

## Category of Files and Patches

We model each repository state (“file”) as an object in a category, and each patch as a morphism between files. For example, Mimram and Di Giusto define a category **L** whose objects are files (finite sequences of labeled lines) and whose morphisms are **injective, strictly increasing partial functions** that map unchanged lines in one file into another <sup>1</sup>. Such a morphism represents deleting the lines not in its image and inserting new lines in their place. In this framework, **composition of patches** is simply composition of partial functions (sequential application), and identity morphisms are the “null patch” on each file. Because every patch is reversible (we can undo changes), this patch-category is in fact a **groupoid** – a category where every morphism is invertible <sup>2</sup>. Moreover, **L** carries a **(strict) monoidal structure**: concatenating files corresponds to a tensor product of objects, and independent patches on disjoint file segments tensor as well <sup>3</sup>. Thus patch application is algebraically a (partial) monoidal category of file transformations.

## Pushouts and Merges



Figure: A pushout square in category theory, illustrating the universal merge of two parallel patches  $f : Z \rightarrow X$  and  $g : Z \rightarrow Y$  into a common state  $P$  (the pushout) <sup>4</sup>. In categorical terms, **merging two branches** (two patches applied from a common base) corresponds to forming a *colimit* of the span of patches. Concretely, if Alice and Bob both start from file  $Z$  and apply patches  $f : Z \rightarrow X$  and  $g : Z \rightarrow Y$ , their combined history should factor through a merge patch  $h : X \rightarrow P$  and  $k : Y \rightarrow P$  that satisfies  $h \circ f = k \circ g$ . This is exactly the *pushout* of  $f$  and  $g$  <sup>4</sup> <sup>5</sup>. The pushout object  $P$  is the “most general” merged state: anything derivable from  $X$  or  $Y$  can be obtained from  $P$  <sup>5</sup>.

However, not every pair of patches has a well-defined pushout in the basic category **L**, reflecting merge **conflicts** (e.g. both patches edit the same line incompatibly). To handle this, one works in the *free finite cocompletion* of **L** (often called category **P**) obtained by **adjoining all missing pushouts (and other finite colimits)** <sup>6</sup> <sup>7</sup>. In Pijul’s model (a recent patch-based VCS), the file definition is expanded so

that **all** pushouts (even artificial ones) exist – essentially “adding” conflict states into the file space <sup>7</sup>. Algebraically, merge-commits correspond to categorical colimits (pushouts or more general joins); if the patch-effects commute (no conflict), the pushout reduces to a simple union in a join-semilattice. In fact, this forces merge to be **commutative, associative and idempotent**, making the set of states a join-semilattice <sup>8</sup>: merging either order or duplicates yields the same result. This lattice viewpoint underlies *conflict-free replicated data types (CRDTs)*, where concurrent updates merge by computing a least upper bound (join) in a partial order <sup>8</sup>.

## Algebraic Structures: Inverse Semigroups and Groups

Darcs patch theory can be captured by algebraic structures. One formalization models each **patch effect as an element of an inverse semigroup** <sup>9</sup>. Inverse semigroups generalize partial bijections – just like our patch morphisms – and carry a natural partial order:  $x \leq y$  if  $x$  can be obtained by “restricting”  $y$  via an idempotent. This ordering captures when one patch sequence refines another. Jacobson shows that inverse-semigroup laws encode Darcs’s *commutation* and *permutivity* axioms: commuting two patches corresponds to multiplying their semigroup elements, and conflict resolution corresponds to factoring through idempotents in the semigroup <sup>9</sup> <sup>10</sup>.

Similarly, Roscoe (1990) modeled concurrent updates as a **group-like algebra**, using *group conjugation* to describe how one operation can be “moved past” another <sup>11</sup>. Indeed, the invertibility of patches means our category is effectively a groupoid: each patch  $p : X \rightarrow Y$  has an inverse  $p^{-1} : Y \rightarrow X$  such that  $p^{-1}p = \text{id}_X$  and  $pp^{-1} = \text{id}_Y$  <sup>2</sup>. Thus patch sequences form a *partial group* (a groupoid), where composition and inverses govern sequential edits. From this perspective, reasoning about **equivalence or idempotence** of patch sequences reduces to checking equality in the groupoid (or semigroup) – e.g. applying the same patch twice is idempotent only up to the inverse relation.

## Symmetric Monoidal and Rewriting Models

The series-parallel structure of a two-terminal DAG suggests a **monoidal category** perspective: sequential (series) composition is categorical composition, and parallel forks are handled by a tensor (monoidal) product. If independent patches truly commute, this tensor could be symmetric, embodying the idea that order of parallel branches does not matter. While the basic file-patch category  $\mathcal{L}$  above is monoidal, one can embed patching into richer *symmetric* monoidal categories or *rewriting* frameworks. For instance, patches can be seen as **rewriting rules** on file trees, and merges as *confluence* in a string-diagram or term-rewriting setting. String diagram rewrite theories (symmetric monoidal rewriting) could model merges as commutation of independent operations <sup>12</sup>.

Alternatively, one can view an entire TTSP workflow as a **functor** or labeling of an event structure. Each node (patch) is an event; edges indicate causal order. Categorical event structures label each event by a morphism in the patch category, and merges correspond to gluing along common causes. In general, the theory of higher-dimensional rewriting or *homotopical patch theory* treats patches as higher paths, with homotopies encoding commutation <sup>13</sup>. (For example, Angiuli et al.’s Homotopical Patch Theory presents patch sequences as a higher-inductive type where equalities of outcomes are paths in a homotopy space <sup>13</sup>.) While these approaches are quite abstract, they ensure that merging and reorderings respect the same algebraic laws as in category-based models.

# Lattice and Merge Semantics

Another view emphasizes *lattice*-like structure on repository state. If we identify a branch state with the set of patches applied, then merging branches is set-union, provided patches commute. This yields a **bounded join-semilattice** of states, where the least upper bound (join) of two states is their merge <sup>8</sup>. The CRDT perspective formalizes this: the merge function must compute the join of state-lattice elements <sup>8</sup>, so merges are commutative, associative, and idempotent by design. Git’s “merge commit” can be seen as producing a state that **joins** the two parent states. Formal lattice models (or Boolean algebra models) ensure that repeated merges or different orderings yield the same result (providing eventual consistency). This perspective also supports reasoning about **rollback and retries**: reversing a patch corresponds to moving downwards in the lattice (or applying an inverse morphism in the category), and retries/reorders correspond to choosing another join or lifting a patch sequence through the lattice’s partial order.

## Applying to a Two-Terminal Series-Parallel DAG (TTSPG)

In a TTSP DAG workflow, each node is labeled by a patch and edges specify partial ordering (series or parallel) of patches. The above models apply naturally: sequential edges correspond to composite morphisms, while a parallel fork is a monoidal (tensor) split. A merge point in the DAG is exactly the pushout (colimit) of its incoming branches. Thus, one can interpret the entire DAG as a diagram in the patch category whose colimit (in  $\mathcal{P}$ ) yields the final repository state. Equivalences of patch sequences (different linearizations of the same DAG) follow from the groupoid/semigroup relations (commutation of independent patches). If the patches commute (or we explicitly add merge conflicts as separate states), the DAG’s sources and sinks form a join-semilattice of states.

In practice, one could **functorially map** the TTSPG into the category  $\mathcal{L}$  (or its cocompletion  $\mathcal{P}$ ). Each node’s patch is a morphism in  $\mathcal{L}$ , and compositional and colimit properties guarantee that any valid merge in the DAG corresponds to a unique morphism in  $\mathcal{P}$ . Formal tools like the “category of files and patches” <sup>14</sup> or Darcs’ patch algebra can be embedded into the orchestration: for example, using an inverse semigroup to represent allowable patch reorderings <sup>9</sup>, or treating the DAG itself as a presheaf (as in Mimram’s concrete model of  $\mathcal{P}$ ) <sup>14</sup>. This ensures merges (including conflict resolution) are handled by the universal properties of colimits, and rollback/reordering operations correspond to categorical inverses or lattice downs. Overall, category theory and algebraic models offer a precise language for the TTSPG workflow: patches are morphisms, sequential edges are compositions, parallel branches are monoidal tensors, and merges are pushouts or joins <sup>4</sup> <sup>8</sup>.

**Sources:** Formal theories of Darcs-style patches include the category-theoretic model by Mimram & Di Giusto <sup>1</sup> <sup>4</sup>, Jacobson’s inverse-semigroup algebra <sup>9</sup>, and homotopical patch theory <sup>13</sup>. Pijul’s documentation explicitly uses pushouts to merge branches <sup>5</sup> <sup>7</sup>. The categorical pushout and colimit construction generalizes merges <sup>4</sup>, while join-semilattices and CRDT theory impose commutative, associative, idempotent merge behavior <sup>8</sup>. Roscoe’s and others’ algebraic models emphasize group-like inverse operations <sup>11</sup>. These structures can all be adapted to the TTSPG setting by interpreting series/parallel composition in the DAG as categorical composition/tensor and treating merge commits as universal colimits (or lattice-joins) in the resulting patch algebra.

1 3 4 **A Categorical Theory of Patches**

[https://www.researchgate.net/profile/Samuel-Mimram/publication/258521116\\_A\\_Categorical\\_Theory\\_of\\_Patches/links/546cadb10cf24b753c628ef9/A-Categorical-Theory-of-Patches.pdf](https://www.researchgate.net/profile/Samuel-Mimram/publication/258521116_A_Categorical_Theory_of_Patches/links/546cadb10cf24b753c628ef9/A-Categorical-Theory-of-Patches.pdf)

2 **Patch theory, part II: some basics | blog :: Brent -> [String]**

<https://byorgey.wordpress.com/2008/02/13/patch-theory-part-ii-some-basics/>

5 7 **Pijul - Model**

<https://pijul.org/model/>

6 14 **[1311.3903] A Categorical Theory of Patches**

<https://ar5iv.labs.arxiv.org/html/1311.3903>

8 **Conflict-free replicated data type - Wikipedia**

[https://en.wikipedia.org/wiki/Conflict-free\\_replicated\\_data\\_type](https://en.wikipedia.org/wiki/Conflict-free_replicated_data_type)

9 10 11 **ww3.math.ucla.edu**

<https://ww3.math.ucla.edu/camreport/cam09-83.pdf>

12 **[1602.06771] Rewriting modulo symmetric monoidal structure - arXiv**

<https://arxiv.org/abs/1602.06771>

13 **cs.cmu.edu**

<https://www.cs.cmu.edu/~rwh/papers/htpt/jfp.pdf>