to specify graphs that shall not be contained in any system graph and a set of *positive constraints* to specify graphs that must be explicitly constructed as parts of a system graph.

Definition 2 (Security Policy Framework). *A* security policy framework*, or just* framework*, is a tuple $SP = (TG, (P, rules_P), Pos, Neg)$, where TG is a type graph, P a set of rule names, $rules_P : P \to \mathbf{Rule}(TG)$ a total mapping from names to TG–typed rules, Pos is a set of positive constraints, and Neg is a set of negative constraints.*

The graphs constructed by the rules of a framework represent the system states possible within the policy model. These graphs are called *system graphs*.

Definition 3 (Coherence). *A security policy framework is* coherent *if all system graphs satisfy the constraints in Pos and Neg.*

Integration is concerned with the merging of AC policies and consists of two levels, a syntactical level, i.e. a merge of the security policy frameworks, and a semantical level, i.e. the merge of the system graphs representing the state at merge time. The integration of two AC policies on the syntactical level is a pushout of the frameworks in the category **SP**. It has been shown in [KMPP01b] that the category **SP** of frameworks and framework morphisms is closed under finite colimit constructions. An important aspect of integration is the preservation of coherence: if two frameworks are coherent, is their gluing also coherent? Generally, this is not the case. Conflicts also arise when modifying a framework by adding/removing a rule or by adding/removing a positive/negative constraint. In the next three sections, the problems of conflicting constraints, conflicting rules and conflicts between a rule and a constraint are addressed.

3 Constraint-Constraint Conflict

One way to determine whether a framework is contradictory is to analyze constraints in pairs.

Definition 4 (Conflict of Constraints). *Given constraints $c_i : X_i \to Y_i$ for $i = 1, 2$, c_1 is* in conflict *with c_2 iff there exist morphisms $f_X : X_1 \to X_2$ and $f_Y : Y_1 \to Y_2$ such that $f_Y \circ c_1 = c_2 \circ f_X$ and f_X does not satisfy c_1. The conflict is* strict *if the diagram is a pushout.*

$$\begin{tikzcd} X_1 \arrow[r, "c_1"] \arrow[d, "f_X"'] & Y_1 \arrow[d, "f_Y"] \\ X_2 \arrow[r, "c_2"'] & Y_2 \end{tikzcd}$$

When two constraints contain redundant restrictions, the conflict is *harmless.*

Proposition 1 (Harmless Conflicts). *Let c_1 be in conflict with c_2 and G satisfy c_1. Then G satisfies c_2 whenever either $c_1, c_2 \in Neg$ or $c_1, c_2 \in Pos$ and c_1 is in strict conflict with c_2.*

When the two constraints in conflict are one positive and one negative, then any graph satisfying one cannot properly satisfy the other one.

Proposition 2 (Critical Conflicts). *Let c_1 be in conflict with c_2 and G properly satisfy c_2. If either $c_1 \in Pos$, $c_2 \in Neg$ and the conflict is strict, or $c_1 \in Neg$, then G does not properly satisfy c_1.*

Critical conflicts between constraints can be resolved by removing or weakening one of the constraints by adding a condition.

Definition 5 (Conditional Constraint). *A positive (negative)* conditional constraint (x, c) *consists of a negative constraint $x : X \to N$, called* constraint condition*, and a positive (negative) constraint $c : X \to Y$. A total morphism $p : X \to G$ satisfies (x, c) iff whenever p satisfies x, p satisfies c. A graph G satisfies (x, c) iff each total morphism $p : X \to G$ satisfies (x, c).*

A conditional constraint solves the conflict of c_1 with c_2 (via f_X and f_Y) by introducing a constraint condition for c_1 that requires the satisfaction of c_1 if and only if c_2 is vacuously satisfied.

Proposition 3. *Let $c_1 : X_1 \to Y_1$ be in conflict with $c_2 : X_2 \to Y_2$ via f_X and f_Y, then G satisfies f_X if and only if G vacuously satisfies c_2.*

Definition 6 (Weak Constraint). *If c_1 is in conflict with c_2 via f_X and f_Y, then the* weak constraint $c_1(c_2)$ *for c_1 with respect to c_2 is the conditional constraint $c_1(c_2) = (f_X, c_1)$.*

Proposition 4. *If c_1 is in conflict with c_2, then the weak constraint $c_1(c_2)$ is not in conflict with c_2.*

Weakening a constraint is one strategy to solve conflicts. A general discussion of strategies is outlined in [KMPP01b]. It is worth stressing that determining a conflict between constraints can be performed statically and automatically.

4 Rule-Rule Conflicts

Two rules are in a *potential-conflict (p-conflict)* if they do (partly) the same things but under different conditions. A *conflict* occurs if p-conflicting rules can be applied to a common graph. If the choice for one rule in a conflict may prevent the applicability of the other rule, the conflict is called *critical*, otherwise it is a *choice conflict*. The LBAC rule `new object` and the access control list (ACL) rule `create object` in Figure 5 are in p-conflict, since both rules create a new object node Ox. An ACL (such as the one in UNIX) is a structure that stores the access rights to an object with the object itself. The rule `create object` specifies the creation of an object by a process that runs on behalf of a user. Initially, there are no access rights to the new object and the user becomes the owner of the new object[1].

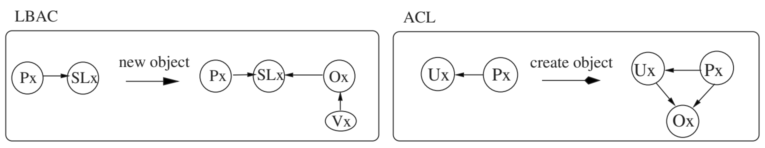

Fig. 5. *The LBAC rule* `new object` *and the ACL rule* `create object`*.*

The rule `new object` creates an object with a security level, the rule `create object` an object without one. Which rule shall be applied to introduce a new object in the system? A static analysis of the rules can detect the critical and the choice conflicts before run-time so that rules can be changed to avoid conflicts.

Definition 7 (p-Conflict, Conflict Pair, Conflict). *Rules $p_i : L_i \xrightarrow{r_i} R_i$, $i = 1,2$, with NAC $A(p_i)$ are in* p-conflict *if there is a common non-empty subrule*[2] *for p_1 and p_2. Each pair of matches $(m_1 : L_1 \to G, m_2 : L_2 \to G)$ is a* conflict pair *for p_1 and p_2. The rules p_1 and p_2 are in* conflict*, if they are in p-conflict and there is a conflict pair for p_1 and p_2. Otherwise, they are called* conflict-free.

Generally, there exist an infinite number of matches for one rule, so the set of matches must be reduced for a static analysis. To detect a rule conflict, it is sufficient to consider the left-hand sides of the rules.

[1] The complete specification of the framework for the ACL is given in [KMPP01a].

[2] A rule $p_0 : L_0 \xrightarrow{r_0} R_0$ is a subrule of rule $p : L \xrightarrow{r} R$ if there are total morphisms $f_L : L_0 \to L$ and $f_R : R_0 \to R$ with $r \circ f_L = f_R \circ r_0$.

Definition 8 (Set of Conflict Pairs). *The* set $CP(p_1, p_2)$ of conflict pairs *for rules* $(p_i : L_i \xrightarrow{r_i} R_i, A(p_i))$*,* $i = 1, 2$*, consists of all pairs of matches* $(m_1 : L_1 \to G, m_2 : L_2 \to G)$*, where* m_1 *and* m_2 *are jointly surjective.*

The set of conflict pairs for two rules in a p-conflict consists of a finite number of pairs since the left-hand side of a rule is a finite graph.

Proposition 5 (Conflict Freeness). *Let* $CP(p_1, p_2)$ *be the set of conflict pairs for the p-conflicting rules* $(p_1 : L_1 \xrightarrow{r_1} R_1, A(p_1))$ *and* $(p_2 : L_2 \xrightarrow{r_2} R_2, A(p_2))$*. Then, the rules* p_1 *and* p_2 *are conflict-free if and only if* $CP(p_1, p_2)$ *is empty.*

The set of conflict pairs for rules may be split into *choice* and *conflict* critical pairs: in the latter, after applying p_1 at match m_1, the rule p_2 is no longer applicable at m_2 or vice versa, while in the former, the order does not matter and after applying p_1 at m_1, p_2 is still applicable and vice versa. Critical and choice conflict pairs are detected by the concept of *parallel independence* [EHK+97].

Definition 9 (Parallel Independence). *Given rules* $(p_i : L_i \xrightarrow{r_i} R_i, A(p_i))$ *,* $i = 1, 2$*, the derivations* $G \overset{p_1}{\Rightarrow} H_1$ *and* $G \overset{p_2}{\Rightarrow} H_2$ *are* parallel independent *if* $r_2^* \circ m_1$ *is total and satisfies* $A(p_1)$ *and* $r_1^* \circ m_2$ *is total and satisfies* $A(p_2)$*. Otherwise, the derivations are called* parallel dependent*.*

$$\begin{tikzcd}
R_1 \arrow[d] & L_1 \arrow[l, "r_1"'] \arrow[dr, "m_1"] & & L_2 \arrow[r, "r_2"] \arrow[dl, "m_2"'] & R_2 \arrow[d] \\
H_1 & & G \arrow[ll, "r_1^*"] \arrow[rr, "r_2^*"'] & & H_2
\end{tikzcd}$$

In the case of parallel independence, the application of p_1 at m_1 and the delayed application of p_2 at $r_1^* \circ m_2$ results in the same graph (up to isomorphism) as the application of p_2 at m_2 and the delayed application of p_1 at $r_2^* \circ m_1$.

Definition 10 (Choice and Critical Conflict Pair). *A conflict pair* (m_1, m_2) *for rules* p_1 *and* p_2 *is a* choice conflict *if the derivations* $G \overset{p_1,m_1}{\Rightarrow} H_1$ *and* $G \overset{p_2,m_2}{\Rightarrow} H_2$ *are* parallel independent*. It is a* critical conflict *otherwise.*

We propose two strategies to solve rule conflicts. In the first strategy, we take one rule p_1 as *major rule*, and one p_2 as *minor rule*. For a conflict pair (m_1, m_2), p_2 is changed by adding a NAC that forbids its application at match m_2 if p_1 can be applied at m_1. The second strategy integrates the rules into one rule.

Definition 11 (Weak Condition, Weak Rule). *Given a conflict pair* (m_1, m_2) *for rules* $(p_i : L_i \xrightarrow{r_i} R_i, A(p_i))$*,* $i = 1, 2$*, the* weak condition *for* p_2 *w.r.t.* (m_1, m_2)*, denoted by* $WC(p_1, p_2, (m_1, m_2))$*, is given by the NAC* (L_2, N)*,* *where the outer diagram is a pullback and the diagram (1) is a pushout diagram.*

$$\begin{tikzcd}
 & S \arrow[dl] \arrow[dr] \arrow[d, phantom, "(1)"] & \\
L_1 \arrow[r] \arrow[dr, "m_1"'] & N & L_2 \arrow[l, "n"] \arrow[dl, "m_2"] \\
 & G &
\end{tikzcd}$$

The rule p_2 with this added NAC is called weak rule.

The weak condition for the minor rule ensures that the major and the minor rule cannot be both applied to a common system graph at match m_1 and m_2.

Example 4 (weak rule). The top of Figure 6 shows the p-conflicting ACL rule `create object` and the LBAC rule `new object`. Conflict pairs for these rules are the inclusions $(in_1 : L_1 \to L_1 \oplus L_2, in_2 : L_2 \to L_1 \oplus L_2)$ of the left-hand sides into their disjoint union, and the inclusions $(in'_1 : L_1 \to G, in'_2 : L_2 \to G)$ of the left-hand sides into the graph G (the gluing of the left-hand sides over the node Px). Figure 6 shows the weak rules with respect to the second conflict pair. The weak rule for `create object` w.r.t. `new object` has a NAC that forbids the application when there is a security level for the process. Therefore, the weak rule for `create object` is only applicable to processes created with the ACL rule 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 presence of a user connected to the process. Since each user is connected to a process, the rule is not applicable to processes created by ACL rules.

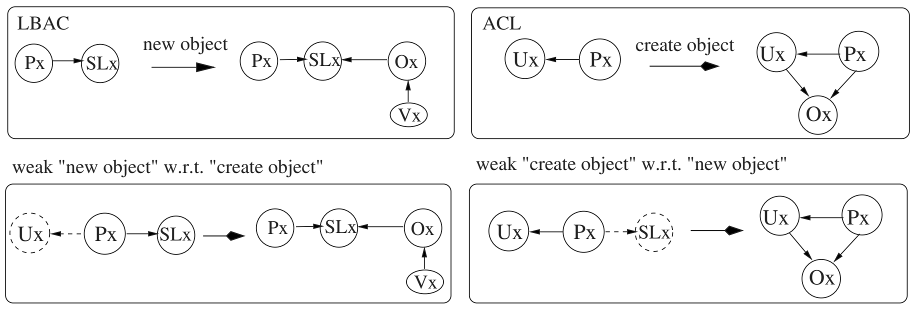
LBAC
new object
Px
SLx
Ox
Vx
ACL
create object
Ux
Px
Ox
weak "new object" w.r.t. "create object"
weak "create object" w.r.t. "new object"

Fig. 6. *The weak rules for* `new object` *and* `create object`*.*

Theorem 1 (Weak Rule is Conflict-free). *Given the set of conflict pairs $CP(p_1, p_2)$ for p_1 and p_2, the rule p_1 and the rule p_2, extended by $WC(p_1, p_2, (m_1, m_2))$ for each $(m_1, m_2) \in CP(p_1, p_2)$, are conflict-free.*

The second solution for solving conflicts between rules is the *amalgamation* of the p-conflicting rules over their common subrule.

Definition 12 (Integrated Rule). *Let* $(p_i : L_i \xrightarrow{r_i} R_i, A(p_i))$ *for* $i = 1, 2$ *be p-conflicting rules and* $p_0 : L_0 \xrightarrow{r_0} R_0$ *with* $f_{L_i} : L_0 \to L_i$ *and* $f_{R_i} : R_0 \to R_i$ *their common subrule (cf. Figure 7).*

The integrated rule *is given by* $(p : L \xrightarrow{r} R, A(p))$, *where diagram (1) is the pushout of* f_{L_1} *and* f_{L_2}, *diagram (2) is the pushout of* f_{R_1} *and* f_{R_2} *and* r *is the universal pushout morphism.*

The set $A(p)$ *contains a NAC* $n : L \to N$ *for each pair of NACs* $n_1 : L_1 \to N_1 \in A(p_1)$ *and* $n_2 : L_2 \to N_2 \in A(p_2)$, *where* N *is the pushout of* $n_1 \circ f_{L_1}$ *and* $n_2 \circ f_{L_2}$ *and* n *is the universal pushout morphism.*

$$\begin{tikzcd}
 & & & L_0 \arrow[rr, "r_0"] \arrow[dl] \arrow[dd] & & R_0 \arrow[dl] \arrow[dd] \\
N_1 \arrow[dd] & & L_1 \arrow[ll] \arrow[rr] \arrow[dd] \arrow[dr, phantom, "(1)"] & & R_1 \arrow[dd] \arrow[dr, phantom, "(2)"] & \\
 & N_2 \arrow[dl] & & L_2 \arrow[ll] \arrow[rr] \arrow[dl] & & R_2 \arrow[dl] \\
N & & L \arrow[ll, "n"'] \arrow[rr, "r"'] & & R &
\end{tikzcd}$$

Fig. 7. *Amalgamation of p-conflicting rules.*

Example 5. Figure 8 shows the integrated rule for the rules `create object` and `new object`. Their common subrule is marked in the rules and contains the process node Px in the left-hand side and the nodes Px and Ox in the right-hand side. The integrated rule creates an object that belongs to a user, as well as a process, and that carries a security level.

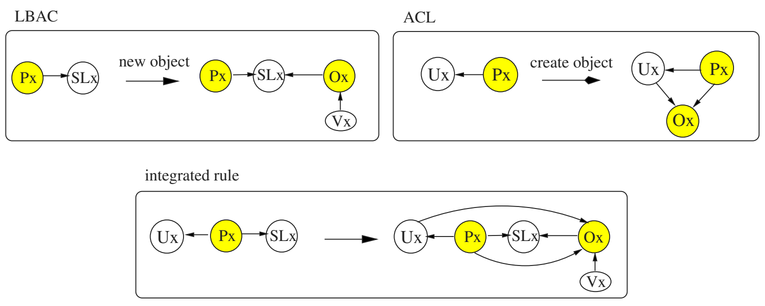

Fig. 8. *Amalgamation of p-conflicting rules* `create object` *and* `new object`.

5 Rule-Constraint Conflict

Rules can be classified into *deleting rules* that only delete graph elements, without adding anything (i.e., $dom(r) = R \subset L$) and *expanding rules* that only add graph elements, but do not delete anything (i.e., $dom(r) = L \subseteq R$).

A *conflict* between a rule and a constraint occurs when the application of the rule produces a graph which does not satisfy the constraint. The potential for conflict can be checked statically directly with the rule and the constraint without knowledge of specific graphs and derivations. A deleting rule p and a positive constraint c are in conflict if the added part required by c (i.e., $Y \setminus c(X)$) overlaps with what p removes (i.e., $L \setminus dom(r)$). Similarly, an expanding rule p conflicts with a negative constraint c if what is added by p (i.e., $R \setminus r(L)$) overlaps with something forbidden by c (i.e., $Y \setminus c(X)$).

Definition 13 (Rule-Constraint Conflicts). *Let $p : L \overset{r}{\rightharpoonup} R$ be an expanding rule and $c : X \to Y$ a constraint, then p and c are in* conflict *if there exists a nonempty graph S and injective total morphisms $s_1 : S \to R$ and $s_2 : S \to X$ so that $s_1(S) \cap (R \setminus r(L)) \neq \emptyset$.*

Let $p : L \overset{r}{\rightharpoonup} R$ be a deleting rule and $c : X \to Y$ a positive constraint, then p and c are in conflict *if there exists a nonempty graph S and injective total morphisms $s_1 : S \to L$ and $s_2 : S \to Y$ so that $s_1(S) \cap (L \setminus dom(r)) \neq \emptyset$ and $s_2(S) \cap (Y \setminus c(X)) \neq \emptyset$.*

Conflicts between rules p and constraints $c : X \to Y$ can be resolved (in favor of the constraint) by adding NACs to the rules p. For the conflict between expanding rules and negative constraints, the NACs prevent the rule from completing the conclusion of the constraint. For the conflict between expanding rules and positive constraints, the NACs prevent the rule from completing the condition X, and for the conflict between deleting rules and positive constraints, the NACs prevent the rule from destroying the conclusion Y.

Definition 14 (Reduction). *Given a rule $p : L \overset{r}{\rightharpoonup} R$ and a nonempty overlap S of R and the condition X of the constraint $c : X \to Y$.*

$$\begin{tikzcd}
L \arrow[r, harpoon, "r"] \arrow[d] & R \arrow[d, "h"'] & S \arrow[l, "s_1"'] \arrow[r, "s_2"] & X \arrow[d, "c"] \\
N \arrow[r, harpoon, "r^*"'] & C & & Y \arrow[ll]
\end{tikzcd}$$

Let C be the pushout object of $s_1 : S \to R$ and $c \circ s_2 : S \to Y$ in **Graph**, *and let $C \overset{r^{-1},h}{\Rightarrow} N$ be the derivation with the inverse rule $p^{-1} : R \overset{r^{-1}}{\to} L$ at match h. The* reduction $p(c)$ of p by c *consists of the partial morphism $L \overset{r}{\rightharpoonup} R$ and the set $A(p, c) = \{(L, N) | C \overset{(r^{-1},h)}{\Rightarrow} N,\ C = R +_S Y$ for some overlap S $\}$ of NACs.*

The construction considers arbitrary rules and constraints, i.e., it is not restricted to deleting or expanding rules, respectively. This construction reduces to the one in [HW95] if the constraint $c : X \to Y$ is the identity morphism.

Theorem 2 (Reduction preserves Satisfaction). *Let $p : L \xrightarrow{r} R$ be a rule and G a graph that satisfies the constraint $c : X \rightarrow Y$.*

1. *If c is negative, p is expanding, $p(c)$ the reduction of p by c and $G \overset{p(c)}{\Rightarrow} H$ is a derivation with $p(c)$, then H satisfies c.*
2. *If c is positive, p is expanding, $p(id_X)$ the reduction of p by $id_X : X \rightarrow X$, and $G \overset{p(id_X)}{\Rightarrow} H$ a derivation with $p(id_X)$, then H satisfies c.*
3. *If c is positive, p is deleting, $p(c) = (r, A(id_L, c))$, and $G \overset{p(c)}{\Rightarrow} H$, then H satisfies c.*

Consider the negative constraint $c(succ)$ in Figure 9 forbidding two (or more) successor levels, and the (expanding) rule `new level 2` in Figure 3 that may produce an inconsistent state by adding a successor level. We describe now, in algorithmic form, the construction of the reduction of `new level 2` by $c(succ)$:

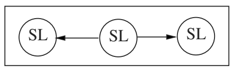

Fig. 9. *Negative constraint c(succ) forbidding more than two successor security levels.*

Step 1: Construction of all possible nonempty overlaps of R of the rule `new level 2` and the graph of $c(succ)$. Figure 10 shows the nonempty overlaps $S1$, $S2$ and $S3$ with morphisms s_1 and s_2. The remaining overlaps of R and X use the same subgraphs $S1, S2, S3$, but different morphisms s_1 and s_2.

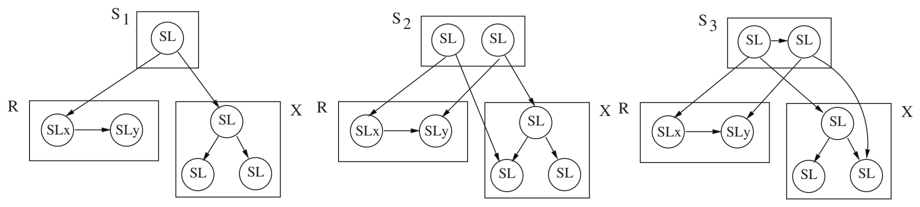

Fig. 10. *Nonempty overlaps between* `new level 2` *and* $c(succ)$.

Step 2: For each overlap S in step 1, the pushout C of the morphisms $S \rightarrow R$ and $S \rightarrow X$ is constructed. The application condition (L, N) is constructed by applying the inverse rule of `new level 2` at match $R \rightarrow C$ resulting in graph N. The inverse rule of `new level 2` deletes a security level. Figure 11 shows the pairs (L, N) for the three overlaps in Figure 10.

The construction in Definition 14 may generate redundant application conditions. In fact, if we assume that G already satisfies the constraint c, some application conditions are automatically satisfied. This corresponds to the case

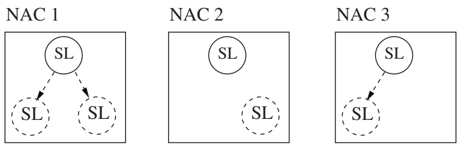

Fig. 11. *NACs constructed from the overlaps.*

where the overlap $S \to R$ can be decomposed into $S \to L \to R$. The graph N generated from such overlap can be eliminated directly from Definition 14 by requiring only overlaps S for which $s_1(S) \cap (R \setminus r(L)) \neq \emptyset$. In this manner, the application condition NAC_1 of Figure 11 can be removed.

Another form of redundancy stems from the fact that, if S_1 with morphisms s_1^1 and s_2^1 and S_2 with morphisms s_1^2 and s_2^2 are overlaps and, say, $S_1 \subseteq S_2$, $s_1^1|_{S_1} = s_1^2$, $s_2^1|_{S_1} = s_2^2$ then $C_2 = R +_{S_2} Y \subseteq C_1 = R +_{S_1} Y$ and thus $N_2 \subseteq N_1$. Hence, if a match $L \to G$ satisfies (L, N_2), then it also satisfies (L, N_1) and the application condition (L, N_1) can be removed from $A(p, c)$. For example, the overlap S_1 is included into the overlap S_3 (cf. Figure 10). Therefore, $NAC3 \subseteq NAC1$ (cf. Figure 11) and we can remove $NAC1$.

The solution of conflicts between expanding rules and negative constraints and of conflicts between deleting rules and positive constraints is a reasonable reduction of the number of system graphs which the rules can produce. The solution for conflicts between expanding rules and positive constraints, however, is not very satisfactory, since it reduces more than necessary the number of system graphs that can be generated. Another solution is a construction which extends the right-hand side of a rule so that the rule creates the entire conclusion Y of a constraint $c : X \to Y$ and not only parts of it.

Definition 15 (Completing Rule). *The* completing rule *for an expanding rule $p : L \xrightarrow{r} R$ and a positive constraint c is defined by $p^c(c) = v_i \circ h_i \circ r$, where*

$$\begin{tikzcd}
L \arrow[r, "r"] \arrow[d, "p^c(c)"'] & R \arrow[d, "h_i"] & S_i \arrow[l, "s_1^i"'] \arrow[r, "s_2^i"] & X \arrow[d, "c"] \\
R' & C_i \arrow[l, "v_i"'] & & Y \arrow[ll, "y_i"']
\end{tikzcd}$$

- *$\Omega = \{R \overset{s_1^i}{\leftarrow} S_i \overset{s_2^i}{\rightarrow} X\}$ is the set of all nonempty overlaps of R and X so that $s_1^i(S_i) \cap (R \setminus r(L)) \neq \emptyset$,*
- *for each $S_i \in \Omega$, (C_i, h_i, y_i) is the pushout of s_1^i and $c \circ s_2^i$ in* **Graph**,
- *$(R', v_i : C_i \to R')$ is the pushout of the morphisms $h_i : R \to C_i$ in* **Graph**.

The completing rule for the ACL rule `create object` and the positive constraint requiring a value for each object is shown in Figure 12.

Lemma 1. *If $p^c(c) : L \to R'$ is the completing rule for p, c, then R' satisfies c.*

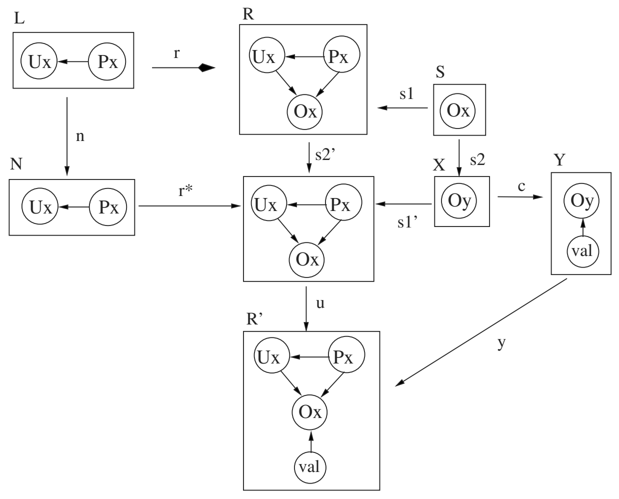

Fig. 12. *Construction of the completing rule.*

The completing rule, however, does not preserve consistency for each positive constraint. If we restrict positive constraints to *single* (X contains at most one node) or *edge-retricted* (for each edge $s \xrightarrow{e} t \in (Y \setminus c(X))$, $s, t \in (Y \setminus c(X))$), the construction results always in a consistence preserving rule.

Proposition 6. *If c is a single or edge-retricted positive constraint, the completing rule $p^c(c)$ for a rule p is consistent with respect to c.*

The construction of the completing rule could be generalized to arbitrary constraints by using *set nodes*: a set node in the left-hand side of a rule matches all occurrences of this node in a graph and the rule is applied to all the occurrences.

Another possibility to solve conflicts between positive constraints and expanding rules p is to transform the constraint $X \to Y$ into a rule and require that this rule is applied (after the application of p) as long as there are occurrences of X not "visited" in H. The new rule is just the constraint $X \to Y$ with negative application condition (X, Y) to avoid its application repeatedly on the same part of H. It is neccessary to add "control" on the framework to ensure that this new rule is applied 'as long as possible'. Control can be introduced either by using rule expressions [GRPPS00] or transformation units [EKMR99] as an encapsulation mechanism used in a way similar to procedure calls.

6 Concluding Remarks

In a graph-based approach to the specification of 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). An important problem addressed here is how to deal with inconsistencies caused by conflicts between two of the constraints, two of the rules or between a rule and a constraint. Often such problems arise when trying to predict the behavior of an AC policy obtained by integrating two separate coherent policies [KMPP01a]. The conflict between a rule of one policy and a simple constraint of the other policy has been addressed in part elsewhere [KMPP00], where it is also shown the adequacy of this framework to represent a Role-based Access Control policy. Here we have tackled the problem of conflicts by making effective use of the graph based formalism. Conflicts are detected and resolved statically by using standard formal tools typical of this graph based formalism. In the process, we have introduced the notions of conditional constraint and of weakening of a rule.

A tool, based on a generic graph transformation engine, is under development to assist in the systematic detection and resolution of conflicts and in the stepwise modification of an evolving policy while maintaining its coherence.

References

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.

EKMR99. H. Ehrig, H.-J. Kreowski, U. Montanari, and G. Rozenberg, editors. *Handbook of Graph Grammars and Computing by Graph Transformations. Vol. III: Concurrency, Parallelism, and Distribution.* World Scientific, 1999.

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 Lect. Notes in Comp. Sci., pages 368–382. Springer, 2000.

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.

KMPP00. 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 Lect. Notes in Comp. Sci., 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 S. Osborne, editor, *Proc. 6th ACM Symp. on Access Control Models and Technologies*, pages 121–130. ACM, May 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 F.Honsell and M.Miculan, editors, *Proc. of Foundations of Software Science and Computation Structures (FoSSaCS 2001)*, number 2030 in Lect. Notes in Comp. Sci., pages 287–302. Springer, 2001.

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.