Foundations for Software Configuration Management Policies using Graph Transformations[⋆]

Francesco Parisi-Presicce[1] and Alexander L. Wolf[2]

[1] Dip. Scienze dell'Informazione, Universitá degli Studi di Roma *La Sapienza*
Via Salaria 113, 00198 Roma, Italy,
parisi@dsi.uniroma1.it

[2] Department of Computer Science, University of Colorado at Boulder
Boulder, Colorado, USA,
alw@cs.colorado.edu

Abstract. Existing software configuration management systems embody a wide variety of policies for how artifacts can evolve. New policies continue to be introduced. Without a clean separation of configuration management policies from configuration management mechanisms, it is difficult to understand the policies as well as difficult to reason about how they relate. We introduce a formal foundation for specifying configuration management policies by viewing the policies in terms of graph transformation systems. Not only are we able to precisely capture the semantics of individual policies, we can, for the first time, describe formal properties of the relationship between policies.

1 Introduction

Managing the evolution of interrelated software artifacts is a central activity in software engineering. This activity is often referred to as *version control* or, more generally, as *configuration management* (CM) [14]. Among the many relationships that exist among software artifacts, three are the principal concern of CM.

- *Revision:* a relationship reflecting the history of modifications made to an artifact over time. A revision of an artifact is considered to be a *replacement* for a previous revision of that artifact.

⋆ This work was performed while A.L. Wolf was a visitor at the Dip. di Scienze dell'Informazione, supported in part by the Universitá degli Studi di Roma *La Sapienza* and by the Air Force Materiel Command, Rome Laboratory, and the Defense Advanced Research Projects Agency under Contract Numbers F30602-94-C-0253 and F30602-98-2-0163. The work of F. Parisi-Presicce was partially supported by the EC under TMR Network GETGRATS (GEneral Theory of GRAph Transformation Systems) and Esprit Working Group APPLIGRAPH.

– *Variant:* a relationship reflecting the variation in the realizations of an artifact to fit within different contexts. A variant of an artifact is considered to be an *alternative* to other variants of that artifact, where an alternative is chosen based on an environmental concern such as target operating system.
– *Configuration:* a set of artifacts considered to be complete and compatible with respect to some model of a system. A configuration is made up of one revision of one variant (i.e., a *version*) of each distinct artifact that is a component of the system.

Tools supporting the CM activity are responsible for recording the relationships among versions of artifacts in a *repository*, as well as for enforcing the *policies* by which developers are permitted to manipulate artifacts to create versions and their relationships. In effect, version relationships induce a graph, called the *version graph*, and policies determine a set of legal version graphs.

Looking at the landscape of CM tools, we can see a large number and wide variety of policies [3]. For example, SCCS [11] directly supports only revisions and not variants. Access to artifacts is controlled through a mechanism called *check-out/check-in* in which a developer must first lock an artifact before it can be modified, and then must release that lock before the changes become visible, and available, to other developers. New revisions are added successively to form a linear chain of versions for each artifact in the repository (Figure 1a). RCS [13] extends SCCS by supporting a version tree for each artifact, where variants are indicated by branches in the tree and revisions are indicated by successive versions forming a trunk or limb of the tree (Figure 1b). CVS [1] is a variant of RCS that does not support locking. Instead, CVS allows developers to concurrently make changes to private copies and then later merge them. In effect, CVS turns the RCS version tree into a more general directed acyclic graph (Figure 1c). DVS [2] is a variant of SCCS that follows the locking paradigm and supports revisions, but adds a grouping mechanism based on arbitrary sets of artifacts. The groups, called *collections*, are themselves artifacts and, therefore, are subject to locking and exhibit a recorded revision history (Figure 1d).

SCCS, RCS, CVS, and DVS represent just a small sampling of the many policies that have been invented. New ones appear regularly, some of which are quite involved. An example is a policy that embodies a "deep" semantics for the versioning of collections (e.g., the policy introduced by Lin and Reiss in their programming environment POEM [8]). A deep semantics requires that whenever a new version of an artifact is created, then new versions of any containing collections must also be created. Clearly, this is a recursive definition when collections are themselves treated as artifacts that can be contained in other collections. A simple illustration appears in Figure 2. In 2a is an empty collection. A second version of the collection contains two artifacts, an empty collection and an atomic artifact, as shown in 2b. When an artifact is added to the empty collection, this results in the creation of a new version of that collection, which in turn results in a new version of the top-level collection, as shown in 2c. Notice that the third version of the top-level collection shares the same version of the atomic artifact with the second version of the top-level collection.

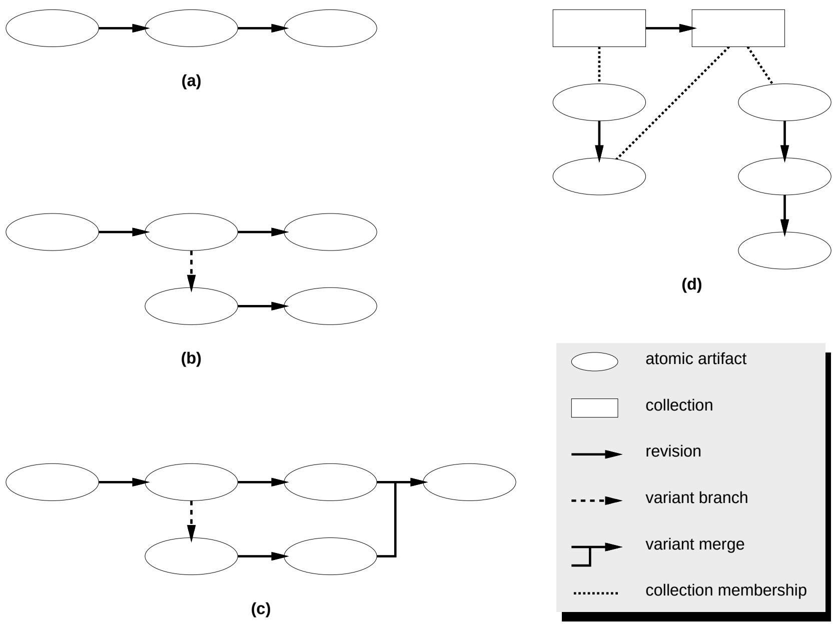


Fig. 1. Example Version Graphs.

Typically, CM policies are embedded deeply within the implementations of CM tools. As a way to make the tools more flexible, van der Hoek et al. [15] have logically separated CM policies from CM mechanisms. Their approach is to define a generic abstraction of a CM repository and to provide a programmatic interface to that repository. Specific CM policies are then realized by programming against this interface. While the approach has been successfully employed in the implementation of a wide variety of CM policies, it suffers from the fact that those policies are being captured at the low level of implementation code written in a procedural programming language. Significant leverage could be achieved if the policies could instead be defined declaratively and at a higher level of abstraction. In particular, a declarative and higher-level specification could lead to a better understanding of the policies, as well as a more appropriate basis upon which to reason about various properties of the policies.

We have begun to develop an improved method for specifying CM policies. The foundation for this method is the theory of *graph transformation systems*. Graph transformation provides an ideal perspective from which to view the problem, since the evolution of artifacts in a CM repository can be seen as a deliberate and regulated transformation of version graphs. We can use this perspective to flexibly define a CM policy in terms of either or both the allowed and the disallowed version graphs, such that the operations applied to an artifact repository are suitably constrained to follow the policy. Perhaps more importantly, we can

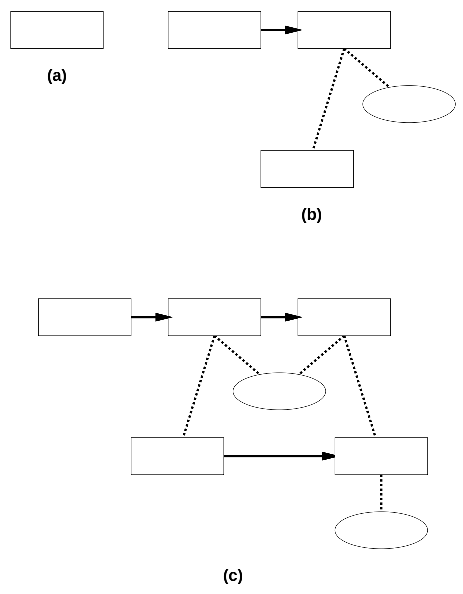


Fig. 2. Example of "Deep" Collection Versioning.

begin to perform meaningful analyses of the relationships among the policies themselves. For instance, if we wish to institute a new policy, is the existing repository compatible with that policy? If we wish to combine the work of two development teams, each of which uses a different CM tool incorporating a different CM policy, will they conflict? If we wish to integrate two policies to form a third, what are the possible ways to do this and what are the properties of the possible policies that arise? The ability to answer these and other such questions has not previously been possible and represents a significant contribution to the field of software configuration management.

This paper introduces our approach to specifying CM policies using graph transformation systems. In the next section we briefly review the basics of graph transformation systems. Section 3 details our use of graph transformation to specify CM policies. Our ability to reason about the relationship between different CM policies is illustrated in Section 4. We conclude in Section 5 with a look at related and future work.

2 Background on Graph Transformation Systems

In this section we recall the basic definitions and properties of typed graphs and typed graph transformation systems. A *graph* $G = (N, E, src, tar)$ is given by a set N of nodes, a set E of edges and functions $src, tar : E \to N$ that assign source and target nodes to edges. A *graph morphism* $f = (f_N, f_E) : G \to G'$ is given by functions $f_N : N \to N'$ and $f_E : E \to E'$ such that $src' \circ f_E = f_N \circ src$ and $tar' \circ f_E = f_N \circ tar$. With identities and composition being defined componentwise, this defines the category **Graph**. To structure graphs [4–6], let $TG \in \mathbf{Graph}$ be a fixed graph, called *typed graph.* A *TG-typed graph* (G, t_G) is given by a graph G and a graph morphism $t_G : G \to TG$. A *(type-preserving) morphism of TG-typed graphs* $f : (G, t_G) \to (G', t_{G'})$ is a graph morphism $f : G \to G'$ that satisfies $t_{G'} \circ f = t_G$. With composition and identities this yields the category $\mathbf{Graph}_{TG}$. Note that $\mathbf{Graph}_{TG}$ is the comma category **Graph** over TG, thus it is complete and cocomplete.

While the type graph TG can be used to classify the components of a graph, *labels* are needed to distinguish elements of the same type. If $C = (C_N, C_E)$ is a pair of disjoint, possibly infinite, sets, then a *C-labelled graph* is a graph G as above along with two labelling functions $c_N : N \to C_N$ and $c_E : E \to C_E$. For simplicity, the adjective "labelled" will be omitted in the rest of the paper.

Definition 1 (Retyping Functors). *Any graph morphism* $f : TG \to TG'$ *induces a* backward retyping functor $f^< : \mathbf{Graph}_{TG'} \to \mathbf{Graph}_{TG}$*, defined by* $f^<((G', t_{G'})) = (G^*, t_{G^*})$ *and* $f^<(k' : (G', t_{G'}) \to (H', t_{H'})) = k^* : (G^*, t_{G^*}) \to (H^*, t_{H^*})$ *by pullbacks and mediating morphisms as in the following diagram,*

$$\begin{tikzcd}
& H^* \arrow[rr] \arrow[dl, "k^*"'] \arrow[dd] & & H' \arrow[dl, "k'"'] \arrow[dd] \\
G^* \arrow[rr] \arrow[dr] & & G' \arrow[dr] & \\
& TG \arrow[rr, "f"'] & & TG'
\end{tikzcd}$$

and a forward retyping functor $f^> : \mathbf{Graph}_{TG} \to \mathbf{Graph}_{TG'}$*, given by* $f^>((G, t_G)) = (G, f \circ t_G)$ *and* $f^>(k : (G, t_G) \to (H, t_H)) = k$ *by composition.*

As shown by Große-Rhode et al. [5], backward and forward retyping functors are left and right adjoints.

In general, the algebraic approaches to graph transformations (see Rozenberg [12] for a complete treatment) are based on the concept of gluing of graphs, modeled by pushouts in suitable categories: in the Double Pushout (DPO) approach a derivation step is based on a two-pushout construction in the category $\mathbf{Graph}_{TG}$ of (labeled, typed) graphs and graph morphisms while, in the SPO approach, it is defined by a single pushout in the category $\mathbf{Graph}^{\mathbf{P}}_{TG}$ of (labeled) graphs and partial morphisms. Our approach is based on double pushouts, although the specific example investigated uses particular kinds of rules that do not erase and thus can be thought of as rules in either approach.

A *TG-typed graph rule* is a span $((L, t_L) \xleftarrow{l} (K, t_K) \xrightarrow{r} (R, t_R))$ where (L, t_L), (K, t_K), (R, t_R) are typed over the same type graph TG and l, r are TG-typed graph morphisms. The left graph (L, t_L) is matched to the actual graph when the rule is applied and the right graph (R, t_R) is substituted to the occurrence of (L, t_L). The span expresses which items of (L, t_L) are related to which items of (R, t_R), and the interface graph (K, t_K) contains the items preserved by the rule application.

TG-typed rules and TG-typed rule morphisms (as triples $f = (f_L, f_K, f_R)$ of TG-typed graph morphisms compatible with the spans) define, with the component-wise identities and composition, the category $\mathbf{Rule}_{TG}$ as the comma category **Rule** over TG. Since **Rule** is complete and cocomplete, so is $\mathbf{Rule}_{TG}$.

Definition 2 (Typed Graph Transformation System Specification). *A* typed graph transformation system specification (tgts-specification) $\mathbf{G} = (TG, P, \pi)$ *consists of a type graph TG, a set of rule names P and a mapping $\pi : P \to |\mathbf{Rule}_{TG}|$, associating with each rule name a TG-typed rule.*

Definition 3 (Morphisms of Typed Graph Transformation Systems). *A morphism of tgts-specifications (tgts-morphism), $f = (f_{TG}, f_P) : \mathbf{G} \to \mathbf{G}'$ from $\mathbf{G} = (TG, P, \pi)$ to $\mathbf{G}' = (TG', P', \pi')$ is given by an injective type graph morphism $f_{TG} : TG \to TG'$ and a mapping $f_P : P \to P'$ between the sets of rule names, such that $f_{TG}^{>}(\pi(p)) = \pi'(f_P(p))$ for all $p \in P$.*

As shown by Große-Rhode et al. [5], tgts-specifications and morphisms form a category, called **TGTS** closed under colimits.

Notation. If G and G' have the same type, $G \cap G'$ denotes the tgts G'' where the range of π'' is $\pi(P) \cap \pi'(P')$ regardless of the set P'' of names chosen (G'' is well defined up to isomorphism)

Given a tgts-specification $\mathbf{G} = (TG, P, \pi)$, a *direct derivation* $p/m : (G, t_G) \Rightarrow (H, t_H)$ over $\mathbf{G}$ from a graph (G, t_G) via a rule p and a matching morphism $m : (L, t_L) \to (G, t_G)$ is given by the following double pushout diagram

$$\begin{tikzcd}
(L, t_L) \arrow[d, "m"] & (K, t_K) \arrow[l, "l"'] \arrow[r, "r"] \arrow[d, "k"'] & (R, t_R) \arrow[d, "h"'] \\
(G, t_G) & (D, t_D) \arrow[l, "l'"'] \arrow[r, "r'"] & (H, t_H)
\end{tikzcd}$$

in $\mathbf{Graph}_{TG}$, where $\pi(p) = ((L, t_L) \overset{l}{\leftarrow} (K, t_K) \overset{r}{\rightarrow} (R, t_R))$. (G, t_G) is called the *input*, and (H, t_H) the *output* of $p/m : (G, t_G) \Rightarrow (H, t_H)$. A *derivation* $p_1/m_1, \ldots, p_n/m_n : (G, t_G) \Rightarrow (H, t_H)$ over $\mathbf{G}$ from a graph (G, t_G) via rules $p_1, \ldots, p_n$ and matching morphisms $m_1, \ldots, m_n$ is a sequence of direct derivations over $\mathbf{G}$, such that the output of the ith direct derivation is the input of the $(i+1)$st direct derivation. The set of all derivations over $\mathbf{G}$ is denoted $Der(\mathbf{G})$ and, considering $Der(\mathbf{G})$ as the behavior of $\mathbf{G}$, the following property holds [5]:

Proposition 1 (Preservation of Behavior). *Let $f = (f_{TG}, f_P) : \mathbf{G} \to \mathbf{G}'$ be a tgts-morphism. For each derivation $d : (G, t_G) \Rightarrow (H, t_H)$ with $d =$*

$(p_1/m_1; \ldots ; p_n/m_n)$ in $Der(\mathbf{G})$ there is a derivation $f(d) : f^{>}_{TG}(G, t_G) \Rightarrow f^{>}_{TG}(H, t_H)$ in $Der(\mathbf{G}')$, where $f(d) = (f_P(p_1)/f^{>}_{TG}(m_1); \ldots ; f_P(p_n)/f^{>}_{TG}(m_n))$. Moreover, $f^{<}_{TG}(f(d) : f^{>}_{TG}(G, t_G) \Rightarrow f^{>}_{TG}(H, t_H)) = (d : (G, t_G) \Rightarrow (H, t_H))$.

In other words, any graph generated in $\mathbf{G}$ can be generated in $\mathbf{G}'$ after appropriate translation via the type morphism.

The formalism presented so far is not sufficient to model the rules in the example of Figure 2 in the preceding section. What is needed is to add to the rules some *context* conditions that prevent the application of a rule even in the presence of a match m.

Definition 4 (Application Conditions).

- *An* application condition *for a match $m : L \to G$ is a total graph morphism $c_i : L \to L_i$.*
- *A* positive *application condition c_i is satisfied by m if there exists a (total) graph morphism $n : L_i \to G$ such that $n \circ c_i = m$.*
- *A* negative *application condition c_i is satisfied by m if there is no (total) graph morphism $n : L_i \to G$ such that $n \circ c_i = m$.*
- *A* conditional rule *is a rule p with a set of application conditions* Cond *and a derivation $p/m : (G, t_G) \Rightarrow (H, t_H)$ takes place only if the match m satisfies every condition in* Cond.

Notation. In the remainder of the paper, the pushout object of two morphisms $a \to b$ and $a \to c$ in a cocomplete category is denoted by $b +_a c$. Similarly, in a complete category, the pullback object of the morphisms $b \to d$ and $c \to d$ is denoted by $b \times_d c$.

3 Formalization of CM Policies

Informally, a policy for a software configuration manager describes (among other things): how and when an artifact can be *checked out* for a possible modification; how and who can *check in* an artifact after a possible modification; how to introduce new versions.

Furthermore, a policy specifies which kinds of structures are forbidden and should never be constructed (an example is a cycle of version dependencies) when introducing new versions. Finally, a policy should keep track of the *current* environment and be able to specify which developers can access which parts of the systems, each with the allowed set of rules to modify the repository model.

We are not addressing here the problem of describing and integrating different environments, which will be tackled in a subsequent paper. The objective is to give a formal framework to describe wanted and unwanted structures.

Definition 5 (Policy). *A policy A is a triple (T, Pos, Neg) where*

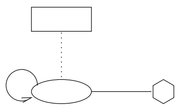
- *$T = (C, TG)$ is the type of the policy consisting of a set C of labels and a type graph TG;*

– *Pos is a* (C,TG)*-graph transformation system; and*
– *Neg is a set of* (C,TG)*-graphs and* (C,TG)*-graph morphisms.*

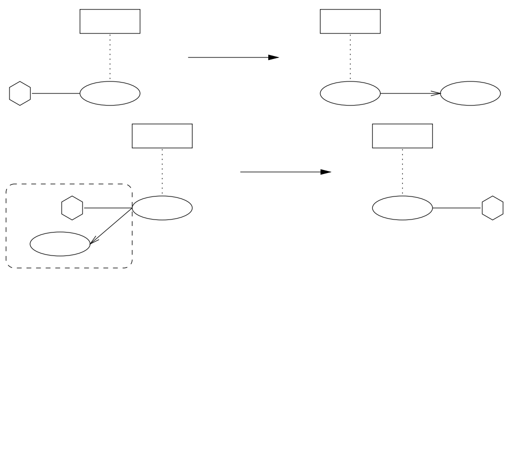
The three components are denoted by $T(A)$, $Pos(A)$ *and* $Neg(A)$*, respectively and* $GPos(A)$ *denotes the set of graphs generated by* $Pos(A)$ *(starting from the empty graph).*

Interpretation. The first component describes the "type" of the policy, with the type graph TG indicating what kind of entities it deals with (for example, artifacts can only be subject to revisions, or the "receiving" end of a membership relation can only be a collection) and the labels in C denoting, for example, the numbering to be used for revisions or for variants. The second component $Pos(A)$ describes intentionally the graphs that the policy intends to generate by giving the rules to do so. The third component describes the unwanted structures in an extensional way. Any graph $H \in Neg(A)$ indicates that no graph containing H (via a morphism) can be accepted. Any morphism $N \to E$ contained in $Neg(A)$ indicates that no graph containing N can be accepted unless it contains also E. Formally, a graph G is acceptable by H if there is no morphism $H \to G$ and a graph G is acceptable by $N \to E$ if any injective morphism $N \to G$ can be extended to a morphism $E \to G$ such that $N \to E \to G = N \to G$. In general, a set of graphs $\mathbf{G}$ is acceptable by $Neg(A)$ if any $G \in \mathbf{G}$ is acceptable by every $H \in Neg(A)$ and every $N \to E \in Neg(A)$.

Example 1. The definitions are illustrated with a formalization of the policy REV that allows the explicit revision of artifacts. Since the policy does not delete any item, the rules have $L = K$ and thus only $K \to R$ is shown. The policy is over the type graph T_R

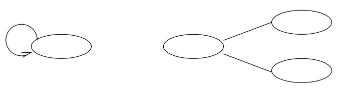


where the rectangle represents a collection that contains the connected artifact, the hexagon is a tag indicating that the connected artifact is checked out and the loop on the ellipsis that the artifact can have a revision. $Pos(REV)$ contains one rule that allows a checked-out artifact to be checked-in and declared a revision of the previously checked-out artifact, and one rule that allows the checking-out of an artifact provided that it has not been checked-out already and does not have a revision (for the rule to be applicable, the matching morphism cannot be extendable to the part [negative condition] enclosed in the dashed rectangle).



$Pos(REV)$ also contains the three rules needed to implement the "deep" collection versioning (not included for lack of space), a rule to add a new artifact to an existing collection and a rule with empty left-hand side to introduce one node representing a new collection. (The user determines which of the two rules is "current".)

$Neg(REV)$ contains only two graphs: one stating that no artifact can be a revision of itself and the other that an artifact can have only one revision.



Definition 6. *A policy A is* coherent *if $GPos(A)$ satisfies $Neg(A)$, i.e., if its rules cannot generate a graph containing an unwanted graph.*

A coherent policy A is closed *if any graph not in $GPos(A)$ is rejected by $Neg(A)$, i.e., if the positive and negative parts of A describe all the graphs over $T(A) = (C,TG)$.*

Example 2. (cont.)
It is not difficult to check that the policy REV given above is a coherent policy since the second rule controls (at check-out time) that an artifact is not already revisioned and can be checked-out, while the only way to introduce a revision is by the first rule and only for checked-out artifacts.

In order to compare policies, it is helpful to view any $H \in Neg(A)$ as the identity morphism $H \to H$ (**NOT** to be interpreted as a morphism $N \to E$ in $Neg(A)$!) . This allows us to treat $Neg(A)$ as a tgts with rules $(H \leftarrow H \rightarrow H)$ and $(N \leftarrow N \rightarrow E)$ and thus to use tgts-morphisms. Policies can be compared by comparing their two significant components Pos and Neg. Consider, for example, a pessimistic policy PES prescribing that only the last version of an artifact can be further versioned and an optimistic policy OPT that allows any version, and not only the last one, to be versioned again. The pessimistic policy generates only graphs that are acceptable by the optimistic policy, while the converse need not be true, i.e., $GPos(PES) \subseteq GPos(OPT)$. Furthermore, any graph rejected by OPT is also rejected by PES, which deals with a particular version (the last one) among those dealt with by OPT. This situation can be formalized by the notion of subsumption.

Definition 7 (Subsumption). *A policy A* subsumes *a policy B of the same type (C,TG) if $GPos(B) \subseteq GPos(A)$ and $Neg(A) \subseteq Neg(B)$.*

The idea can be generalized by a morphism between policies.

Definition 8 (Policy Morphism). *A policy morphism $f : A \to B$ between policies A and B is a triple (f_T, f_P, f_N) where*

- *$f_T = (f_C, f_{TG}) : (C_A, TG_A) \to (C_B, TG_B)$ is a pair consisting of a total function and a total (untyped) graph morphism and*

– *$f_P : Pos(A) \to Pos(B)$ and $f_N : Neg(B) \to Neg(A)$ are tgts-morphisms as in Def.3 with respect to the type morphism f_T.*

Remark 1. A tgts-morphism f_N indicates that policy A rejects at least all the graphs that policy B rejects (up to retyping) and possibly more. A tgts-morphism f_P indicates that, up to the retyping induced by f_T, policy B has all the rules of policy A and thus can generate all the graphs generated by A (Propostion 1). Hence if the types of A and B are the same and there is at least one policy morphism $A \to B$, then B subsumes A. The converse is in general not true because f_P relates the **rules** of the two policies: there may not be any tgts-morphism $f_P : Pos(A) \to Pos(B)$ and yet $GPos(A) \subseteq GPos(B)$.

Policy morphisms can easily be composed component-wise: each component is the composition of functions (f_C), of graph morphisms (f_{TG}) or of tgts-morphisms (f_P and f_N), which is associative with the usual identities. Working component-wise, we can prove the following result.

Theorem 1. *The category* POLICY *of policies and policy morphisms is closed under finite colimits*

Intuitively, the pushout of two policy morphisms $A_0 \to A_1$ and $A_0 \to A_2$ constructs a new policy $A_1 +_{A_0} A_2$ by taking the pushout of the positive rules and the pullback of the negative graphs and morphisms. The policy so constructed need not be coherent even if the policies A_i are coherent. We address this problem at the end of the next section with Theorem 3.

4 Relationships Between Policies

In this section we investigate ways of combining policies to obtain other policies. Unless otherwise specified, we consider policies of the same type $T = (C, TG)$. This assumption is harmless and simplifies the treatment (any graph of type $T_0 = (C_0, TG_0)$ can be considered of type $T_1 = (C_1, TG_1)$ provided that there exists a morphism $f_T : (C_0, TG_0) \to (C_1, TG_1)$) by allowing "set-theoretic" manipulations of policies.

There are (at least) three different ways of combining the negative parts $Neg(A)$ and $Neg(B)$ to obtain the negative part of their combination. The resulting policy rejects a graph G if it contains

– a subgraph forbidden by either A or B;
– a subgraph forbidden by A and one forbidden by B; or
– a subgraph forbidden by both A and B.

More formally

Definition 9 (Negative Strategies). *For sets of morphisms $Neg(A)$ and $Neg(B)$, define*

- $CA(Neg(A), Neg(B)) = Neg(A) \cup Neg(B)$
- $CD(Neg(A), Neg(B)) = \{N_A{+}N_B \to E_A{+}E_B : N_A \to E_A \in Neg(A), N_B \to E_B \in Neg(B)\} \cup \{H_A + H_B : H_A \in Neg(A), H_B \in Neg(B)\}$
- $DA(Neg(A), Neg(B)) = Neg(A) \cap Neg(B)$

Interpretation. A graph is rejected by $CA(Neg(A), Neg(B))$ if it is rejected by policy A or by policy B or by both: it is a CAutious strategy rejecting a graph even if one of the policies could accept it. A graph is rejected by $DA(Neg(A), Neg(B))$ if it is rejected by both policies for the same reason: it is a DAring strategy rejecting a graph only if there is no choice. A graph is rejected by $CD(Neg(A), Neg(B))$ if it is rejected by both policies for possibly different reasons.

Analogous to the negative part, there are (at least) three different ways of combining the generative parts $Pos(A)$ and $Pos(B)$ to obtain the positive part of the combination of the policies A and B.

Definition 10 (Positive Strategies). *Given graph transformation systems $Pos(A)$ and $Pos(B)$, define*

- $CA(Pos(A), Pos(B)) = Pos(A) \cap Pos(B)$
- $CD(Pos(A), Pos(B)) = \{p_A + p_B : p_A \in Pos(A), p_B \in Pos(B)\}$
- $DA(Pos(A), Pos(B)) = Pos(A) \cup Pos(B)$

Interpretation. The graphs generated by $CA(Pos(A), Pos(B))$ are (some of) the graphs generated by both $Pos(A)$ and $Pos(B)$: a CAutious strategy. The graphs generated by $DA(Pos(A), Pos(B))$ includes all the graphs in $GPos(A) \cup GPos(B)$ along with the graphs obtained by the "interaction" of the rules of the two sets: a DAring strategy. The graphs generated by $CD(Pos(A), Pos(B))$ are those obtained by taking the disjoint union of one graph generated by $Pos(A)$ and one by $Pos(B)$.

Policies can be combined by selecting one strategy for the generative part and one for the rejecting part.

Definition 11. *Given policies A and B, the combination of A and B with strategies X and Y is denoted by $[X, Y](A, B)$ and is the policy C where*
$Pos(C) = X(Pos(A), Pos(B))$ and
$Neg(C) = Y(Neg(A), Neg(B))$
for $X, Y \in \{DA, CD, CA\}$

The first result on combining policies is a straightforward application of the definitions.

Proposition 2. *If A and B are policies such that there exists a policy morphism from A to B, then*

1. $[DA, DA](A, B) = B$
2. $[CA, CA](A, B) = A$

The main problem in combining policies is to predict the behavior of the resulting policy. In particular, the two policies to be combined could "interfere" with each other where one of the two generates a graph that is rejected by the other policy. Which of the different ways of combining two coherent policies generates again a coherent policy? The remaining part of this section is devoted to giving partial answers to this question. One special case is already treated in the previous proposition. The two **extreme** ways of combining policies use DAring strategies for both components generating more graphs than the two policies generate individually and rejecting only when both policies agree, and CAutious strategies for both components generating a "small" set of graphs and rejecting a graph when just one of the policies rejects it.

Proposition 3. *1. If A and B are coherent, then $C = [CA, CA](A, B)$ is coherent.*
2. There exist coherent policies A and B such that $D = [DA, DA](A, B)$ is not coherent.

Proof. (Sketch) (1) Since $GPos(C) \subseteq GPos(A) \cap GPos(B)$ and $Neg(C) = Neg(A) \cup Neg(B)$, if $G \in Neg(C)$ is a subgraph of $H \in GPos(C)$, then it is a graph generated by both A and B contradicting the coherence of A if $G \in Neg(A)$ or the coherence of B if $G \in Neg(B)$.

(2) Consider in fact a type graph for both policies consisting of 2 isolated nodes (call them **a** and **b**). Policy A (resp. B) has only one rule generating from the empty graph one with a single node of type **a** (resp. **b**). Both policies reject all the graphs that contain a node of type **b** and a node of type **a**. The two policies are obviously coherent but $Pos(D)$ contains a graph with one node of type **a** and one node of type **b** and thus rejected by definition of $Neg(D)$.

The next few results try to narrow the gap between these two extremes.

Proposition 4. *If A and B are coherent, then $[CA, X](A, B)$ is coherent for any $X \in \{CA, CD, DA\}$.*

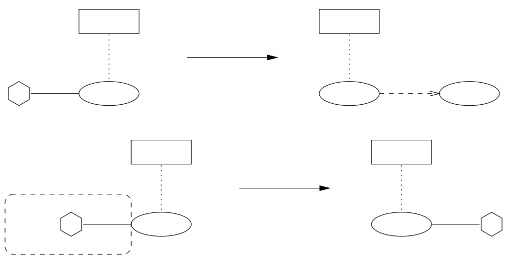
Proposition 5. *(a) For any $X \in \{CA, CD, DA\}$, if $[X, CA](A, B)$ is coherent, then so are $[X, CD](A, B)$ and $[X, DA](A, B)$*
(b) There exist coherent policies A, B, P, and Q such that $[CD, DA](P, Q)$ and $[CD, CD](A, B)$ are not coherent.

The crucial case is when the largest number of graphs is generated while allowing either policy to reject a graph.

Theorem 2. *If $[DA, CA](A, B)$ is coherent, then $[X, Y](A, B)$ is coherent for any $X, Y \in \{CA, CD, DA\}$.*

Example 3. (cont.)
Consider the policy VAR over the same type graph $T_V = T_R$ used for the policy REV but where the loop on the artifact node indicates a variant of an

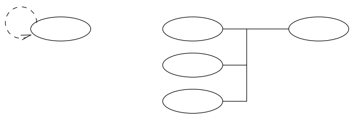
artifact. $Pos(VAR)$ contains one rule that allows a checked-out artifact to be checked-in and declared a variant of the previously checked-out artifact, and one rule that allows the checking-out of an artifact provided that it has not been checked-out already. (Again the user selects the "current" rule.)



(Again the user selects the "current" rule)

$Pos(VAR)$ also contains rules to add a new collection and to add a new artifact within an existing collection.

$Neg(VAR)$ contains only two graphs: one stating that no artifact can be a variant of itself, and the other one that no more than two variants can be merged at a time.



Again it is easy to check that VAR is a coherent policy.

The two policies REV and VAR can be thought of the same type, namely T_R with **two** distinct loops, one denoting revision and one denoting variant.

Claim. The policy $M = [DA, CA](REV, VAR)$ is coherent.

In fact, the interaction of the rules of REV and VAR cannot generate the forbidden graphs since, for example, REV is coherent and the rules in $Pos(VAR)$ cannot generate "variant" arcs. In other words, there are **no** forbidden graphs over the "common" type consisting of the graph T_R without the loop.

The idea behind this example can be generalized. To determine whether the most "dangerous" combination $[DA, CA]$ of coherent policies is coherent, it is sufficient to check only the forbidden graphs of either policy that are of the common type.

Theorem 3. *Let A and B be coherent policies over types T_A and T_B, respectively, and $f_A : T \to T_A$, $f_B : T \to T_B$ type morphisms with $f : T \to T_A +_T T_B$. Denote by $A*$ and $B*$ the policies A and B, respectively, viewed over the type $T_A +_T T_B$. The policy $[DA, CA](A*, B*)$ is coherent if and only if the policy $(DA(A*, B*), NEG)$ is coherent, where*
$NEG = \{n \in Neg(A*) \cup Neg(B*) : f^{>}(f^{<}(n)) = n\}$.

Notice that $[DA, CA](A*, B*)$ corresponds to the pushout object (in POLICY) with respect to the empty set of shared policy rules (cf. Theorem 1).

We close this section with a simple result involving closed policies.

Definition 12. *Policies A and B over the same type are* equivalent *if $GPos(A) = GPos(B)$ and $Neg(A) = Neg(B)$.*

Proposition 6. *Policies A and B are equivalent if and only if A subsumes B and B subsumes A.*

Theorem 4. *Given closed policies A and B, $[DA, CA](A, B)$ is coherent if and only if A and B are equivalent.*

5 Conclusion

The use of graph transformation systems to model various aspects of software engineering is well established. In the particular area of configuration management, three efforts stand out as representative of related work.

- Heimbigner and Krane [7] use graph transformation systems to model the software build process, which is an orthogonal activity to versioning within the general area of configuration management. The build process describes how tools (e.g., compilers and linkers) should be applied to artifacts (e.g., source files) to derive other artifacts (e.g., object and executable files).
- Westfechtel [16] has developed a graph transformation framework for describing the structure of documents and a particular policy for how document structures should evolve.
- Mens [9] uses labelled typed graphs to represent reusable software components and conditional graph rewriting for describing a particular policy by which those components should evolve.

Our work contrasts with these and related efforts in that it is more generally applied to multiple policies, and to understanding the relationships among those policies.

The choices of *DA*, *CA* and *CD* to construct new policies are just "policies"' themselves on policy building: under investigation are more general ways of putting policies together. Also under study are the possible ways of converting a non-coherent policy into a coherent one by modifying either *Neg* (easy) or *Pos* (not as easy) or both. Such modifications could be modelled within the rule-base framework itself [10].

Our future work is aimed at modeling the full spectrum of existing CM policies and finding further critical properties that relate them to each other. Going further, we plan to design and build a tool to take as input policies specified as graph transformation systems and produce as output policy enforcement code in a procedural programming language. As a first target, we will generate policies implemented as calls to the library functions of the NUCM configuration management repository [15].

References

1. B. Berliner. CVS II: Parallelizing Software Development. In *Proceedings of 1990 Winter USENIX Conference*, pages 341–352. USENIX Association, January 1990.
2. A. Carzaniga. *Distributed Versioning System Manual, Version 1.2.* Department of Computer Science, University of Colorado, Boulder, Colorado, June 1998.
3. R. Conradi and B. Westfechtel. Version Models for Software Configuration Management. *ACM Computing Surveys*, 30(2):232–282, June 1998.
4. A. Corradini and R. Heckel. A Compositional Approach to Structuring and Refinement of Typed Graph Grammars. In *Proc. SEGRAGRA'95 (Graph Rewriting and Computation)*, volume 2 of *ENTCS*, pages 167–176. Elsevier, 1995.
5. M. Große-Rhode, F. Parisi Presicce, and M. Simeoni. Spatial and Temporal Refinement of Typed Graph Transformation Systems. In *Proc. MFCS'98 (Mathematical Foundations of Computer Science)*, volume 1450 of *Lecture Notes in Computer Science*, pages 553–561, 1998.
6. M. Große-Rhode, F. Parisi Presicce, and M. Simeoni. Refinements of Graph Transformation Systems via Rule Expressions. In *Proc. Sixth Int. Workshop on Theory and Application of Graph Transformations (TAGT'98)*, Lecture Notes in Computer Science, 1999. To appear.
7. D. Heimbigner and S. Krane. A Graph Transformation Model for Configuration Management Environments. In *SIGSOFT '88: Proceedings of the Third Symposium on Software Development Environments*, pages 216–225. ACM SIGSOFT, November 1988.
8. Y.-J. Lin and S.P. Reiss. Configuration Management with Logical Structures. In *Proceedings of the 18th International Conference on Software Engineering*, pages 298–307. Association for Computer Machinery, March 1996.
9. T. Mens. Conditional Graph Rewriting as an Underlying Formalism for Software Evolution. In *Proceedings of the International Symposium on Applications of Graph Transformation with Industrial Relevance*, Lecture Notes in Computer Science. Springer-Verlag, 1999. To appear.
10. F. Parisi Presicce. Transformations of graph grammars. In *Proc. 5th Int. Workshop on Graph Grammars*, volume 1073 of *Lecture Notes in Computer Science*, pages 426–442, 1996.
11. M.J. Rochkind. The Source Code Control System. *IEEE Transactions on Software Engineering*, SE–1(4):364–370, December 1975.
12. G. Rozenberg, editor. *Handbook of Graph Grammars and Computing by Graph Transformation*, volume 1. World Scientific, New Jersey, 1997.
13. W.F. Tichy. RCS, A System for Version Control. *Software—Practice and Experience*, 15(7):637–654, July 1985.
14. W.F. Tichy. Tools for Configuration Management. In *Proceedings of the International Workshop on Software Versioning and Configuration Control*, pages 1–20, January 1988.
15. A. van der Hoek, A. Carzaniga, D.M. Heimbigner, and A.L. Wolf. A Reusable, Distributed Repository for Configuration Management Policy Programming. Technical Report CU–CS–849–98, Department of Computer Science, University of Colorado, Boulder, Colorado, September 1998.
16. B. Westfechtel. A Graph-Based System for Managing Configurations of Engineering Design Documents. *International Journal of Software Engineering and Knowledge Engineering*, 6(4):549–583, December 1996.