Available online at www.sciencedirect.com
SCIENCE DIRECT®
ELSEVIER
Journal of Computer and System Sciences 71 (2005) 1–33
JOURNAL OF COMPUTER AND SYSTEM SCIENCES
www.elsevier.com/locate/jcss

Graph-based specification of access control policies☆

Manuel Koch[a], L.V. Mancini[b], Francesco Parisi-Presicce[c,]*

[a]*Freie Universität Berlin, Berlin*
[b]*Univ. di Roma "La Sapienza", Roma*
[c]*Information and Software Engineering Department, George Mason University, 4400 University Dr., Fairfax, VA 22030, USA*

Received 19 November 2003; received in revised form 4 November 2004

Available online 18 January 2005

Abstract

Graph-based specification formalisms for access control (AC) policies combine the advantages of an intuitive visual framework with a rigorous semantical foundation that allows the detailed comparison of different policy models. 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. Several models for AC (e.g. role-based, lattice-based or an access control list) can be specified in this framework. 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 inconsistent policies. Using formal properties of graph transformations, it is possible to systematically detect inconsistencies between constraints, between rules and between a rule and a constraint and lay the foundation for their resolutions.
© 2004 Elsevier Inc. All rights reserved.

Keywords: Security; Access control; Graph transformation; Graphical constraints; Consistency; Conflict detection; Policy verification

☆ Partially supported by the EC under Research and Training Network *Segravis* and by the Italian MIUR under the project PRIN 2003: *WEB-based management of spatial and geographic data*. A preliminary version of some of the results has been presented at FoSSaCS2001 and FoSSaCS2002 and has appeared in the Proceedings published by Springer-Verlag.

* Corresponding author.

E-mail addresses: mkoch@inf.fu-berlin.de (M. Koch), lv.mancini@di.uniroma1.it (L.V. Mancini), fparisi@ise.gmu.edu, fparisip@gmu.edu (F. Parisi-Presicce).

0022-0000/$ - see front matter © 2004 Elsevier Inc. All rights reserved.
doi:10.1016/j.jcss.2004.11.002

1. Introduction

A considerable amount of work has been carried out recently on models and languages for access control (AC). AC is concerned with determining the activities of legitimate users [19], and is usually 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. The three main AC policies commonly used in computer systems are discretionary policies [19], lattice-based policies (also called mandatory policies) [17] and role-based policies [18].

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 [10,12] we have proposed a formalism based on graphs and graph transformations for the specification of AC policies. This conceptual framework, used in [10,12] to specify role-based policies, a lattice-based access control (LBAC) policy and an access control list (ACL) (example of a discretionary policy), allows the uniform comparison of these different models, often specified in ad hoc languages and requiring ad hoc conversions to compare their relative strengths and weaknesses.

Our graph-based specification formalism for AC policies combines the advantages of an intuitive visual framework with the rigor and precision of a semantics 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 only to illustrate the different concepts, with no pretence 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 evolution assistant.

The paper is organized as follows: Section 2 briefly reviews lattice-based access control; Section 3 presents the basic formalism of graph transformations using lattice-based access control to illustrate it; Section 4 defines the formal framework to specify AC policies (its main properties are relegated in the appendix); Section 5 deals with the integration of policies and Section 6 discusses the notion of coherence of a security policy framework. Section 7 discusses analysis and management of conflicts between constraints and between rules, while Section 8 discusses conflicts between a rule and a constraint; the last section mentions related and future work.

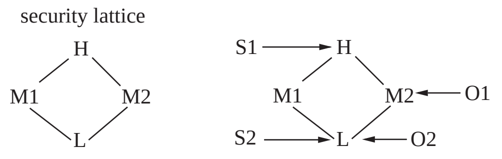

Fig. 1. A security lattice (left-hand side) and the assignment of security levels to subjects and objects (right-hand side).

2. Lattice-based access control

Classic lattice-based access control (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, where the security levels available are H (highest), L (lowest) and M1, M2 (middle).

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. a file or a directory 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 $\lambda(S) \geqslant \lambda(O)$ and to write object O if $\lambda(S) = \lambda(O)$.

The subjects in Fig. 1 on the right-hand side are $S1$ and $S2$, with security levels $\lambda(S1) = H$ and $\lambda(S2) = L$. The objects are $O1$ and $O2$ with security levels $\lambda(O1) = M2$ and $\lambda(O2) = L$, respectively. The LBAC rules ensure that $S1$ can read both objects $O1$ and $O2$ but cannot write either $O1$ or $O2$. Subject $S2$ can read and write object $O2$, but neither read nor write object $O1$. In the Bell-LaPadula model [2], subjects are allowed to write “blindly” in objects that they cannot read. In such a model, $S2$ can write in both objects $O1$ and $O2$, but is still able to read only $O2$.

3. Graph transformations

This section introduces the basic definitions and notation for graph transformations [16]. Parts of the LBAC model are used throughout the section to illustrate the explanations by examples.

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 $l_G : G_V \cup G_E \to Labels$ assigning a label to each node and to each edge. Labels are elements of a disjoint union of sets $Labels = X \cup C$, where X is a set of *variables* and C is a set of *constants*. A binary relation $\prec \subseteq Labels \times Labels$ is defined on *Labels* as $(\alpha, \beta) \in \prec$ if and only if $\alpha \in X$. This binary relation is not a partial order since several distinct variables may be needed [15]. The relation $\alpha \prec \beta$ indicates that (the variable) α can be substituted by β (which, in turn, can be either a variable or a constant)

A path of unspecified length between nodes a and b is indicated by an edge $a \xrightarrow{*} b$ which can be seen as an abbreviation for a set of paths, each representing a possible sequence of edges between a and b.

[1] In [17], security levels are called security labels. We use ‘security level’ here to avoid confusion with the notion of a label for a node or for an edge in a graph.

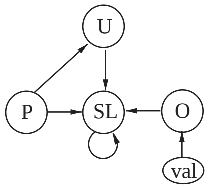

Fig. 2. The type graph for the LBAC model.

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 $f_V \circ t_G = t_H \circ f_E$, respect 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 variable can be replaced by a constant or by another variable), and respect the substitutions, i.e. if $l_G(v_1) = l_G(v_2)$, then $l_H(f_V(v_1)) = l_H(f_V(v_2))$ for all $v_1, v_2 \in G_V$ and $l_G(e_1) = l_G(e_2)$, then $l_H(f_E(e_1)) = l_H(f_E(e_2))$ for all $e_1, e_2 \in G_E$ (different instances of the same variable are substituted with the same value). 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 one of them, say f_p, is *label preserving*, i.e. $l_G(x) = l_H(f_p(x))$ for each $x \in G$ (node or edge) [15].

A *type graph TG* represents the type information in a graph transformation system [3] and it specifies the node and edge types which may occur in the instance graphs modeling system states.

For example, the type graph in Fig. 2 shows the possible types for the LBAC graph model. It provides the node types U, O, P, val and *SL*. The node U is the type of the nodes representing users, the node O the objects, the node *val* the actual information in objects and the node P the processes that run on behalf of users. The node *SL* with its loop represents a whole security lattice, that is, a partial order on security levels as e.g., $High > Middle1 > Low$, $High > Middle2 > Low$. The node *SL* can be any security level (e.g. $High$, $Middle1$, $Middle2$, Low) 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 absence in the type graph of an arc between the nodes P and O indicates that there cannot be, in *any* instance graph, a direct (access control) “connection” between a process and an object. The presence in the type graph of an arc from O to *SL* indicates that an object *may* be associated with security levels, but is not required to be (this requirement is expressed by the constraints in Fig. 6)

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. From now on, in all our figures the typing morphism maps a node with label *Tx* to the type node *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** [3]. 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 by *f* of a graph $\langle G, t_G \rangle$ typed over *TG* is the graph $\langle G, f \circ t_G \rangle$ typed over TG'. 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* [3,6].

General Assumption. In the following, we fix a type graph *TG*, and all graphs and morphisms are from the category **TG** if the type is not explicitly stated.

Notice that types are used to establish similarities among different entities (nodes and edges) while labels are used to distinguish among similar entities.

A *graph rule* $p : r$, or just *rule*, is given by a rule name *p*, from a set *RNames*, and a label preserving *injective morphism* $r : L \rightharpoonup R$. The graph *L*, *left-hand side*, describes the elements a graph must contain for the rule *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. Note that the actual deletions/additions are performed on the graphs to which the rule is applied.

Example 1 (*LBAC graph rules*). Fig. 3 shows the schemes for the rules of the LBAC policy. The labels for the nodes (Ux, Px, SLx, SLy, ...) of the rules are variables taken from the set of variables in Labels.

The rule `new object` creates a new object *Ox* connected to a node *valx* (the initial value of the object). The object *Ox* is given the security level *SLx*. The variable *SLx* is generic: it 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 *Px* is attached to a security level *SLy* that is no higher than the security level *SLx* of the user *Ux* in the security lattice graph. This

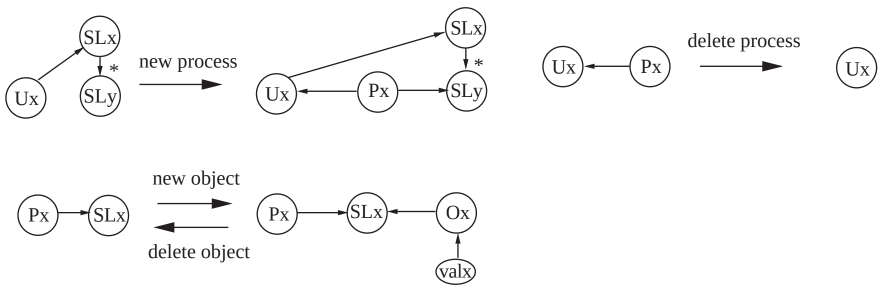

Fig. 3. Graph rules for the LBAC policy.

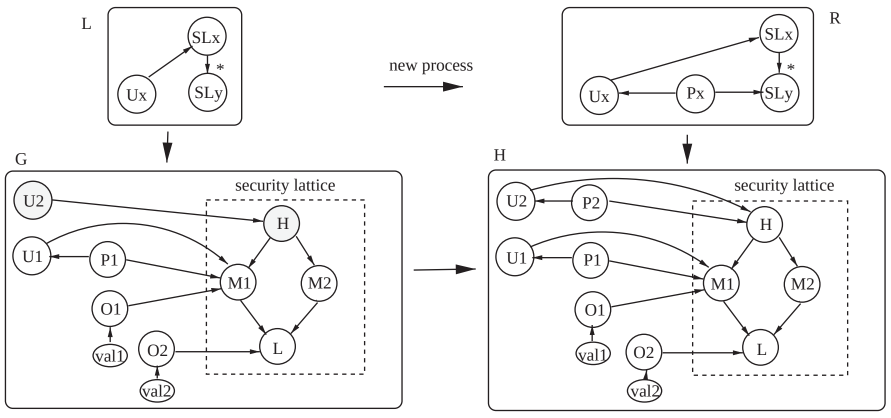

Fig. 4. Application of rule `new process`.

requirement is specified by the path from *SLx* to *SLy* indicated by an edge decorated by a "*": this is a shortcut in our diagrams to denote a set of rules, each with a path of different length (possibly zero) connecting the nodes labeled *SLx* and *SLy* consisting of edges of the security lattice graph. Processes (and their connections to the user) are removed by the rule `delete process`.

For the application of rules we use the single pushout (SPO) approach to graph transformations [4]. 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 r is label preserving [15].

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

Example 2 (*Application of a graph rule*). In Fig. 4, the left-hand side L of the rule `new process` 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. The application of the rule inserts the new process node connected to the user $U2$ and the security level H.

For the specification of AC policies by graph transformations, *negative application conditions* for rules are needed [4]. A negative application condition (NAC) for a rule $p : L \overset{r}{\rightharpoonup} R$ consists of a set $A(p)$ of total injective morphisms $a_i : L \to N$, where the part $N \setminus a_i(L)$ represents a structure that must not occur in a graph G for the rule to be applicable. In the figures, all negative application conditions are simple

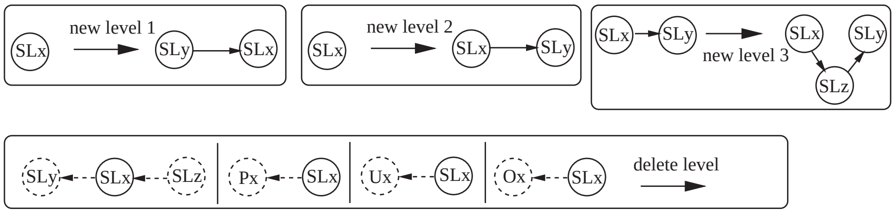

Fig. 5. LBAC rules for modifying the security lattice.

inclusions and we depict $a_i : L \to N$ by the graph N, where the subgraph L is drawn with solid lines and $N \setminus L$ with dashed lines. A rule $p : L \xrightarrow{r} R$ with a NAC $A(p)$ is applicable to G if there is a total morphism $m : L \to G$ (L occurs in G) and there is no total morphism $n_i : N \to G$ such that $m = n_i \circ a_i$ (it is not possible to extend L to N) for any $a_i : L \to N$ in $A(p)$. From now on, a *match for p with NAC* $A(p)$ is a total morphism $m : L \to G$ that cannot be extended to a total morphism $n_i : N \to G$ for any $a_i : L \to N$ in $A(p)$.

Example 3 (*Negative application condition*). Fig. 5 shows the rules for modifying the security lattice. New security levels can be inserted before an existing security level *SLx* (rule `new level 1`), after an existing security level *SLx* (`new level 2`) or between two security levels *SLx* and *SLy* (`new level 3`). The rule `delete level` removes a security level that does not connect two security levels, i.e., *SLx* has no predecessor *SLz* and *SLx* has no sucessor *SLy* (expressed by the first pair (L, N) of the NAC). Therefore, it is not possible to delete a security level between two security levels, to ensures that the security level hierarchy remains connected. More complex rules can specify the deletion between security levels, but they are not introduced here. Since users, processes and objects need a security level, security levels cannot be removed if any user, process or object possesses this security level. Therefore, the NAC of the rule `delete level` has also the following three pairs (L, N): one to prevent the deletion of a security level that belongs to a process (the NAC with dashed node *Px*), the second one concerns the users (dashed node *Ux*) and the third one the objects (dashed node *Ox*). Only if (each condition in) the NAC is satisfied, a security level can be removed.

4. Security policy framework

This section presents the framework for the specification of AC policies based on graph transformations [9]. 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, and 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. Since in some AC policies it is meaningful to restrict the set of system graphs constructed by the graph rules (as not all of them represent valid states), a security policy framework contains also two sets of *constraints*. Constraints can be *negative constraints* to specify graphs that shall not be contained in any system graph and *positive constraints* to specify graphs that must be explicitly constructed as parts of a system graph. In any implementation of an AC policy, the constraints are not needed since the only acceptable states are those explicitly built by the implemented rules. But

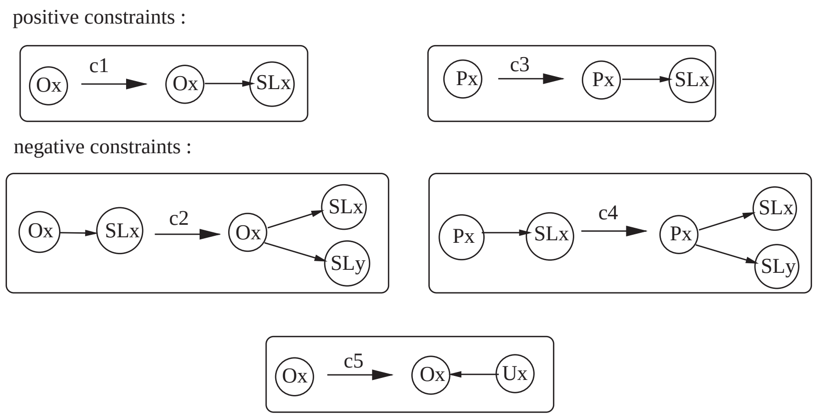

Fig. 6. Positive and negative constraints for LBAC.

when developing an AC policy through successive refinement steps, or when trying to predict the behavior of a policy, it is useful to have the additional information provided by the constraints. Positive and negative constraints can be considered as a formal documentation of the initial requirements and of the development process of rules.

Both positive and negative constraints are formally specified by morphisms [8]. It is the semantics of the morphism that distinguishes between positive and negative constraints.

Definition 4 (*Constraints*). A *constraint* (positive or negative) is given by a total graph morphism $c : X \to Y$.

Definition 5 (*Constraint satisfaction*). A total injective graph morphism $k : X \to G$ *satisfies* a positive (negative) constraint $c : X \to Y$ if there exists (does not exist) a total injective graph morphism $q : Y \to G$ such that $X \xrightarrow{c} Y \xrightarrow{q} G = X \xrightarrow{k} G$. A graph G *satisfies* a positive (negative) constraint c if each total injective graph morphism $k : X \to G$ satisfies c.

A graph G *vacuously* satisfies $c : X \to Y$ if there is no total graph morphism $k : X \to G$; G *properly* satisfies c otherwise.

A negative constraint $c : X \to Y$ is equivalent with respect to satisfaction to the simpler negative constraint $c_Y : Y \to Y$. The former form is more intuitive for a policy designer than the latter one, in the sense that it is easier to see exactly which parts are allowed (X) if the remaining parts ($Y \setminus X$) do not occur.

Example 6 (*Constraints for LBAC*). Fig. 6 shows positive and negative constraints for the LBAC model. The positive constraint $c1$ and the negative constraint $c2$ require that objects always have a security level (the positive constraint) and that there does not exist more than one security level for each object (negative constraint). The constraints $c3$ and $c4$ specify the same existence and uniqueness requirements for subjects. In mandatory access control policies, there is usually no concept of an owner of an object.

Since LBAC belongs to the class of mandatory access control policies, the negative constraint $c5$ forbids an edge between a user and an object to indicate object ownership by users. Actually, this is already expressed in the LBAC type graph. If we consider the evolution of access control policies, the constraint c prevents changing the policy to a policy with a concept of ownership.

We are now ready to define a security policy framework, which is characterized by a type graph (of all the graphs involved), by a set of rules (from the set of all rules built with instances of that type graph) with associated names, and sets of constraints. Formally:

Definition 7 (*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 P and a total mapping $r_P : P \to |\mathbf{Rule}(TG)|$ mapping each rule name to a rule $L \overset{r}{\rightharpoonup} R$ of *TG*-typed graphs,
- *Pos* is a set of positive constraints, and *Neg* is a set of negative constraints.

The security policy framework for the LBAC policy consists of the type graph in Fig. 2 and the negative c_2, c_4, c_5 and positive c_1, c_3 constraints in Fig. 6. The rule names $\{newprocess, deleteprocess, newobject, deleteobject\}$ 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 structure of the rules in the sense that the rule corresponding to the name $f_P(x)$ reduces to the rule corresponding to the name x if the retyping induced by f_{TG} is forgotten. More precisely:

Definition 8 (*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 $p \in P_1$.

Note that the definition of a framework morphism does not constrain the sets of constraints.

Example 9 (*Framework morphism*). Consider, as an example, the framework morphism $f : SP_1 \to SP_2$ with the total graph morphism f_{TG} in Fig. 7 between the type graphs TG_1 and TG_2. The intended meaning of this morphism is that the types A and B of the security framework SP_1 are renamed to C and D, respectively, and that there is a new type E in the security framework SP_2. The rule $p1$ of $SP1$ can be mapped to the rule $p2$ of SP_2, since the application of the forgetful functor $V_{f_{TG}}$ to $p2$ yields the rule $p1$. The rule $p1$ cannot be mapped to the rule p_2' since the forgetful functor yields a rule different from $p1$.

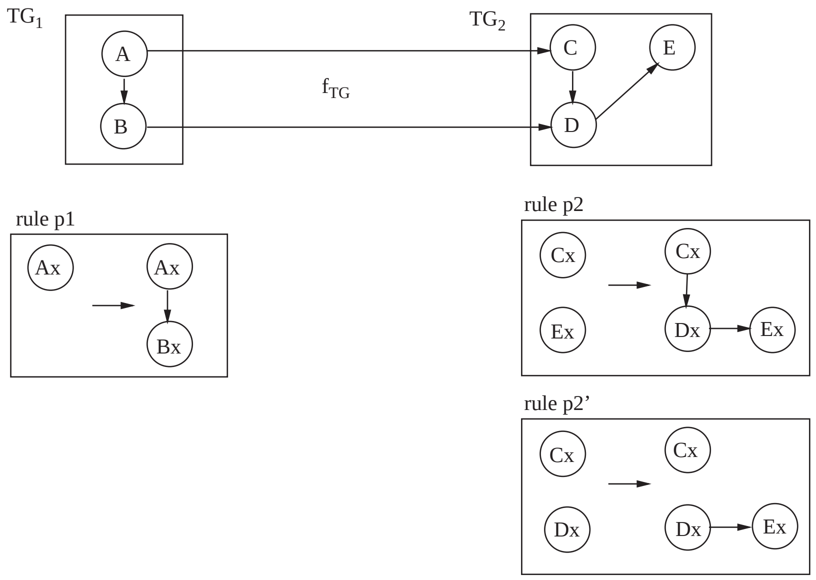

Fig. 7. Framework morphism.

5. Access control policy integration 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. We consider here only the integration on the syntactical level and omit the integration on the semantical level (semantical integration is considered in more detail in [10]). The integration of two AC policies on the syntactical level is a pushout of the security policy frameworks in the category **SP** (see the appendix). 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 formally 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*. Informally, it is the *union* of the two policies SP_1 and SP_2 where the common subpolicy SP_0 is not duplicated.

Throughout this section, the integration of the lattice-based access control (LBAC) framework with an ACL framework (introduced next in Section 5.1) is used as an example.

5.1. *Access control list*

The access control list (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 rights and can change the access permissions of the object with respect to the world.

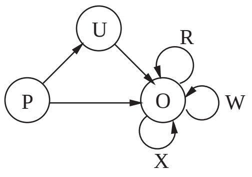

Fig. 8. The type graph for the ACL.

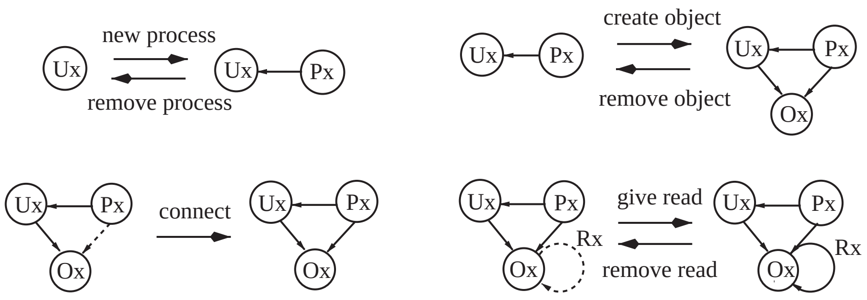

Fig. 9. Graph rules for the ACL model.

Example 10 (*Graph rules for the ACL*). The type graph TG_{ACL} in Fig. 8 provides the node types *U*, *O* and *P*. Just as in the LBAC model, a node of type *U* represents a user, a node of type *O* an object, and a node of type *P* a process. An edge between a user node *U* and an object node *O* specifies that *U* is the owner of the object *O*. 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. 9. 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 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 to the world the read permission on an object, provided that it has not already been granted. Other rules such as `give write` and `give execution` are similar and not shown.

Example 11 (*Constraints for ACL*). The constraints for the ACL framework in Fig. 10 require that each process belongs to a unique user (the positive constraint $d1$ and the negative constraint $d2$), that each object belongs to a unique user (the positive constraint $d3$ and the negative constraint $d4$) and that there is at most one permission loop with the same permission attached to the same object (negative constraint $d5$). Note that the last diagram represents three negative constraints, one for *R*, one for *W* and one for *X*.

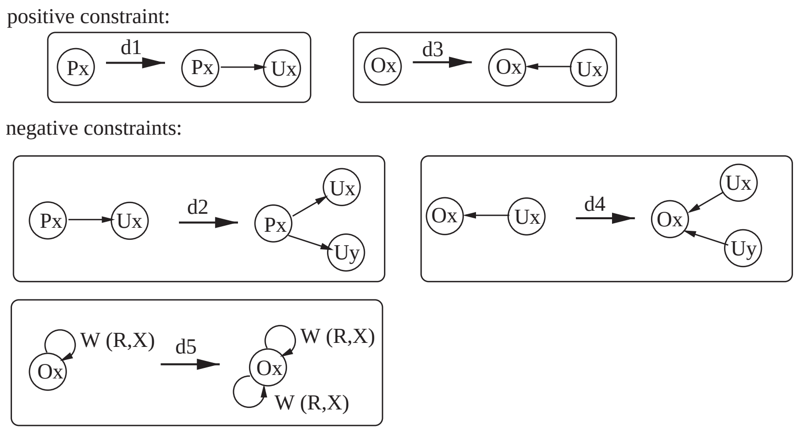

Fig. 10. Positive and negative constraints for the ACL.

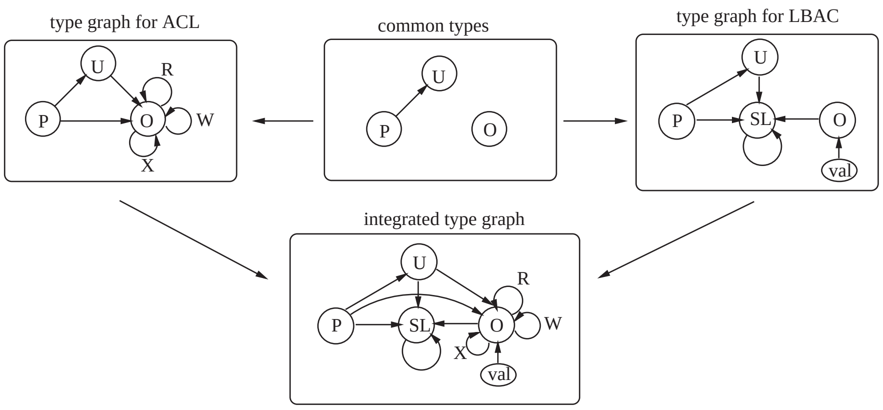

Fig. 11. Integrated type graph for the combined LBAC and ACL security model.

5.2. *Integration of ACL and LBAC*

The type graph in the middle of Fig. 11 shows the types common to ACL and LBAC. The *U*, *O* and the *P* type nodes are to be considered the same in both models. The edge between the *U* and the *P* node is a common part as well. The gluing (pushout) of the two type graphs is the type graph at the bottom of Fig. 11.

All rules are kept in the integrated security policy framework, but their component graphs are now typed over the integrated type graph. The constraints of the integrated policy framework in this example are given by the union of the constraint sets of the LBAC model (now typed over the integrated type graph) and the ACL model. Other combinations of the sets of constraints are possible (see the appendix and the next section).

6. Coherence

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. We Have added the constraints to the security policy framework to have a declarative description of wanted and unwanted system graphs, but we have not related the constraints with the rules of a framework. Do the rules construct all the wanted system graphs required by the positive constraints and do the rules prevent the construction of unwanted system states expressed by the negative constraints? If the rules do so, the framework is called *coherent*.

Definition 12 (*Coherence*). A security policy framework is *positive* (resp. *negative*) *coherent* if all system graphs satisfy the constraints in *Pos* (resp. *Neg*).

A security policy framework is *coherent* if it is both positive and negative coherent.

If we consider the integrated security framework in Section 5, in which the set of rules and constraints is constructed by the union of rules and constraints of the LBAC security framework and the ACL security framework, we realize that the integrated framework is not coherent. An example of such an inconsistency is given by the LBAC constraints $c1$ and $c3$ in Fig. 6, which require a security level for each object and process, and by the ACL rules that create objects and processes without a security level, generating graphs that do not satisfy the constraints $c1$ and $c3$. Moreover, we have now both the negative LBAC constraint $c5$ which forbids an owner for any object and the positive ACL constraint $d3$ which requires an owner for each object. The constraints are in conflict in the sense that it is not possible to find a graph satisfying both constraints at the same time.

Beside the conflicts that render a framework incoherent, because of the rules that produce graphs which does not satisfy the constraints, we have conflicts even if the rules may produce only graphs that satisfy the constraints. These conflicts occur between rules stemming from different component policy frameworks. Consider as an example the LBAC rule `new object` in Fig. 3 and the ACL rule `create object` in Fig. 12. The rule `create object` creates an object with a security level, the rule `new object` an object without one. Which rule shall be applied in this type of conflict?

The examples show that an integration of previously coherent frameworks does not lead in general to a coherent framework. Problems may occur between a rule and a constraint, between two or more constraints or between two or more rules. We investigate first how the pushout preserves coherence. Since the pushout cannot guarantee conflict-freeness, in the following sections we consider conflict management strategies to resolve conflicting constraints, conflicting rules and conflicts between a rule and a constraint.

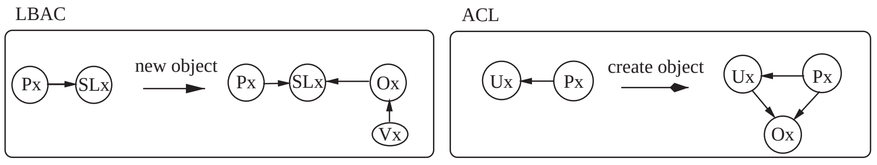

Fig. 12. Two rules for creating objects: the rule `new object` of the LBAC model (left-hand side) and the rule `create object` of the ACL (right-hand side).

The question now is:
if the frameworks SP_1 and SP_2 are coherent, is the pushout *SP* also coherent? As shown before, this is not generally the case. If we choose the intersection as the operation to construct the constraint sets of the pushout framework (see Theorem A.4), coherence is preserved.

Proposition 13 (*Preservation of coherence*). *Given the coherent frameworks SP_1 and SP_2 and the framework morphisms $f_1 : SP_0 \to SP_1$ and $f_2 : SP_0 \to SP_2$, the pushout object $SP = (TG^{\text{int}}, (P^{\text{int}}, r_{P^{\text{int}}}), Pos^{\text{int}}, Neg^{\text{int}})$ of f_1 and f_2 in* **SP** *with $Pos^{\text{int}} = F_{g'_{TG}}(Pos_1) \cap F_{f'_{TG}}(Pos_2)$ and $Neg^{\text{int}} = F_{g'_{TG}}(Neg_1) \cap F_{f'_{TG}}(Neg_2)$ is coherent.*

Proof. Each constraint in Neg^{int} is also given in Neg_1 and Neg_2 (up to re-typing). The intersection requires that the negative constraints in Neg^{int} refer to common types only. Let *G* be a graph generated by the rules in *SP* and $c \in Neg^{\text{int}}$. Since the policies SP_1 and SP_2 are coherent, their rules do not create a graph that does not satisfy *c* (after re-typing). If the rules of SP_1 and SP_2 are not identified by *f* and *g*, they occur up to re-typing in *SP*, i.e., they create in *SP* the same graphs as in SP_1 and SP_2, respectively, all satisfying *c*. If a rule p_1 from SP_1 and a rule p_2 from SP_2 are identified by *f* and *g*, their amalgamated rule is constructed in *SP*. The rules p_1 and p_2 perform the same action on the common types, as they differ only on the non-common types. Since the constraint *c*, however, refers only to the common types, the amalgamated rule in *SP* cannot create more on the common types than the component rules p_1 and p_2. Since they construct only coherent graphs, so does the amalgamated rule in *SP*.

The argument for the satisfaction of constraints in Pos^{int} is similar. □

Coherence with respect to the union operation on constraints (in Theorem A.4) is generally not preserved by the pushout construction, as the ACL-LBAC integration example shows. The positive ACL constraint *d*3, which requires a user for each object, is satisfied by the ACL rules, but the integrated framework contains also the LBAC rules and, in particular, the rule `new object`, so that graphs that do not satisfy *d*3 can be constructed.

The reason for the incoherence with respect to the constraints, in the case where the union operation is used, can be reduced to the parts of the constraints referring to the common types. Coherence of constraints referring to types occurring only in SP_1 or only in SP_2 is preserved.

Proposition 14. *Given the coherent frameworks SP_1 and SP_2 and the framework morphisms $f_1 : SP_0 \to SP_1$ and $f_2 : SP_0 \to SP_2$, the pushout $SP = (TG^{\text{int}}, (P^{\text{int}}, r_{P^{\text{int}}}), Pos^{\text{int}}, Neg^{\text{int}})$ of f_1 and f_2 in* **SP** *with $Pos^{\text{int}} F_{g'_{TG}}(Pos_1) \cup F_{f'_{TG}}(Pos_2)$ and $Neg^{\text{int}} F_{g'_{TG}}(Neg_1) \cup F_{f'_{TG}}(Neg_2)$ is incoherent if and only if SP is incoherent with respect to the constraints containing types in TG_0.*

Proof. The direction $\Leftarrow$ follows by definition. For the direction $\Rightarrow$, consider a graph *G* generated by the rules of *SP* that does not satisfy a constraint *c* of *SP*. If *c* is a constraint that refers to non-common types occurring only in SP_1 (resp. SP_2), only the rules of SP_1 (SP_2) are concerned with these types. By pushout construction, the rules of SP_1 (SP_2) occur up to re-typing as rules or sub-rules in *SP*. There are no other actions on non-common types in *SP* than those in SP_1 (SP_2). Therefore, the generated graph structures with respect to the non-common types in SP_1 and *SP* coincide and each constraint referring only to non-common types is satisfied. □

7. Conflict management

The previous section shows that the satisfaction of constraints is generally not preserved when policies are combined. By restricting the construction of the set of constraints, coherence can be achieved in some cases, but there are still many practical cases that lead to inconsistent frameworks. Therefore, this section investigates static conflict detection and automatic conflict resolution strategies to transform an incoherent framework into a coherent framework.

We deal first with conflicts between constraints, then consider conflicts between rules. The next section deals with conflicts between rules and constraints.

7.1. Constraint–constraint conflict

In this section we discuss the problem of a security policy framework having constraints which require contradictory properties of a system graph.

Definition 15 (*Contradictory constraints, contradictory policy*). Two constraints are *contradictory* iff there are no graphs that properly satisfy both constraints. A security policy framework $SP = (TG, (P, r_P), Pos, Neg)$ is *contradictory* iff $Neg \cup Pos$ contains at least a pair of contradictory constraints.

An example is the integrated ACL-LBAC framework, where the ACL constraint $d3$ and the LBAC constraint $c5$ cannot be satisfied by the same graph.

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

Definition 16 (*Conflict of constraints*). Given two constraints $c_i : X_i \to Y_i$ for $i = 1, 2$, c_1 is *in conflict* with c_2 iff there exist graph morphisms $f_X : X_1 \to X_2$ and $f_Y : Y_1 \to Y_2$ such that $f_Y \circ c_1 = c_2 \circ f_X$.

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

The conflict is *strict* if the diagram is a pushout. Two constraints c_1 and c_2 are in conflict if either c_1 is in conflict with c_2 or c_2 is in conflict with c_1.

Conflicts of constraints can be classified in *critical* and *harmless*, the latter referring to constraints that contain redundant restrictions as the following result indicates. In the harmless case, if c_1 is in conflict with c_2, then c_1 is really a subconstraint of c_2.

Fact 17 (*Harmless conflicts*). (1) *If $c_1, c_2 \in Neg$ and c_1 is in conflict with c_2, then if G satisfies c_1, then G satisfies c_2.*

(2) *If $c_1, c_2 \in Pos$ and c_1 is in strict conflict with c_2, then if G satisfies c_1, then G satisfies c_2.*

Proof. (1) If $g : X_2 \to G$ then $g \circ f_X : X_1 \to G$. If G did not satisfy c_2, then there would exist $h : Y_2 \to G$ such that $h \circ c_2 = g$. But then $g \circ f_X = h \circ c_2 \circ f_X = h \circ f_Y \circ c_1$, contradicting the fact that G satisfies c_1.

(2) If $g : X_2 \to G$ then $g \circ f_X : X_1 \to G$. Since G satisfies c_1, there exists $h : Y_1 \to G$ such that $h \circ c_1 = g \circ f_X$. By the Universal Property of pushouts, there exists $k : Y_2 \to G$ such that $k \circ f_Y = h$ and $k \circ c_2 = g$. The last equality is exactly what is needed to conclude that G satisfies 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 18 (*Critical conflicts*). *If c_1 is in conflict with c_2, then*

(1) *if $c_1 \in Neg$ and $c_2 \in Pos$, then if G satisfies c_1, then G does not properly satisfy c_2,*
(2) *if $c_1 \in Pos$ and $c_2 \in Neg$, and the conflict is strict, then if G satisfies c_1, then G does not properly satisfy c_2,*
(3) *if $c_1 \in Neg$ and f_X does not satisfy c_1, then if G properly satisfies c_2 then G does not satisfy c_1.*

Proof. Let c_1 be in conflict with c_2 via f_X and f_Y.

(1) If $g : X_2 \to G$ and G satisfies c_2, then there exists $h : Y_2 \to G$ such that $h \circ c_2 = g$. But then for $g \circ f_X : X_1 \to G$ there exists $h \circ f_Y : Y_1 \to G$ such that $h \circ f_Y \circ c_1 = h \circ c_2 \circ f_X = g \circ f_X$, which says that G does not satisfy c_1.
(2) If $g : X_2 \to G$ and G satisfies c_1, then there exists $h : Y_1 \to G$ such that $h \circ c_1 = g \circ f_X$. By the Universal Property of pushouts, there exists $k : Y_2 \to G$ such that $k \circ f_Y = h$ and $k \circ c_2 = g$, and thus G does not satisfy c_2.
(3) Since by Definition 16 f_X does not satisfy c_1 there is a total morphism $y : Y_1 \to X_2$ so that $y \circ c_1 = f_X$. If $g_2 : X_2 \to G$ then $g \circ f_X : X_1 \to G$ and $g \circ y : Y_1 \to G$ with $g \circ f_X = g \circ y \circ c_1$. Therefore, G does not satisfy c_1. □

The ACL constraint $d3$ in Fig. 10 and the LBAC constraint $c5$ in Fig. 6 are in a critical conflict.

Conflicts between constraints that render a framework contradictory can be resolved by removing or weakening one of the constraints. Weakening a constraint means to require the satisfaction of the constraint only conditionally. A condition for a constraint is a negative constraint that has to be satisfied before the constraint is checked.

Definition 19 (*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 graph morphism $k : X \to G$ satisfies a conditional constraint (x, c) **if and only if** whenever k satisfies the constraint condition x, k also satisfies c. A graph G satisfies (x, c) iff each total graph morphism $k : X \to G$ satisfies (x, c).

A conditional constraint solves the conflict of a constraint c_1 with a constraint 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 (i.e., the premise of c_2 does not occur). The constraint condition f_X has this property for c_1.

Fact 20. *Let $c_1 : X_1 \to Y_1$ be a constraint in conflict with the constraint $c_2 : X_2 \to Y_2$ via $f_X : X_1 \to X_2$ and $f_Y : Y_1 \to Y_2$, then G satisfies f_X (considered as negative constraint) if and only if G vacuously satisfies c_2.*

Proof. ($\Rightarrow$) If there exists a morphism $g : X_2 \to G$, then $g \circ f_X : X_1 \to G$ and, therefore, G would not satisfy f_X. This is a contradiction to the assumption that G satisfies f_X.

($\Leftarrow$) The graph G can satisfy f_X either vacuously or properly. If there is no morphism $p : X_1 \to G$, then G satisfies f_X vacuously. If there is a morphism $p : X_1 \to G$, then there cannot be a morphism $g : X_2 \to G$ with $g \circ f_X = p$ since G vacuously satisfies c_2. Hence G satisfies f_X. □

Definition 21 (*Conditional constraint conflict*). A conditional constraint $(x_1 : X_1 \to N, c_1 : X_1 \to Y_1)$ is in conflict with a conditional constraint $(x_2 : X_2 \to N, c_2 : X_2 \to Y_2)$ if c_1 is in conflict with c_2 (cf. Definition 16) and f_X satisfies x_1.

Definition 22 (*Weak constraint*). Let c_1 be a constraint in conflict with the constraint c_2 via f_X and f_Y. The *weak constraint* $c_1^{c_2}$ for c_1 with respect to c_2 is the conditional constraint $c_1^{c_2} = (f_X, c_1)$.

Fact 23. *If c_1 is a constraint in conflict with the constraint c_2, then the weak constraint $c_1^{c_2}$ is not in conflict with the constraint c_2.*

Proof. While there are still morphisms f_X and f_Y, the morphism f_X satisfies $c_1^{c_2}$ by construction of the weak constraint, since f_X does not satisfy the constraint condition of $c_1^{c_2}$. Therefore, $c_1^{c_2}$ and c_2 are not in conflict. □

The strategy adopted to solve conflicts (removing or weakening) depends on the particular application and on the context of the conflict. If the conflict arises from a transition between two policy frameworks, then a radical strategy giving priority to the new policy would consistently choose the constraint from the surviving policy and remove the other one. In conflicts arising from integration, another strategy may select constraints from either policy depending on the specific pair and weaken them. A general discussion of strategies is outlined in [11]. It is worth stressing that determining a conflict between constraints can be performed statically and automatically.

7.2. Rule-rule conflicts

Two rules are in a *p-conflict* (potential conflict) if they do (partly) the same things but under different conditions. A *conflict* occurs if p-conflicting rules can be applied to the same system graph. The choice for one rule in a conflict may prevent the applicability of the other rule. This kind of conflict is called *critical*, otherwise it is only a *harmless conflict*.

The example in Fig. 12 shows the LBAC rule `new object` and the ACL rule `create object`. These rules are in p-conflict, since both rules create a new object node *Ox*. The rule `new object` creates an object with a security level, the rule `create object` an object without one.

A static analysis of the rules can detect the critical and the harmless conflicts before run-time so that the rules can be changed to avoid the conflicts. The static analysis of the rules make is based on graph transformation concepts.

Definition 24 (*Conflict pair*). Two rules $p_1 : L_1 \overset{r_1}{\rightharpoonup} R_1$ with application condition $A(p_1)$ and $p_2 : L_2 \overset{r_2}{\rightharpoonup} R_2$ with application condition $A(p_2)$ are in *p-conflict* if there exists a common non-empty sub-rule for p_1 and p_2.[2]

Each pair $(m_1 : L_1 \to G, m_2 : L_2 \to G)$ of matches m_1 and m_2 for rules p_1 and p_2, respectively, 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 exists a conflict pair for p_1 and p_2. Otherwise, they are called *conflict-free*.

Notice that in Definition 24, m_i, $i = 1, 2$, is a match for p_i and therefore it satisfies the application condition $A(p_i)$. The definition of a conflict between rules considers matches for p-conflicting rules into arbitrary graphs. In general there exist infinitely many matches for one rule, so that the decision cannot be made whether two p-conflicting rules are in conflict by checking each of them. Therefore, the set of matches must be reduced for a static analysis. To detect a conflicting rule pair, it is sufficient to consider all the gluings of the left-hand sides of the rules.

Definition 25 (*Set of conflict pairs*). Given p-conflicting rules $(p_1 : L_1 \overset{r_1}{\rightharpoonup} R_1, A(p_1))$ and $(p_2 : L_2 \overset{r_2}{\rightharpoonup} R_2, A(p_2))$, the *set $CP(p_1, p_2)$ of conflict pairs* for p_1 and p_2 consists of all conflict pairs $(m_1 : L_1 \to G, m_2 : L_2 \to G)$, where m_1 and m_2 are jointly surjective.[3]

The set of conflict pairs for two rules in a rule-conflict consists of a finite number of pairs since the left-hand side of a rule is a finite graph. It is sufficient to investigate the conflict pairs into the set $CP(p_1, p_2)$ to decide the conflict-freeness of the two rules.

Theorem 26 (*Conflict-freeness*). *Let $CP(p_1, p_2)$ be the set of static conflict pairs for the p-conflicting rules $(p_1 : L_1 \overset{r_1}{\rightharpoonup} R_1, A(p_1))$ and $(p_2 : L_2 \overset{r_2}{\rightharpoonup} R_2, A(p_2))$. Then, the rules p_1 and p_2 are conflict-free if and only if $CP(p_1, p_2)$ is empty.*

Proof. We show that p_1 and p_2 are in conflict if and only if there is a conflict pair $(m_1, m_2) \in CP(p_1, p_2)$.

($\Rightarrow$) If p_1 and p_2 are in conflict, there is a conflict pair $(m_1 : L_1 \to G, m_2 : L_2 \to G)$. Let the outer diagram below be the pullback of m_1 and m_2 and diagram (1) be the pushout of in_1 and in_2.

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

[3] that is, $m_1(L_1) \cup m_2(L_2) = G$.

$$\begin{tikzcd}
& S \arrow[ld, "in_1"'] \arrow[rd, "in_2"] \arrow[d, phantom, "(1)"] & \\
L_1 \arrow[r, "m'_1"] \arrow[rd, "m_1"'] & G' & L_2 \arrow[l, "m'_2"'] \arrow[ld, "m_2"] \\
& G &
\end{tikzcd}$$

The pair (m'_1, m'_2) is a conflict pair in $CP(p_1, p_2)$ since m'_1 and m'_2 are jointly surjective by (pushout) construction. Furthermore, they satisfy $A(p_1)$ and $A(p_2)$, respectively, since otherwise, there would be total morphisms $n'_i : N_i \to G'$ with $n'_i \circ x_i = m'_i$ for $(x_i : L_i \to N_i) \in A(p_i)$ and $i = 1, 2$. Since diagram (1) is a pushout diagram, there would exist a unique $u : G' \to G$ with $u \circ m'_i = m_i$ $(i = 1, 2)$. Therefore, the morphism $u \circ n'_i$ for $i = 1, 2$ would prevent the satisfaction of $(x_i : L_i \to N_i)$ for m_i. This is a contradiction.

The direction $\Leftarrow$ follows directly from Definition 24. □

The set of conflict pairs for rules may be split into *harmless conflict pairs* and *critical conflict pairs*. The distinction is based on whether the order of rule application is critical. For a *critical conflict pair* (m_1, m_2) the order is important: after applying p_1 at match m_1, the rule p_2 is no longer applicable or vice versa. For a *harmless conflict pair* (m_1, m_2) the order does not matter: after applying p_1 at match m_1, the rule p_2 is still applicable and vice versa. Critical and harmless conflict pairs are defined and detected by the graph transformation concept of *parallel independence* [4].

Definition 27 (*Parallel independence*). Given rules $(p_1 : L_1 \overset{r_1}{\rightharpoonup} R_1, A(p_1))$ and $(p_2 : L_2 \overset{r_2}{\rightharpoonup} R_2, A(p_2))$, 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*.

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

Two derivations are parallel independent if the first rule does not delete anything needed by the second rule and it does not create anything that the NAC of the second rule forbids. The same conditions must be satisfied for the second rule with respect to the first rule. In the case of parallel independence, the application of rule p_1 at match m_1 and the subsequent application of rule p_2 at $r_1^* \circ m_2$ results in the same graph (up to isomorphism) as the application of rule p_2 at match m_2 and the subsequent application of rule p_1 at $r_2^* \circ m_1$. For a proof of this result, see [4].

Definition 28 (*Harmless/critical conflict pair*). A conflict pair (m_1, m_2) for rules p_1 and p_2 is a *harmless 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*. Otherwise, the conflict pair is a *critical conflict*.

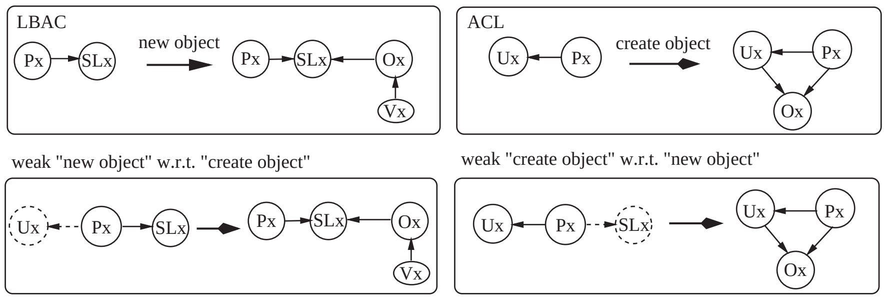
LBAC
new object
Px
SLx
Ox
Vx
ACL
create object
Ux
weak "new object" w.r.t. "create object"
weak "create object" w.r.t. "new object"

Fig. 13. The weak rule for `new object` and `create object`.

We propose two possible strategies to solve critical conflicts between p-conflicting rules. In the first strategy, one rule is given priority over the other one. One rule is chosen as *major rule* and denoted by p_1, and one as *minor rule* and denoted by p_2. For a conflict pair (m_1, m_2), the rule 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 two rules into one rule. In [11] we discuss alternative strategies.

Definition 29 (*Weak rule*). Given a conflict pair (m_1, m_2) for rules $(p_1 : L_1 \xrightarrow{r_1} R_1, A(p_1))$ and $(p_2 : L_2 \xrightarrow{r_2} R_2, A(p_2))$, the *weak rule* for p_2 w.r.t. (m_1, m_2), denoted by $WR(p_1, p_2, (m_1, m_2))$, is the rule p_2 with the NAC (L_2, N), where the outer diagram below is a pullback and diagram (1) (the top half) is a pushout diagram.

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

We call the NAC (L_2, N) the *weak condition* and denote it by $WC(p_1, p_2, (m_1, m_2))$

The addition of the negative application condition $WC(p_1, p_2, (m_1, m_2))$ to the minor rule ensures that the major and the minor rules cannot be applied both to the same system graph with matches m_1 and m_2, respectively.

Example 30 (*Weak rule*). Let us show an example of the *weak rule*, the rule extended by the weak condition $WC(p_1, p_2, (m_1, m_2))$ in Definition 29. Fig. 13 shows the example of the p-conflicting ACL rule `create object` and the LBAC rule `new object`. The set of conflict pairs for these two rules has two elements: 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 the disjoint union of left-hand sides, and the inclusions $(in'_1 : L_1 \to G, in'_2 : L_2 \to G)$ of the

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

Fig. 14. Amalgamation of rule-conflicting rules.

left-hand sides into the graph *G*, which is the gluing of the left-hand sides over the node *Px*. Fig. 13 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 a security level for the process. Therefore, the weak rule for `create object` is only applicable to processes belonging to the ACL model and without a counterpart in the LBAC model. The weak rule for `new object` with respect to `create object` has a NAC that forbids a user connected to the process. Since each user is connected to a process, the rule is never applicable.

Proposition 31 (*Major rule and extended minor rule are conflict-free*). *Given the set of conflict pairs $CP(p_1, p_2)$ for p-conflicting rules p_1 and p_2, the rule p_1 and the weak rule p_2' extended by a $WC(p_1, p_2, (m_1, m_2))$ for each $(m_1, m_2) \in CP(p_1, p_2)$ are conflict-free.*

Proof. Let $m_2 : L_2 \to G$ be a match for p_2 extended by $WC(p_1, p_2, (m_1, m_2))$ (see Definition 29), i.e. m_2 satisfies $WC(p_1, p_2, (m_1, m_2))$. If we assume that $m_1 : L_1 \to G$ is a match for p_1, then there exists, by construction of $WC(p_1, p_2, (m_1, m_2))$, a unique morphism $u : N \to G$ so that $u \circ n = m_2$ (the outer diagram commutes by the pullback property). This is a contradiction, since m_2 satisfies $WC(p_1, p_2, (m_1, m_2))$. □

The second solution for solving conflicts between rules is the *amalgamation* of the conflicting rules over their common subrule. The amalgamated rule for two rules over a common subrule has as left-hand side the colimit of the subrule-morphisms for the left-hand sides, as right-hand side the colimit of the subrule-morphisms for the right-hand sides and the rule morphism is given by the universal colimit property [1]. The NACs of the rules are integrated over the common objects specified in the left-hand sides.

Definition 32 (*Amalgamated rule*). Let $(p_i : L_i \overset{r_i}{\rightharpoonup} R_i, A(p_i))$ for $i = 1, 2$ be p-conflicting rules and $p_0 : L_0 \overset{r_0}{\rightharpoonup} R_0$ with $f_{L_i} : L_0 \to L_i$ and $f_{R_i} : R_0 \to R_i$ their common subrule (cf. Fig. 14).

The *amalgamated rule* of p_1 and p_2 with respect to p_0 is given by $(p : L \overset{r}{\rightharpoonup} R, A(p))$, 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 induced universal pushout morphism.

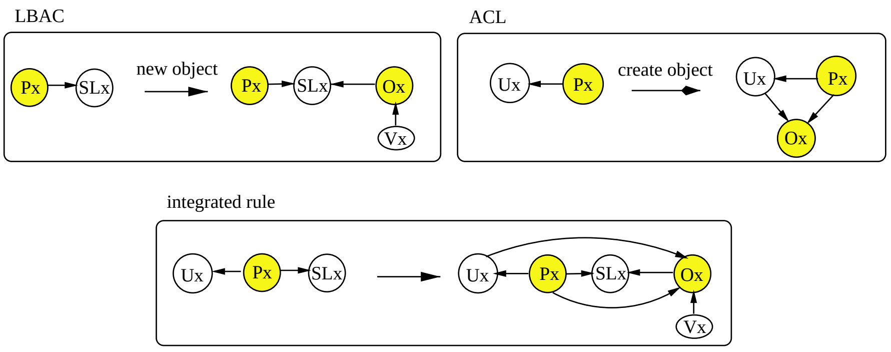

Fig. 15. Amalgamation of rule-conflicting rules `create object` and `new object`.

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 induced universal pushout morphism.

Example 33. Fig. 15 shows the amalgamated rule for the rules `create object` and `new object`. Their common subrule is shaded 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 amalgamated rule creates an object that belongs to a user as well as a process and carries a security level.

The amalgamated rule answers the question “which rule shall be applied in a conflict” by “both” instead of favouring one. As in the case of constraints, the actual conflict resolution strategy depends on the application and the context of the conflict. Also rule conflicts can be determined statically and automatically.

8. Rule-constraint conflict

In this section, in order to address conflicts between constraints and rules, we classify rules in *deleting* and *expanding* rules. Deleting rules delete graph elements, but do not add anything (i.e., $range(r) = R \subset L$); expanding rules may 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 never violates a negative constraint, since the rule does not add forbidden graph elements. But a deleting rule may violate a positive constraint if the rule deletes conditionally required graph elements but preserves the condition for their existence. An expanding rule may violate both negative and positive constraints: it may add forbidden graph elements specified in a negative constraint or it may complete the condition for a positive constraint without making sure that the required graph elements exist under this condition as well.

Definition 34 (*Rule-constraint conflicts*). Let $p : L \overset{r}{\rightharpoonup} R$ be an expanding rule and $c : X \rightarrow Y$ a constraint, then p and c are in *conflict* if there exists a non-empty graph S and injective total morphisms $s_1 : S \rightarrow R$ and $s_2 : S \rightarrow X$ so that $s_1(S) \cap (R \setminus r(L)) \neq \emptyset$.

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

Conflicts between rules p and constraints $c : X \rightarrow Y$ can be resolved by adding NACs to the rules p. We present next the construction of these negative application conditions and then show how it is used to resolve conflicts.

Definition 35 (*Reduction*). Given a rule $p : L \overset{r}{\rightharpoonup} R$ and a non-empty overlap S between R and the conclusion Y of the constraint $c : X \rightarrow Y$ as in the following diagram:

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

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

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 described in [8] if the constraint $c : X \rightarrow Y$ is the identity morphism.

The construction in Definition 35 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 where the overlap $S \rightarrow R$ can be decomposed into $S \rightarrow L \rightarrow R$. The graph N generated from such an overlap can be eliminated directly from Definition 35 by requiring only overlaps S for which $s_1(S) \cap (R \setminus r(L)) \neq \emptyset$. In this manner, the application condition $NAC1$ of Fig. 18 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 $L \rightarrow G$ satisfies (L, N_2), then it also satisfies (L, N_1) and the application condition (L, N_1) can be removed from $A(p, c)$. Consider for example Fig. 17, where the overlap S_1 is included into the overlap S_3. Therefore, $NAC3 \subseteq NAC1$ (cf. Fig. 18) and we can remove $NAC1$.

In the next subsections, the different combinations of expanding/deleting rules and positive/negative constraints are analyzed and the appropriate preservation results presented.

8.1. Conflicts between constraints and deleting rules

There can be no conflict between a deleting rule $p : L \overset{r}{\rightharpoonup} R$ and a negative constraint $c : X \to Y$, since the deleting rule may remove parts of Y, in which case c is trivially satisfied, or parts of X, in which case c is vacuously satisfied.

Theorem 36 (*Deleting preserves satisfaction*). *Let $p : L \overset{r}{\rightharpoonup} R$ be a deleting rule, G a graph that satisfies the negative constraint $c : X \to Y$ and $G \overset{p}{\Rightarrow} H$, then H satisfies c*

Proof. By definition, the following is a pushout diagram:

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

Since p is deleting, there exist total morphisms $f : R \to L$ and $g : H \to G$ such that $m \circ f = g \circ m^*$. Let $k : X \to H$. If H did not satisfy c, then there would exist a morphism $q : Y \to H$ such that $X \xrightarrow{c} Y \xrightarrow{q} H = X \xrightarrow{k} H$. But then $X \xrightarrow{c} Y \xrightarrow{q} H \xrightarrow{g} G = X \xrightarrow{k} H \xrightarrow{g} G$, contradicting the assumption that G satisfies c. □

For the conflict between deleting rules and positive constraints, it is possible to add NACs that prevent the rule from destroying the conclusion Y, by preventing the applicability in the presence of X if part of the conclusion Y is intended to be deleted by the rule.

Theorem 37 (*Satisfaction by reduction*). *Let $p : L \overset{r}{\rightharpoonup} R$ be a deleting rule and G a graph that satisfies the positive constraint $c : X \to Y$. Furthermore, let $p(id_Y) = (id_L, A(id_L, c))$ be the reduction of $id_L : L \to L$ by $id_Y : Y \to Y$, and define $p(c) = (r, A(id_L, c))$. If $G \overset{p(c)}{\Rightarrow} H$ is a derivation with $p(c)$, then H satisfies c.*

Proof. Let $G \overset{p(c)}{\Rightarrow} H$ via the matching morphism $m : L \to G$ and $k : X \to H$ a morphism.

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

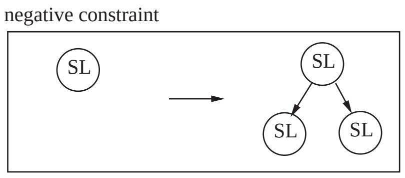

Fig. 16. Negative constraint for the LBAC model which requires at most one successor security lattice level.

Since p is deleting and r is injective, $k' = r^{*^{-1}} \circ k : X \to H \to G$ is well-defined and total. By the assumption that G satisfies $c : X \to Y$, there exists a total morphism $q : Y \to G$ such that $q \circ c = k'$. Now $(r^* \circ q) \circ c = r^* \circ (q \circ c) = r^* \circ (r^{*^{-1}} \circ k) = k$, so that the proof is complete if we can show that $r^* \circ q : Y \to H$ is total. Since q is total, it is sufficient to show that r^* is defined for $q(y)$ for every $y \in Y$. Suppose not. Then, for some $y \in Y$ and $l \in L \setminus dom(r)$, $q(y) = m(l)$ defining hence an overlap of L and Y and thus contradicting the assumption that m satisfies the NAC $A(id_L, c)$ of $p(c)$. □

8.2. Conflicts between negative constraints and expanding rules

For the conflict between expanding rules and negative constraints, the NACs prevent the rule from completing the conclusion Y of the negative constraint $c : X \to Y$. The following result confirms that the construction is the appropriate one.

Theorem 38 (*Reduction preserves satisfaction*). *Let $p : L \overset{r}{\rightharpoonup} R$ be an expanding rule and G a graph that satisfies the negative constraint $c : X \to Y$. If $p(c)$ is the reduction of p by c and $G \overset{p(c)}{\Rightarrow} H$ is a derivation with $p(c)$, then H satisfies c.*

Proof. Suppose, looking for a contradiction, that there exists a morphism $f : Y \to H$. Since $H = R +_L G$, there exist partial morphisms $f_R : Y \to R$ and $f_G : Y \to G$ such that $f_R \cup f_G = f$. Since G satisfies c, f_R cannot be empty. Hence there exist an overlap S of R and Y which generates one of the NACs in $A(p, c)$ in Definition 35. This contradicts the applicability of $p(c)$ to G necessary to produce H. □

Example 39 (*Negative constraints and expanding rules conflict*). We give an example for the LBAC model to which we add the negative constraint in Fig. 16, denoted by $c(succ)$ in the sequel, which forbids two (or more) successors for a security level. The (expanding) rule `new level 2` in Fig. 5 may produce an inconsistent state by adding a successor level to a security level which already has a successor.

Fig. 17 shows non-empty overlaps $S1$, $S2$ and $S3$ of the right-hand side of the rule `new level 2` and the constraint $c(succ)$. The remaining overlaps use the same subgraphs $S1$, $S2$ and $S3$, but different morphisms s_1 and s_2. For each overlap S, the pushout $(C, R \to C, X \to C)$ of the morphisms $S \to R$ and $S \to X \to Y$ is constructed (see Fig. 17 for the example overlaps). The application condition (L, N) is constructed by applying the inverse rule[4] of `new level 2` to the graphs Ci resulting in the graphs

[4] The inverse rule of a rule $p : L \overset{r}{\rightharpoonup} R$ with an application condition is the rule $p^{-1} : R \overset{r^{-1}}{\rightharpoonup} L$ without the application condition.

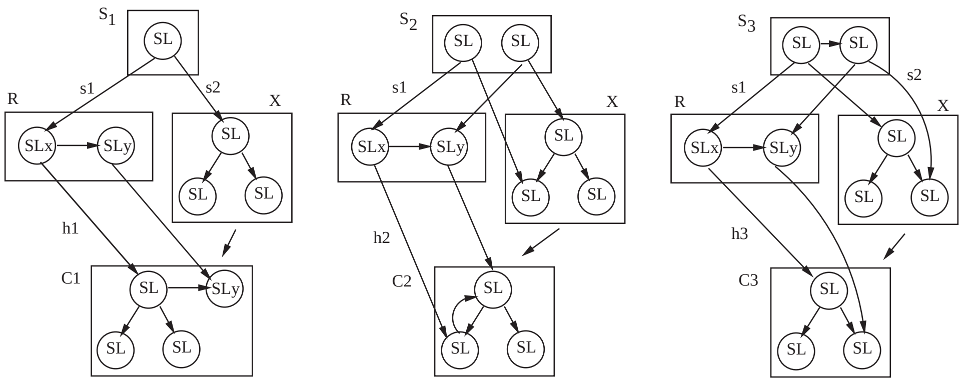

Fig. 17. Non-empty overlaps between the right-hand side R of `new level 2` and the conclusion Y of the negative constraint $c(succ)$.

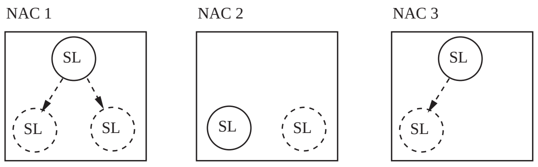

Fig. 18. NACs constructed from the overlaps.

Ni. The inverse rule of `new level 2` deletes a security level. The generated application condition for the gluing C is then the pair (L, N). Fig. 18 shows the pairs (L, N) for the three overlaps in Fig. 17. We depict only the graph N in which the graph L is drawn by solid lines, and the part $N \setminus L$ by dashed lines.

8.3. Conflicts between positive constraints and expanding rules

The following result shows that the construction in Definition 35 is sufficient to guarantee the preservation of satisfaction of constraints in the case of an expanding rule and a positive constraint too.

Theorem 40 (*Reduction preserves satisfaction*). *Let $p : L \xrightarrow{r} R$ be an expanding rule and G a graph that satisfies the positive constraint $c : X \to Y$. If $p(id_X)$ is the reduction of p by $id_X : X \to X$, and $G \overset{p(id_X)}{\Rightarrow} H$ is a derivation with $p(id_X)$, then H satisfies c.*

Proof. The proof is this result is straightforward. By construction, the reduction of p by id_X prevents the application of p from constructing additional occurrences of X in H. Therefore, either H vacuously satisfies c (no occurrences of X in H) or occurrences of X in H are inherited from G. Since G satisfies c, the occurrence of X can be extended to Y and, since p is expanding, it remains in H. □

For the conflict between expanding rules and positive constraints, the NACs prevent the rule from completing the condition X.

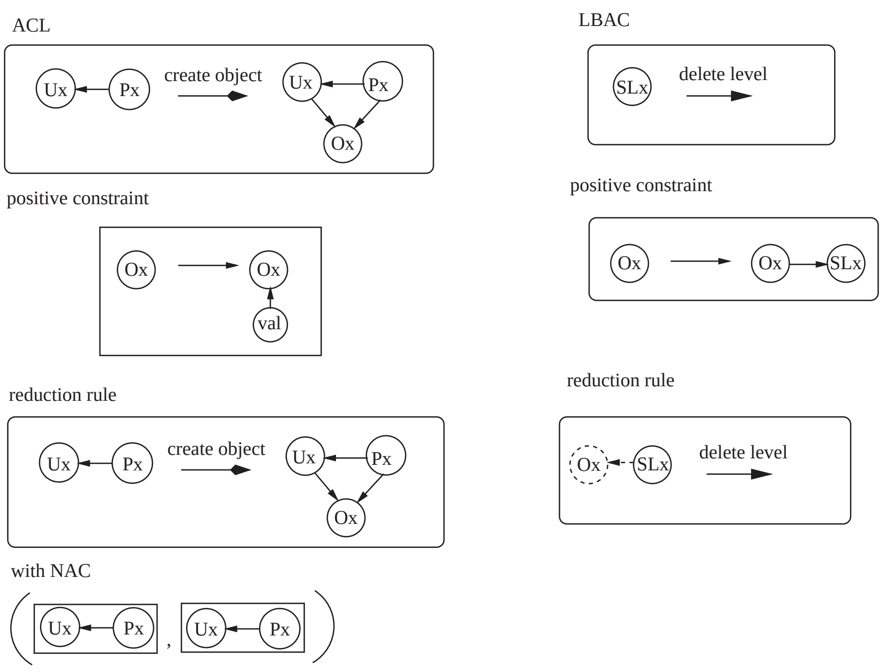

Fig. 19. The negative application condition is too strong.

Example 41 (*Positive constraints and expanding rules conflict*). Fig. 19 on the left-hand side describes an example for the reduction of the expanding rule `create object` and a positive constraint which requires a value for each object. The reduction adds a NAC (L, L) which has been drawn explicitly. The right-hand side of the figure shows the deleting rule `delete level` and a positive constraint which requires a security level for each object. The NAC for the rule `delete level` is the NAC of the reduction of the identity rule on the node *SLx* by the identity morphism on the conclusion of the constraint. This NAC forbids the deletion of security levels that are connected to an object. This NAC is already described in Fig. 5 and shows the correctness of our LBAC graph model with respect to the rule `delete level` and the constraint for the existence of a security level for each object.

The solution for conflicts between expanding rules and negative constraints as well as for 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 the number of system graphs that can be generated more than necessary. For example, the reduction of the rule `create object` by the positive constraint in Fig. 19 preserves consistency, but it cannot be applied since the NAC is never satisfied. This example of the required object value suggests for positive constraints and expanding rules 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, when the rule constructs part of X. In the object value example, the right-hand side of the rule `create object` must not only create the object node *Ox*, but *Ox* together with a value node. The construction in Definition 42 describes the details of a possible solution.

Definition 42 (*Completing rule*). Let $c : X \to Y$ be a positive constraint and $p : L \overset{r}{\to} R$ an expanding rule. The *completing rule* $p^*(c)$ for p and c is given by $v_i \circ h_i \circ r : L \to R'$, where

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

- $\Omega = \{R \overset{s_1^i}{\leftarrow} S_i \overset{s_2^i}{\rightarrow} X, I = 1, \ldots n\}$ is the set of all non-empty overlaps of R and X so that $s_1^i(S_i) \cap (R \setminus r(L)) \neq \emptyset$,
- for each $R \overset{s_1^i}{\leftarrow} S_i \overset{s_2^i}{\rightarrow} X \in \Omega$, (C_i, h_i, y_i) is the pushout of the injective morphisms s_1^i and $c \circ s_2^i$ in **Graph**,
- $(R', v_i : C_i \to R')$ is the colimit of the morphisms $h_i : R \to C_i$ in **Graph**.

Notice that, by definition of colimit, $v_i \circ h_i = v_j \circ h_j$, for all i, j, and therefore the choice of the index in defining $p^*(c) = v_i \circ h_i \circ r : L \to R'$ is immaterial.

Fig. 20 illustrates the construction of the completing rule for the rule `create object` and the positive constraint in the left-hand side of Fig. 19. The figure shows the only non-empty overlap of R and X which contains created elements. The construction extends the right-hand side of the rule `create object` so that the value node for the new object is also created.

The completing rule, however, does not preserve consistency for each positive constraint. The counterexample in Fig. 21 shows the completing rule for a rule which creates a node A and a positive constraint c which requires all nodes A and B to be connected via a C node. The completing rule $p^c(c)$ is applied to a consistent graph G on the right-hand side of Fig. 21, but the resulting graph H is not consistent.

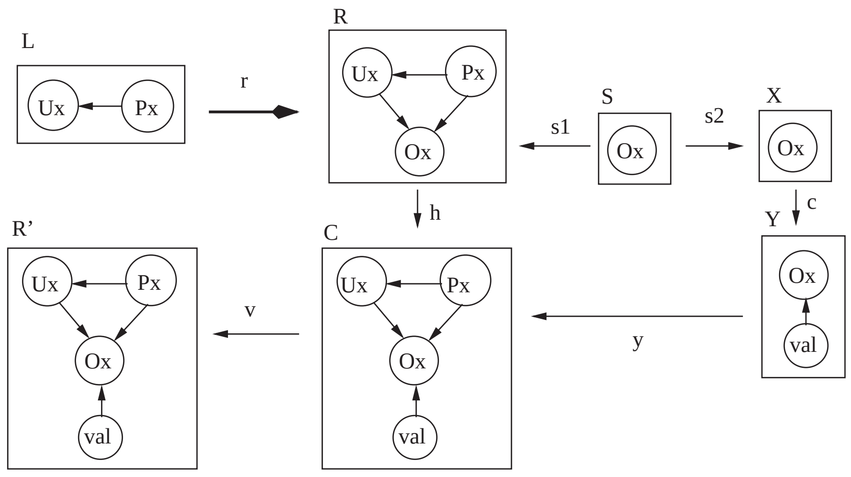

Fig. 20. Construction of the completing rule.

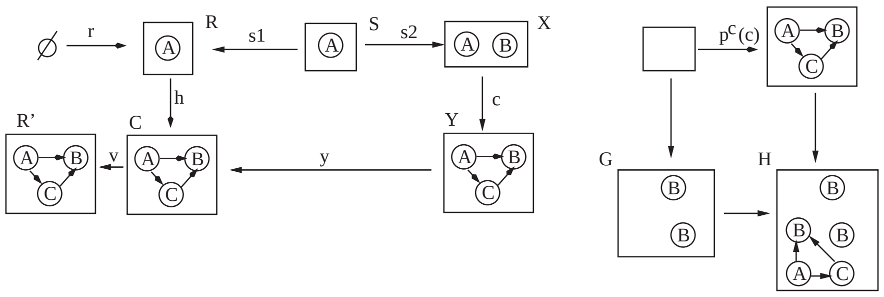

Fig. 21. Counterexample construction.

If we restrict positive constraints to *single* positive constraints, the construction always produces a consistency—preserving rule. A positive constraint $c : X \to Y$ is a single positive constraint if X contains at most one node.

Proposition 43. *If c is a single positive constraint, the completing rule $p^*(c)$ for a rule p is consistent with respect to c.*

Proof. Let G be a graph which is consistent with respect to $c : X \to G$, and $G \overset{p^*(c)}{\Rightarrow} H$ a derivation with $p^*(c)$ and co-match $m^* : R' \to H$. If there is no morphism $g : X \to H$, then H vacuously satisfies c.

If c is a single positive constraint and there is a graph morphism $g : X \to H$, then $g(X) \subseteq H \setminus m^*(R')$ or $g(X) \subseteq m^*(R')$. If $g(X) \subseteq H \setminus m^*(R')$ then $g(X)$ contains only elements already occurring in G. Since G satisfies c, there is a morphism $q : Y \to H$ with $g = q \circ c$. If $g(X) \subseteq m^*(R')$ then X is one of the overlaps S_i between X and R in the construction of Definition 42. If S_1 is the overlap, then, since C_1 is the pushout object, there is a morphism $y_1 : Y \to C_1$ so that $m^* \circ v_1 \circ y_1$ is the needed extension of g. □

Another possibility to resolve conflicts between positive constraints and expanding rules p is to transform the positive constraint into a rule and then require that this rule be applied (after the application of p) as long as there are occurrences of X not “visited” in H. The new rule is just the total morphism $X \to Y$ with negative application condition (X, Y) to avoid its application repeatedly on the same part of H.

Definition 44 (*Constraint-repair rule*). For a positive constraint $c : X \to Y$, the *constraint-repair rule* for c is given by $rep(c) = c : X \to Y$ with NAC (X, Y).

The proof of the following result is straightforward.

Proposition 45 (*Satisfaction after repair*). *Let $p : L \xrightarrow{r} R$ be an expanding rule and G a graph that satisfies the positive constraint $c : X \to Y$. If $G \Rightarrow H$ is a derivation consisting of applying the expanding rule p once and then H' is obtained by applying to H the constraint-repair rule $rep(c)$ as long as possible, then H' satisfies c.*

It is necessary 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 [7] or the notion of a transformation unit [14] as an encapsulation mechanism used in a way similar to procedure calls.

9. Concluding remarks

In the formalism presented here 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 framework offers the conceptual tools needed to discuss the effect of integrating two policies using a pushout in the category of policy frameworks and framework morphisms.

An important problem addressed here is how to deal with inconsistencies caused by conflicts between two constraints, between two 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 [10]. The conflict between a rule of one policy and a simple constraint of the other policy has already been addressed in part elsewhere [12,13], where it is also shown the adequacy of this framework to represent different Access Control policies. 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.

We have shown how conflicts can be detected in a security framework and proposed several strategies to automatically resolve the conflicts. Since there are several possible resolution strategies, the administrator has a choice. These resolution strategies are *meta policies*, that deal with the choices in the application of policies. We have introduced the notion of a metapolicy in [13] and proposed three examples:

- The meta policy *radical* chooses a major policy SP_1 and a minor policy SP_2 and solves the problem of conflicting rules or constraints globally by selecting the rules or constraints of SP_1 and deleting the rules and constraints of SP_2.
- The meta policy *weakRadical* keeps the conflicting rules or constraints of SP_2, but weakens them to favour the application or the satisfaction, respectively, of the corresponding rules and constraints of SP_1.
- The meta policy *static* keeps some rules and constraint from SP_1 and some from SP_2, weakening the corresponding conflicting rules and constraints of the other policy. The choice of which one to weaken is made on a pair-by-pair basis.

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.

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.

Appendix A. Category of security policy frameworks

We provide in this appendix the categorical formalization of our model by defining the category of security policy frameworks and framework morphisms.

Definition A.1 (*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.

Proof. For each framework *SP*, the pair $id_{SP} = (id_{TG}, id_P)$ is the identity morphism and the composition of framework morphisms $f = (f_{TG}, f_P) : SP_1 \to SP_2$ and $g = (g_{TG}, g_P) : SP_2 \to SP_3$ is defined componentwise as the morphism $g \circ f = (g_{TG} \circ f_{TG}, g_P \circ f_P) : SP_1 \to SP_3$. The composition is associative since the composition for the component morphisms is associative. □

Since a framework morphism does not constrain the sets of constraints, frameworks with an isomorphic type graph and isomorphic rules are isomorphic, independently of their constraint sets.

Proposition A.2 (*Framework isomorphisms*). *The framework morphism* $f = (f_{TG}, f_P) : SP \to SP'$ *with* $SP = (TG, (P, r_P), Pos, Neg)$ *and* $SP' = (TG', (P', r'_P), Pos', Neg')$ *is an isomorphism if and only if* $f_{TG} : TG \to TG'$ *and* $f_P : P \to P'$ *are isomorphisms in the appropriate categories.*

Proof. ⇒: If *f* is an isomorphism, there is a framework morphism $g : SP' \to SP$ so that $f \circ g = id_{SP'}$ and $g \circ f = id_{SP}$. Then, $f_{TG} \circ g_{TG} = id_{SP'_{TG}}$ and $g_{TG} \circ f_{TG} = id_{SP_{TG}}$, i.e., f_{TG} is an isomorphism. Furthermore, $f_P \circ g_{P'} = id_{SP'_P}$ and $g_{P'} \circ f_P = id_{SP_{P'}}$, i.e. f_P is an isomorphism.

⇐: If the component morphisms f_{TG} and f_P are isomorphisms, then by definition of composition so is *f*. □

Proposition A.3 (*Initial security policy framework*). *The initial object in* **SP** *is given by the security policy framework* $SP_I = (\emptyset, (\emptyset, r), \emptyset, \emptyset)$.

Proof. Given $SP = (TG, (P, r_P), Pos, Neg)$, we define $i = (i_{TG}, i_P) : SP_I \to SP$, where $i_{TG} : \emptyset \to TG$ is the unique morphism for the initial graph ∅ and *TG* and $i_P : \emptyset \to P$ is the unique morphism for the set ∅ and *P*. The properties of the initial component objects in the respective categories ensure the required property for SP_I. □

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

Theorem A.4 (*Pushouts*). *The category* **SP** *has all pushouts.*

Proof. Let Op : **Set** × **Set** → **Set** be an operation on sets. For given framework morphisms $f : SP_0 \to SP_1$ and $g : SP_0 \to SP_1$ the pushout $SP = (TG, (P, r_P), Pos, Neg)$ is constructed as follows:

(1) Construct the pushout of f_{TG} and g_{TG} in the category **Graph** (diagram (1)).

(2) Construct the pushout of f_P and g_P in **Set** (diagram (2)).

(3) The mapping $r_P(x)$ is defined for all $x \in P$ as follows:

 (a) if there is a $y \in P_1$ with $g'_P(P_1)(y) = x$ and $x \notin f'_P(P_2)$ then $r_P(x) = F_{g'_{TG}}(r_{P_1}(y))$,

 (b) if there is a $y \in P_2$ with $f'_P(P_2)(y) = x$ and $x \notin g'_P(P_1)$ then $r_P(x) = F_{f'_{TG}}(r_{P_2}(y))$,

 (c) if $x \in g'_P(P_1) \cap f'_P(P_2)$, then there is a non-empty set $X \subset P_0$ so that for each $x_0 \in X$, $g'_P(f_P(x_0)) = f'_P(g_P(x_0)) = x$ by construction of pushouts in **Set**. We explain the construction of $r_P(x)$ for the case of a singleton set $X = \{x_0\}$. This construction can then be generalized to an arbitrary set X. By definition of framework morphisms, $r_{P_0}(x_0)$ is a subrule of $r_{P_1}(f_P(x_0)) := y$ and $r_{P_2}(g_P(x_0)) := y'$. The two squares $L_0 \to L_1 \to L \leftarrow L_2$ and $R_0 \to R_1 \to R \leftarrow R_2$ in the diagram below on the right are pushouts in **Graph** and $r_P(x)$ is given by the unique pushout property. [5]

$$\begin{tikzcd}
TG_0 \arrow[r, "f_{TG}"] \arrow[d, "g_{TG}"'] \arrow[dr, phantom, "(1)"] & TG_1 \arrow[d, "g'_{TG}"] \\
TG_2 \arrow[r, "f'_{TG}"'] & TG
\end{tikzcd}
\qquad
\begin{tikzcd}
P_0 \arrow[r, "f_P"] \arrow[d, "g_P"'] \arrow[dr, phantom, "(2)"] & P_1 \arrow[d, "g'_P"] \\
P_2 \arrow[r, "f'_P"'] & P
\end{tikzcd}
\qquad
\begin{tikzcd}
 & L_0 \arrow[r, "r_{P_0}(x_0)"] \arrow[dl] \arrow[dr] & R_0 \arrow[dl] \arrow[dr] & \\
L_1 \arrow[r, "y"] \arrow[dr] & R_1 \arrow[dr] & L_2 \arrow[r, "y'"] \arrow[dl] & R_2 \arrow[dl] \\
 & L \arrow[r, "r_P(x)"'] & R &
\end{tikzcd}$$

(4) The set of positive constraints is defined as $PosOp(F_{g'_{TG}}(Pos_1), F_{f'_{TG}}(Pos_2))$, the set of negative constraints as $Neg = Op(F_{g'_{TG}}(Neg_1), F_{f'_{TG}}(Neg_2))$

The pushout morphisms are defined as $g' = (g'_{TG}, g'_P) : SP_1 \to SP$ and $f' = (f'_{TG}, f'_P) : SP_2 \to SP$.

To check, that *SP* is a framework, the well-definedness of r_P has to be shown. For cases 3(a) and 3(b) the well-definedness follows from the fact that $g'_P|_{P_1 \setminus f_P(P_0)}$ and $f'_P|_{P_2 \setminus g_P(P_0)}$ are injective. For case 3(c), the rules $r_{P_0}(x_0)$ for each $x_0 \in X$ are equal due to the definition of a framework morphism. This ensures a well-defined construction of $r_P(x)$. By construction, the morphisms f', g' are framework morphisms such that $g' \circ f = f' \circ g$. Let $SP' = (TG', (P' : r_{P'}), Pos', Neg')$ be a security policy framework with framework morphisms $a : SP_1 \to SP'$ and $b : SP_2 \to SP'$ so that $a \circ f = b \circ g$. By construction, there is a unique total graph morphism $u_{TG} : TG \to TG'$ with $u_{TG} \circ g'_{TG} = a_{TG}$ and $u_{TG} \circ f'_{TG} = b_{TG}$. Moreover, there is a unique mapping $u_P : P \to P'$ with $u_P \circ g'_P = a_P$ and $u_P \circ f'_P = b_P$. The definition $u = (u_{TG}, u_P) : SP \to SP'$ yields a framework morphism by definition of r_P. The commutativity property and the uniqueness of this morphism follow from the corresponding properties of the component morphisms.

The pushout construction defines the constraint sets in the pushout framework as the result of an operation *Op* on sets. This operation *Op* may be the union or the intersection, etc. Proposition A.2 shows that the choice of the operation *Op* does not influence the pushout property. The actual operation, however, becomes important when we consider the *coherence* of a security framework in Section 6.

By combining the previous two results, we obtain the last result of this paper. □

---

[5] This construction is known as *amalgamation of rules* [16].

Theorem A.5 (*Colimits*). *The category* **SP** *is finitely complete*.

References

[1] P. Böhm, H.-R. Fonio, A. Habel, Amalgamation of graph transformations: a synchronisation mechanism, J. Comput. System Sci. 34 (1987) 377–408.

[2] D. Bell, L. LaPadula, Secure computer systems: Mathematical foundations and model, Technical Report 2 (2547), MITRE, 1973.

[3] A. Corradini, H. Ehrig, M. Löwe, J. Padberg, The category of typed graph grammars and their adjunction with categories of derivations, in: Fifth International Workshop on Graph Grammars and their Application to Computer Science, Lecture Notes in Computer Science, vol. 1073, Springer, Berlin, 1996, pp. 56–74.

[4] H. Ehrig, R. Heckel, M. Korff, M. Löwe, L. Ribeiro, A. Wagner, A. Corradini, Algebraic approaches to graph transformation part II: single pushout approach and comparison with double pushout approach, in: G. Rozenberg (Ed.), Handbook of Graph Grammars and Computing by Graph Transformations, vol. I: Foundations, World Scientific, Singapore, 1997.

[6] M. Große-Rhode, F. Parisi-Presicce, M. Simeoni, Formal software specification with refinement and modules for typed graph transformation systems, J. Comput. System Sci. 64 (2) (2002) 171–218.

[7] M. Große-Rhode, F. Parisi-Presicce, M. Simeoni, Refinements of graph transformation systems via rule expressions, in: H. Ehrig, G. Engels, H.-J. Kreowski, G. Rozenberg (Eds.), Proceedings of TAGT'98, Lecture Notes in Computer Science, vol. 1764, Springer, Berlin, 2000, pp. 368–382.

[8] R. Heckel, A. Wagner, Ensuring consistency of conditional graph grammars - a constructive approach, in: Proceedings SEGRAGRA'95 Graph Rewriting and Computation, Electronic Notes of TCS, vol. 2, 1995.

[9] M. Koch, L.V. Mancini, F. Parisi-Presicce, A formal model for role-based access control using graph transformation, in: F. Cuppens, Y. Deswarte, D. Gollmann, M. Waidner (Eds.), Proceedings of the Sixth European Symposium on Research in Computer Security (ESORICS 2000), Lecture Notes in Computer Science, vol. 1895, Springer, Berlin, 2000, pp. 122–139.

[10] M. Koch, L.V. Mancini, F. Parisi-Presicce, On the specification and evolution of access control policies, in: S.L. Osborn (Ed.), Proceedings of the Sixth ACM Symposium on Access Control Models and Technologies, ACM, May 2001, pp. 121–130.

[11] M. Koch, L.V. Mancini, F. Parisi-Presicce, Foundations for a graph-based approach to the specification of access control policies, in: F. Honsell, M. Miculan (Eds.), Proceedings of Foundations of Software Science and Computation Structures (FoSSaCS 2001), Lecture Notes in Computer Science, vol. 2030, Springer, Berlin, 2001, pp. 287–302.

[12] M. Koch, L.V. Mancini, F. Parisi-Presicce, A graph based formalism for RBAC, ACM Trans. Inform. System Security 5 (3) (2002) 332–365.

[13] M. Koch, L.V. Mancini, F. Parisi-Presicce, Conflict detection and resolution in access control specifications, in: M. Nielsen, U. Engberg (Eds.), Proceedings of Foundations of Software Science and Computation Structures (FoSSaCS 2002), Lecture Notes in Computer Science, Springer, Berlin, 2002, pp. 223–237.

[14] H.-J. Kreowski, S. Kuske, Graph transformation units and modules, in: H. Ehrig, G. Engels, H.-J. Kreowski, G. Rozenberg (Eds.), Handbook of Graph Grammars and Computing by Graph Transformations, vol. II, World Scientific, Singapore, 1999, pp. 607–638 (Chapter 15).

[15] F. Parisi-Presicce, H. Ehrig, U. Montanari, Graph Rewriting with unification and composition, in: H. Ehrig, M. Nagl, G. Rozenberg, A. Rosenfeld (Eds.), International Workshop on Graph Grammars and their Application to Computer Science, Lecture Notes in Computer Science, vol. 291, Springer, Berlin, 1987, pp. 496–524.

[16] G. Rozenberg (Ed.), Handbook of Graph Grammars and Computing by Graph Transformations, vol. I: Foundations, World Scientific, Singpore, 1997.

[17] R.S. Sandhu, Lattice-based access control models, IEEE Comput. 26 (11) (1993) 9–19.

[18] R.S. Sandhu, Role-based access control, in: M.V. Zelkowitz (Ed.), Advances in Computers, vol. 46, Academic Press, New York, 1998.

[19] R.S. Sandhu, P. Samarati, Access control: principles and practice, IEEE Commun. Mag. (1994) 40–48.