Operational Theories and Computational Substrates

A Sweedler perspective prompted by Denicek

Discussion note for Tomas Petricek and collaborators

17 October 2026 • Draft for discussion, not a verification of Denicek

Abstract

Denicek represents interactive programs as edit histories, using the same small computational substrate to support collaborative editing, programming by demonstration, schema evolution, incremental recomputation, and explanations. This note asks whether *Sweedler theory of monads and comonads* provides a useful semantics of the relationship between a programming-system interface and the machinery implementing it. The key object is a stateful *runner*, which interprets an operational theory T using an implementation theory R. We compare direct document mutation with an append-only history implementation using value-returning observations, then audit strict versus lenient reference semantics: a counterexample defeats an expected simulation, while a guarded adapter restores reference safety and is certified by a Sweedler-coalgebra comparison. We then distinguish the elementary mathematics established here from harder open problems raised by Denicek: semantics of reconciliation, replay of saved interactions, observational equivalence of histories, and composition of certified implementation changes. We also combine the reference-safety audit with proof-carrying reconciliation: a safe-branch counterexample establishes that local safety alone does not ensure safe merging, while a certified rename/add square yields a guarded implementation that safely accepts compatible merges. The organising distinction is between certified commuting squares, which express equations on specified domains, and directed reconciliations, which implement a policy even when no equation holds. These distinctions also suggest an auditing and design discipline. The aim is a basis for conversation, not a verification of Denicek.

1 Why computational substrates invite a semantic account

A programming system is more than a language together with an evaluator. Users interact with an evolving environment: they edit and inspect programs, reuse demonstrations, revisit past versions, and observe the effects of incremental computation. A computational substrate supplies representations and primitive operations from which these experiences can be built. The *Technical Dimensions of Programming Systems* framework makes clear why the choice of these primitives matters [2].

Petricek and Edwards's Denicek is an unusually instructive example [1]. A document is built from records, lists, primitive values, and references; its persistent representation is a history of edits. Three mechanisms—applying histories, merging histories, and detecting conflicts—support otherwise disparate programming experiences. The same edit representation covers user modifications, demonstrated interactions, and formula-evaluation results (§§4–5 of [1]).

This architecture suggests a question that is broader than patch correctness:

> *Can one specify the operations that a programming system exposes, separately from the residual mechanisms through which they are implemented, and prove that changing the substrate preserves the behaviours and guarantees of that interface?*

Sweedler theory provides a candidate language for that separation. It does not, by itself, prove that Denicek's merge algorithm is coherent. Its possible payoff lies instead in comparing *implementations of the same operational interface*, including ones that maintain different persistent state and use different effects.

2 Operational theories, residual theories, runners

Work initially in the standard duoidal setting of accessible endofunctors on **Set**. Let T and R be monads, regarded as an *operational theory* and a *residual implementation theory*, respectively. An R-residual runner with state space S can be presented by a compatible natural family

$$\theta_X : T(X) \times S \longrightarrow R(X \times S). \tag{1}$$

The compatibility conditions ensure that interpreting a returned value does nothing and interpreting a composite program agrees with sequencing the interpretations. In the Sweedler description this is a coalgebra

$$\xi : S \longrightarrow \{T, R\}(S) \tag{2}$$

of the measuring comonad $\{T, R\}$, under the usual existence hypotheses. McDermott–Rivas–Uustalu, Theorem 1, identifies the coalgebras of the Sweedler hom with residual stateful runners for accessible monads on a locally presentable base. This connection is established theory, not a result of this note [3, 5].

The interface represented by T need not expose the operations in R. For example, T may offer AddField, RenameField, and DeleteField while R allows failures or writes to an audit trace. Persistent state lives in S; residual effects live in R. Two systems can implement the same T yet differ in both.

An important distinction follows immediately. Two programs can produce the same *document* but different *residual computations*—different edit traces, costs, failures, or replayable history. Equality should be demanded only at the observational level relevant to the specified interface.

3 A tiny reference-safe document calculus

We first isolate a fragment that has a complete and transparent semantics. This is *Denicek-like*, not the full Denicek language. In particular we omit lists, wildcard selectors, formula evaluation, and reference-behaviour flags.

Fix an infinite set of field names K. Let values be natural numbers or references to field names, $V = \mathbb{N} + \mathsf{Ref}(K)$. A document is a finite partial map $d : K \rightharpoonup V$. Say d is valid when every reference stored in a field resolves:

$$\mathsf{Valid}(d) \iff \forall k \in \mathrm{dom}(d), \quad d(k) = \mathsf{Ref}(j) \Longrightarrow j \in \mathrm{dom}(d). \tag{3}$$

Let D be the set of valid documents. The command signature comprises:

$\mathsf{add}(k, v)$	insert a fresh field, with any referenced field already present;
$\mathsf{rename}(k, \ell)$	rename an existing field to a fresh name, rewriting references;
$\mathsf{delete}(k)$	delete a field only if no remaining field refers to it.

Each command a has an error-sensitive transition $\delta_a : D \to D + E$, where E consists of errors such as missing field, duplicate field, or dangling-reference risk. For example, $\mathsf{rename}(k, \ell)$ replaces the key k by ℓ and simultaneously transforms every occurrence of $\mathsf{Ref}(k)$ into $\mathsf{Ref}(\ell)$. In all successful cases the resulting document remains valid.

Proposition 3.1 (Reference safety)**.** *For every command a and valid document d, a successful result $\delta_a(d) = \mathsf{inl}(d')$ satisfies $\mathsf{Valid}(d')$. Consequently every successfully executed finite command history preserves validity.*

Proof. Insertion checks freshness and the validity of a newly introduced reference. Renaming is a bijective change of a present field name together with all its occurrences in references. Deletion checks that no surviving reference names the removed field. These observations prove preservation for one command; induction on the history length gives the second assertion. □

3.1 A certified reconciliation square

Fix three distinct field names a, b, c. To avoid overloading a as a command, put

$$e = \mathsf{add}(b, \mathsf{Ref}(a)), \qquad \rho = \mathsf{rename}(a, c), \qquad e' = \mathsf{add}(b, \mathsf{Ref}(c)).$$

Composition of histories is read from left to right ($e; \rho$ means execute e, then ρ); function composition $\circ$ is read from right to left.

Proposition 3.2 (A reconciliation square)**.** *For every valid document d, $[\![e; \rho]\!](d)$ succeeds if and only if $[\![\rho; e']\!](d)$ succeeds, and then the results coincide. Thus*

$$[\![e; \rho]\!] = [\![\rho; e']\!] \quad \textit{as partial maps } D \rightharpoonup D.$$

The equality need not hold as an equation in $D + E$ when distinct failures are observed: one order can return duplicate *and the other* missing*.*

Proof. Both sides succeed exactly when $a \in \mathrm{dom}(d)$, $b \notin \mathrm{dom}(d)$, and $c \notin \mathrm{dom}(d)$. On that domain, both insert $b \mapsto \mathsf{Ref}(c)$, rename the key a to c, and redirect all pre-existing references to a toward c. Outside that domain both computations fail, although they may fail at different steps and with different labels. □

For example, $d = \{a \mapsto 7\}$ lies in the common success domain, with result $\{b \mapsto \mathsf{Ref}(c), c \mapsto 7\}$. By contrast, the valid document $\{b \mapsto \mathsf{Ref}(a), c \mapsto \mathsf{Ref}(a), a \mapsto \mathsf{Ref}(a)\}$ can make the left-hand execution fail with *duplicate*, while the other order fails with *missing*. A finite check reported in the accompanying review found coincident success domains on all 319 valid documents tested, but 98 differing failure labels. The general partial-map result follows from the proof, not from enumeration.

This illustrates the first central distinction of the note. A *certified commuting square* asserts an equation in a specified semantics, here partial maps. Whether the square is also an equation in an exception monad depends on whether the interface makes failure labels observable. A *directed reconciliation* may instead implement a choice of priority or conflict policy, without asserting equality at all.

4 Two residual runners with a value-returning observation

A caveat is worth making explicit. With a command-only free monad $A^* \times X$ (equivalently the writer monad for the free monoid A^*), a runner is just a Kleisli action of A^* on its state. Comparing two such actions is standard monoid-action semantics; simply attaching Sweedler notation would add little. We therefore include an observation whose *return value can determine the rest of a program.*

Let A be the three command families of §3 and retain $V = \mathbb{N} + \mathsf{Ref}(K)$. Add a total operation $\mathsf{get}(k)$ returning an element of $V + 1$, with none for an absent field. The interface

theory T is the free monad for the algebraic signature with commands $a \in A$ of arity 1 and observations $\mathsf{get}(k)$ of arity $V+1$:

$$F(X) = A \times X + \coprod_{k \in K} X^{V+1}, \qquad T = F^*.$$

A term may branch on the result of a read. The direct implementation has state $S_{\mathrm{dir}} = D$ and residual exception monad

$$R_{\mathrm{dir}}(X) = X + E.$$

Its commands act by the δ_a; its observations return $\mathsf{some}(d(k))$ or none and leave the state unchanged. This determines a runner

$$\theta^{\mathrm{dir}}_X : T(X) \times D \longrightarrow R_{\mathrm{dir}}(X \times D).$$

Fix an initial document d_0 and define the alternative persistent state

$$S_{\log} = \{h \in A^* \mid h \text{ executes successfully from } d_0\}, \qquad q = \mathsf{fold} : S_{\log} \longrightarrow D.$$

An element of $S_{\log}$ is an append-only history. For commands, the log implementation first checks $\delta_a(q(h))$ and, on success, stores $h; a$. For $\mathsf{get}(k)$, it reconstructs $q(h)$, returns the requested value (or absence), and does not modify h.

Let L be an alphabet of audit events containing command events and, optionally, read events. Use the writer-with-exceptions residual monad

$$R_{\log}(X) = (L^* \times X) + E.$$

The log runner emits the corresponding events for successful steps and has the same errors as the direct runner. There is a monad morphism

$$\alpha_X : (L^* \times X) + E \longrightarrow X + E, \quad \alpha_X(\ell, x) = x, \quad \alpha_X(e) = e. \tag{4}$$

Proposition 4.1 (Implementation comparison)**.** *For every program $t \in T(X)$ and $h \in S_{\log}$,*

$$\theta^{\mathrm{dir}}_X(t, q(h)) = R_{\mathrm{dir}}(\mathrm{id}_X \times q)(\alpha_{X \times S_{\log}}(\theta^{\log}_X(t, h))). \tag{5}$$

The two implementations agree on returned observation values, success or failure (including error labels), and resulting visible documents. In particular, the logged implementation inherits reference safety.

Proof. Proceed by structural induction on the well-founded free-monad term. Returned values agree by the unit law. At a command node, both implementations check δ_a at the same reconstructed document; if successful, $q(h; a) = \delta_a(q(h))$, and α discards the audit event. At a $\mathsf{get}(k)$ node, both compute the same element of $V+1$ from $q(h)$, and therefore select corresponding continuations; apply the induction hypothesis to the selected branch. The induction also covers failure, since both use the same labelled errors. Reference safety follows from Proposition 3.1. □

A concrete value-dependent program reads a and conditionally inserts a fresh field if the returned value is $\mathsf{some}(7)$. The two runners agree not only about the document update but about *which continuation is selected*. This is the modest extra expressivity missing from the command-only comparison.

Proposition 4.2 (Implementation comparison in Sweedler form)**.** *Let the corresponding coalgebras be*

$$\xi_{\mathrm{dir}} : D \to \{T, R_{\mathrm{dir}}\}(D), \qquad \xi_{\log} : S_{\log} \to \{T, R_{\log}\}(S_{\log}).$$

The monad morphism α induces a comonad morphism

$$\{T, \alpha\} : \{T, R_{\log}\} \longrightarrow \{T, R_{\mathrm{dir}}\}.$$

Then q is a $\{T, R_{\mathrm{dir}}\}$-coalgebra morphism

$$(S_{\log}, \{T, \alpha\} \circ \xi_{\log}) \longrightarrow (D, \xi_{\mathrm{dir}}),$$

and this condition is equivalent to equation (5)*.*

Proof. For fixed T, postcomposition of the universal measuring interaction with a monad morphism $\alpha : R \to R'$ induces $\{T, \alpha\}$ by the universal property of the Sweedler hom; compatibility with comonad structure follows by uniqueness. Under the natural runner–coalgebra correspondence of McDermott–Rivas–Uustalu, Theorem 1, postcomposition by α sends $\xi_{\log}$ to the runner $\alpha\theta^{\log}$. The definition of a coalgebra morphism is then precisely the natural family of equations (5). This uses the standard naturality of the representing correspondence, rather than a new Denicek-specific theorem. □

4.1 What the example does and does not earn

Proposition 4.1 is an elementary simulation theorem, now allowing branching on returned values; Proposition 4.2 locates it within the functoriality of the Sweedler measuring construction. Neither claims that Sweedler theory is needed to prove this particular simulation. Its prospective payoff is organising comparisons among richer interfaces and different residual effects, especially when observable replay and reconciliation become part of T.

These runners implement *execution*, not Denicek's full merging, conflict policies, saved-history hash updates, or ordered insertion. The strict fragment is also deliberately *not* a faithful account of Denicek's runtime: source-code analysis reported that missing targets are treated as no-ops, Add behaves as an upsert without checking newly introduced references, renaming onto an existing name may overwrite it, and only selected deletion and copying operations report reference-related errors. These observations concern the examined code/port and require confirmation against the precise version of the running system. In particular, the reference-safety theorem for the strict runner must not be attributed to Denicek's implementation.

This mismatch is a useful second comparison rather than an embarrassment. A *strict* runner and a *lenient* runner can implement the same command signature but fail to satisfy an error- and reference-preserving simulation: the latter may permit a successful update leaving a dangling reference. Determining a weaker observational boundary, or adding a validating residual handler, becomes a concrete design question. A miniature counterexample and a certified validating adapter are developed in Section 5. Confirming a corresponding trace in the source version being discussed remains a separate task.

5 An audit that prescribes a repair: strict and lenient references

The preceding comparison establishes a positive preservation theorem between two implementations deliberately built to use the same checking semantics. We now ask a more diagnostic question: what if the second implementation has different rules for accepting edits? This gives a miniature example of using runner semantics not only to *verify* a design but to *identify a repair*.

The source-code examination reported in the accompanying review found a lenient policy in the examined Denicek implementation: adding a field is an upsert without validating references in the inserted node; a missing edit target may be ignored; renaming onto an existing name can overwrite it. These are claims about that examined implementation/port, not conclusions derivable from the published paper alone and not a comprehensive validation of a running deployment. We model only the first behaviour. We do not claim that Denicek promises our stricter invariant or that a permanently valid document is always desirable while editing.

5.1 One value-returning interface, two incompatible runners

Let $\widehat{D}$ be the set of *all* finite partial maps $K \rightharpoonup V$, with $V = \mathbb{N} + \mathsf{Ref}(K)$ as in §3, and let $D \subseteq \widehat{D}$ be the valid maps. Retain a free effect theory T_0 with the commands $\mathsf{add}(k, v)$ and the value-returning operations $\mathsf{get}(k) : V + 1$. In particular, a program can read a reference-valued field, extract its target key, and inspect whether the target exists. This is an *audit interface* extracted from the document calculus, not an assertion about Denicek's precise user-level API.

The *strict* runner has state D and residual monad $R_{\mathrm{s}}(X) = X + E$. Its insertion transition is

$$\delta^{\mathrm{s}}_{\mathsf{add}(k,v)}(d) = \begin{cases} \mathsf{inr}(\text{duplicate}) & k \in \mathrm{dom}(d), \\ \mathsf{inr}(\text{badref}) & v = \mathsf{Ref}(j),\ j \notin \mathrm{dom}(d), \\ \mathsf{inl}(d[k \mapsto v]) & \text{otherwise.} \end{cases}$$

Here the first test takes priority, and the read $\mathsf{get}(k)$ returns the stored value or none without changing state. All successful strict executions remain in D by Proposition 3.1.

The *raw lenient* runner has state $\widehat{D}$ and residual monad $R_\ell(X) = (L^* \times X) + E$, which records audit events on successful steps. Its insertion transition is the upsert

$$\delta^{\ell}_{\mathsf{add}(k,v)}(d) = d[k \mapsto v],$$

without a freshness or target-existence check. Its reads have the same $V + 1$ response type. Let $\alpha : R_\ell \to R_{\mathrm{s}}$ forget the log and preserve labelled errors, as in (4). This choice of α is part of the specified observational boundary.

Proposition 5.1 (Safety failure and simulation obstruction)**.** *There are valid d and commands e for which the raw lenient runner succeeds and leaves D, while the strict runner reports an error. In particular, the trace-erasing comparison α cannot yield an error- and read-preserving runner simulation extending the identity comparison on all valid starting documents for the full T_0 interface.*

Proof. Choose distinct keys a, b, x and put $d_0 = \{a \mapsto 7\} \in D$ and $e = \mathsf{add}(b, \mathsf{Ref}(x))$. The strict transition returns badref. The lenient transition succeeds with

$$d_1 = \{a \mapsto 7,\ b \mapsto \mathsf{Ref}(x)\} \notin D.$$

Trace erasure maps a successful lenient result to a successful result, not to badref. Thus the comparison equation already fails on the one-command program e. Furthermore, after the lenient step the observation $\mathsf{get}(b)$ returns $\mathsf{some}(\mathsf{Ref}(x))$, whereas $\mathsf{get}(x)$ returns none. The discrepancy cannot be explained away by preserving current values while ignoring the newly dangling reference. This exhibits an obstruction for the specified observable interface and residual map, not for every conceivable abstraction or state-dependent repair. □

The obstruction is not merely failure of the safety invariant. For example, from the *safe* state $\{b \mapsto 7\}$, an incoming $\mathsf{add}(b, 8)$ leaves a safe document under the lenient upsert but gives a duplicate error under the strict semantics. Hence restricting attention to safe starting states and safe *outcomes* does not suffice. The interface must also specify command admissibility and whether errors are observable.

5.2 A certified adapter using the lenient primitive

The audit identifies a small repair. Retain the raw lenient upsert as a *low-level* primitive, but implement the *public* Add operation by a guard:

$$\delta^{\mathrm{g}}_{\mathsf{add}(k,v)}(d) = \begin{cases} \text{duplicate} & k \in \mathrm{dom}(d), \\ \text{badref} & v = \mathsf{Ref}(j),\ j \notin \mathrm{dom}(d), \\ \text{upsert and record event} & \text{otherwise.} \end{cases}$$

The guarded runner has state D and residual monad R_ℓ; its reads return the same observations as the strict runner. The guard and write are assumed atomic relative to other edits. In a concurrent implementation this is a substantive requirement (one might use a transaction), not a consequence of the runner laws.

Proposition 5.2 (Certified repair, in runner and Sweedler form)**.** *Let θ^{g} be the guarded runner of T_0 in R_ℓ, and θ^{s} the strict runner of T_0 in R_{s}, both with state D. Then for every finite computation tree $t \in T_0(X)$ and $d \in D$,*

$$\theta^{\mathrm{s}}_X(t, d) = \alpha_{X\times D}(\theta^{\mathrm{g}}_X(t, d)). \tag{6}$$

In particular, the guarded implementation preserves reference validity, all returned read values, success/failure, and labelled errors. Under the runner–coalgebra correspondence, with ξ^{g} and ξ^{s} the corresponding measuring-comonad coalgebras, this is equivalent to

$$\{T_0, \alpha\}_D \circ \xi^{\mathrm{g}} = \xi^{\mathrm{s}}.$$

Proof. Induct on t. A return is unchanged by α. At a read node, both runners return the same element of $V + 1$, so follow the same continuation, to which the induction hypothesis applies. At an Add node, the two guards test exactly the same preconditions in the same order. If a test fails, both return the same error; otherwise, the lenient primitive is invoked on an admissible input and produces precisely the strict successor document, with only its additional audit record erased by α. Consequently D is closed under every successful guarded step. The final coalgebra equation follows from the naturality, in the residual monad, of the represented runner–coalgebra correspondence, as in Proposition 4.2. □

5.3 The design report

We have three distinct, reproducible verdicts for the same public signature:

1. *Strict implementation:* certified preservation of valid references and explicit errors.
2. *Raw lenient implementation:* a one-step counterexample; no trace-erasing, error-preserving comparison with the strict interface.
3. *Guarded lenient implementation:* the strict guarantees are recovered by an explicit adapter, certified by a coalgebra comparison under $\{T_0, \alpha\}$.

A finite sanity check used three field names, two primitive values, and reference values to each of the three names. Among 216 finite documents, 183 satisfied the chosen invariant. On those states, all 2,745 single-command cases and all 41,175 two-command histories agreed under strict versus guarded execution; 138 raw lenient one-step transitions left the invariant. These finite tests independently corroborate the restricted calculation, but the general result is the induction in Proposition 5.2, and none of the tests exercise Denicek's actual source code.

This is an auditing-to-design inference, rather than merely a post-hoc proof. It says which implementation change suffices for a specified guarantee. Alternative decisions are legitimate: one may expose dangling references as permitted states, or allow temporarily invalid states inside transactions while enforcing validity only at commit. Those are *different operational*

specifications, requiring different invariants and comparisons. The next section supplies a certified reconciliation protocol in this restricted calculus, but this does not verify the full Denicek reconciliation/replay machinery; its local editing clauses and policy choices must still be checked.

6 Certified merging as a safety-preserving design discipline

The preceding audit concerns individual commands and their implementations. Merging is a strictly stronger obligation: two branches can each be executable and reference-safe, yet their combination can be unsafe. In this section we connect the checking runner of §5 to *proof-carrying reconciliation*. The result is a design criterion, not an assertion that Denicek already enforces it.

6.1 Two safe branches, no safe union of their intended effects

Take $d_0 = \{a \mapsto 7\}$ and let

$$e = \mathsf{add}(b, \mathsf{Ref}(a)), \qquad k = \mathsf{delete}(a), \qquad a \neq b.$$

Both one-edit histories execute successfully from d_0 in the strict calculus: e adds a reference to a present field, while k deletes an unreferenced field. Yet the two concatenations cannot both succeed:

$$[\![e; k]\!]_{\mathrm{s}}(d_0) = \text{referenced},$$
$$[\![k; e]\!]_{\mathrm{s}}(d_0) = \text{badref}.$$

If the second order is implemented by unchecked upsert, it instead succeeds and yields the *invalid* document $\{b \mapsto \mathsf{Ref}(a)\}$. This is a miniature model of a failure mode reported for the examined Denicek code/port, not a claim to have verified the full running system.

Proposition 6.1 (Incompatibility of the two requirements)**.** *No valid document can simultaneously satisfy all three postconditions*

$$a \notin \mathrm{dom}(d), \qquad b \in \mathrm{dom}(d), \qquad d(b) = \mathsf{Ref}(a).$$

Thus no merge preserving both the deletion of a and the added reference to a can satisfy reference validity.

Proof. Validity of d and $d(b) = \mathsf{Ref}(a)$ imply $a \in \mathrm{dom}(d)$, contradicting the first postcondition. □

A safe substrate must therefore report a conflict or adopt an explicit policy: for example, keep a, drop the insertion, or require a justified retargeting. These are *directed resolutions*, not semantic equalities. The distinction is meaningful even though both original branches were individually safe.

6.2 Certificates for successful reconciliation squares

Let $\mathcal{D} = D$ denote the valid documents and let $[\![h]\!]_{\mathrm{s}} : D \to D + E$ be strict execution of a history h. A *successful reconciliation certificate* for histories h_1, h_2 over an admissibility predicate $P \subseteq D$ consists of proofs that, for each $d \in P$,

$$[\![h_1]\!]_{\mathrm{s}}(d) = \mathsf{inl}(d') = [\![h_2]\!]_{\mathrm{s}}(d) \quad \text{for some } d' \in D. \tag{7}$$

In particular, the certificate includes *definedness, equality of the resulting document, and preservation of the invariant.* Unlike an equation merely in the category of partial maps, it is also an equation in the error-sensitive semantics on P: neither path fails there. To preserve more observations one can demand the same equality in a chosen observational residual monad Q.

The rename/add square from Proposition 3.2 supplies an explicit certificate. For distinct a, b, c put

$$P_{a,b,c} = \{d \in D \mid a \in \mathrm{dom}(d),\ b, c \notin \mathrm{dom}(d)\}.$$

Writing $\rho = \mathsf{rename}(a, c)$ and $e' = \mathsf{add}(b, \mathsf{Ref}(c))$, the square

$$\underbrace{\mathsf{add}(b, \mathsf{Ref}(a)); \rho}_{\text{insert, then rename}} \equiv_{P_{a,b,c}} \underbrace{\rho; e'}_{\text{rename, then transported insertion}} \tag{8}$$

is successfully certified: both routes rewrite all references to a and finish in the same valid document. Outside $P_{a,b,c}$, the orders may report *different* error labels; we must not promote the scoped certificate to an unrestricted equality in $D + E$.

6.3 Pasting certificates and transporting them across runners

A *certified merge derivation* is a finite sequence of local rewrites of executable histories. Each replacement is accompanied by a successful certificate on a predicate holding at the intermediate state where the replacement occurs. If a rule produces extra edits, those edits are included in the certified history; no unproved interchange of dependent edits is allowed. We additionally require that every produced intermediate state remain in D.

Proposition 6.2 (Safety and observation of certified merging)**.** *Suppose a candidate merge of two histories is accepted only after a finite certified merge derivation, satisfying the preceding applicability conditions. Then all steps of an accepted merge are reference-safe and the histories connected by the derivation have equal strict execution results on its certified initial domain. Moreover, if θ^{g} is the guarded lenient implementation of Proposition 5.2, then the same histories have equal* trace-erased *results in that implementation:*

$$\alpha\, \theta_1^{\mathrm{g}}(h_1, d) = \alpha\, \theta_1^{\mathrm{g}}(h_2, d), \qquad d \in P. \tag{9}$$

On the certified domain both paths succeed with identical resulting documents. No claim is made about failure labels outside that domain, audit-log equality, or history-sensitive replay.

Proof. For the strict runner, replace one certified subhistory at a time. Its successful local certificate permits substitution within the surrounding sequential context, and strict execution preserves D at every step by Proposition 3.1. Repeating finitely many times gives equality of resulting documents and definedness. Proposition 5.2 identifies $\alpha\theta^{\mathrm{g}}$ with θ^{s} on all programs of the value-returning interface, hence transports the established equality to the guarded implementation. In Sweedler form this last step is exactly postcomposition of its measuring coalgebra by $\{T_0, \alpha\}$. The proof assumes a successful complete derivation, rather than the existence of such a derivation for arbitrary branches. □

Proposition 6.3 (Future direct-document observations)**.** *Under the hypotheses of Proposition 6.2, let $t \in T_0(X)$ be an arbitrary continuation, possibly branching on values returned by* get*. For each $d \in P$, executing either certified merge history and then t yields the same result in the strict semantics, including any later error label. The guarded implementations agree after applying α.*

Proof. Both certified histories succeed with the same state d'. The following computation t therefore starts from the same state in both cases. The runner composition law yields identical results for every continuation and returned value. Transport to the guarded runner uses equation (6). The statement concerns the direct-document interface T_0, not interactions that inspect raw histories or replay saved checkpoints. □

The proposition yields a simple *auditing protocol*: check local admissibility; construct a transported candidate (for example, rewrite $\mathsf{Ref}(a)$ to $\mathsf{Ref}(c)$ across a rename); validate a commuting square on the required domain; and commit only the certified result. When no such certificate exists, report a conflict and make the resolution policy visible. Atomic validation and commit must be supplied by the implementation if other actors can concurrently modify the document. The proof does not show that certificate generation is complete or efficient, or that an existing Denicek conflict detector implements this protocol.

6.4 What this audit reveals about substrate design

The two examples distinguish alternatives that would otherwise be easy to conflate:

1. *Unchecked reconciliation:* a successfully executed merge can produce a dangling reference, as in the delete/add example.
2. *Checked but untransported reconciliation:* reference validity is maintained by rejecting the second edit, but a compatible rename/add merge is needlessly lost.
3. *Certified reconciliation on a guarded substrate:* valid reference transport is accepted with a proof, while genuinely incompatible goals are reported as conflicts or resolved explicitly.

Thus the audit identifies two distinct architectural requirements: *invariant preservation*, supplied by guarded or transactional checking, and *semantic preservation of compatible merges*, supplied by local certificates. The result is limited to the direct-document interface. A full Denicek audit would additionally need to certify selector specificity, Copy, list ordering, reference-behaviour flags, saved checkpoints, and replay. Current-document equality alone cannot settle those obligations.

A finite checker enumerating all valid root-record documents over four keys and all distinct rename/add key triples found 456 successful certified instances, with no definedness discrepancies between the two orders; outside the common success domain, 4,392 cases gave different error labels. It also reproduced the delete/add incompatibility and the raw-lenient dangling-reference outcome. These tests validate only the *miniature* operational definitions. Their purpose is to give a reproducible checking example, not empirical assurance about the Denicek implementation.

7 Reference transport as a cocycle

Sections 5 and 6 treated a residual effect (the audit trail) that may be erased by α. Merging depends on a residual that cannot be erased: the record of *where references go*. This section makes that record explicit, uses it to test when replay is determined by the current document, and examines what a conditional edit needs from a merge algorithm. All results concern the flat calculus of §3 extended by $\mathsf{set}(k, v)$ (overwrite an existing field with a value whose reference, if any, resolves).

7.1 The transport cocycle

For a valid d and a command g with $\delta_g(d) = \mathsf{inl}(d')$, let $\kappa_d(g) : \mathrm{dom}(d) \rightharpoonup \mathrm{dom}(d')$ be the partial map tracking the identity of fields: the identity for add and set, $k \mapsto \ell$ (identity elsewhere) for $\mathsf{rename}(k, \ell)$, and undefined at k (identity elsewhere) for $\mathsf{delete}(k)$. For a successful history, compose along the history.

Proposition 7.1 (Transport is a functor)**.** *Let $D/\!\!/A^*$ be the category whose objects are valid documents and whose arrows $d \to d'$ are histories executing successfully from d to d'. Then κ is a functor from $D/\!\!/A^*$ to the category of finite sets and partial maps; that is, $\kappa_d(h; h') = \kappa_{d'}(h') \circ \kappa_d(h)$.*

Proof. By definition on histories; identities come from the empty history. □

Equivalently, κ is the residual component of a runner of the command theory into a writer monad whose monoid consists of transports. This is the writer-to-writer cocycle description; it answers the caveat of §4 that command-only runners are "just" actions, because the residual here is not erasable without losing what merging uses. A saved interaction recorded at d that targets a field k is, after a later history h, retargeted to $\kappa_d(h)(k)$; this is the content of replay through later edits.

7.2 Holonomy: the current document does not determine transport

Call a pair of histories $h_1, h_2 : d \to d'$ with $\kappa_d(h_1) \neq \kappa_d(h_2)$ a *holonomy pair*. When one exists, replay of a saved interaction recorded at d is *not* a function of the current document d'.

Proposition 7.2 (Holonomy)**.**

1. *If $h_1, h_2 : d \to d'$ consist only of renames, then $\kappa(h_2) \circ \kappa(h_1)^{-1}$ is an automorphism of d' (a permutation of keys preserving values, with references relabelled).*
2. *With* set*, or with* add *and* delete*, this fails. From $d = \{c \mapsto 7\}$,*

$$\mathsf{rename}(c, a); \mathsf{set}(a, 8); \mathsf{add}(c, 7) \quad \textit{and} \quad \mathsf{add}(a, 8)$$

both reach $d' = \{a \mapsto 8,\ c \mapsto 7\}$, with transports $c \mapsto a$ and $c \mapsto c$, and d' has no nontrivial automorphism. Also $\mathsf{delete}(c); \mathsf{add}(c, 7)$ and the empty history both reach d, with transports undefined at c and the identity.

Proof. (1) A rename acts as a relabelling isomorphism $d \cong d'$ and κ records it, so both transports are isomorphisms $d \cong d'$ and their discrepancy is an automorphism. (2) Direct computation. □

The simplest rename-only instance is $d = \{a \mapsto 7, b \mapsto 7\}$ with the empty history versus $\mathsf{rename}(a, c); \mathsf{rename}(b, a); \mathsf{rename}(c, b)$. Exhaustive enumeration over three keys (183 valid documents, histories of length at most three, all distinct transports reaching a common endpoint) gives:

Commands	holonomy pairs	definedness differs	same domain, automorphism
rename	36	0	36 of 36
rename, set	720	0	180 of 720
rename, add, delete	4,332	3,414	834 of 918
all four	45,300	38,106	1,527 of 7,194

Proposition 7.3 (Stable identities remove holonomy)**.** *Refine documents to $\widetilde{D}$: each field carries an identifier drawn from an infinite supply and never reused; references and edit targets name identifiers;* rename *changes only the displayed label. Then the transport of any successful history $\tilde{d} \to \tilde{d}'$ is the identity on the identifiers surviving in $\tilde{d}'$, so any two histories with the same endpoints have equal transport.*

Proof. No command changes an identifier, and since identifiers are never reused a surviving identifier denotes the same field at both ends. □

In cohomological language, the name-based cocycle on D becomes a coboundary once pulled back to $\widetilde{D}$: the labelling of names by identifiers is the trivialising cochain. The history state $S_{\log}$ of §4 also removes holonomy, trivially, since each history has a unique path from the root. Denicek uses unique identifiers for list items but names for record fields (§4.1 and §6 of [1]). The calculation therefore predicts replay holonomy through record-field renames and none through list operations. This is a test obligation for the implementation, not a verified claim about it.

7.3 Conditional edits: what must the merge know?

Denicek excludes conditional edits so that merging needs no document state (§4.1 of [1]). Add a conditional rename $\mathsf{crename}(x, y, v)$: rename x to y if $d(x) = v$, otherwise do nothing. Its transport depends on d. Merge single-edit branches from a common d_0 by transporting the second edit's names through the first edit's transport, with the usual syntactic conflict checks (common introduced name, common structural target, common written field, or a name sent to a deleted field). Call a merge *convergent* if both orders succeed and yield the same document *and* the same transport from d_0, so that replay also agrees. We compared five policies on all 47,052 pairs consisting of a conditional rename and any other single edit (the stateless baseline gives no unflagged failures in 45,762 pairs):

Policy	unflagged failures	flagged
Edit-only transport, assume the condition holds	11,292	10,620
Edit-only transport, assume it fails	2,436	10,620
Transport read from d_0; condition re-evaluated at replay	2,316	10,620
plus a read dependency on x (value and existence)	0	37,548
Materialise at d_0 (replace by rename or no-op)	0	432

Two separate obligations appear. Reading the transport from d_0 removes every failure caused by mis-transported references. All remaining failures arise because the condition evaluates differently after the other branch's edit. Removing them requires a read dependency covering the *existence* as well as the value of x: a check on value writes alone still leaves 636 failures, through an add that creates x. Materialising the conditional at the authoring state is convergent and flags fewer than one pair in a hundred. On the 9,504 pairs accepted by both convergent policies the results coincide. Materialisation also accepts 37,116 pairs that the dependency policy flags.

This is exactly Datnicek's expandable edit (§7.2 of [1]), now with a certificate in the flat calculus. It is a *directed* policy in the sense of §7: the condition is fixed at authoring time, which is a semantic choice, not a consequence of an equation.

7.4 What this changes

A tempting conjecture was that a state-dependent edit is mergeable without state exactly when its transport cocycle is cohomologous to a state-independent one over D. The calculation does not support it in that form. Over D holonomy is pervasive, already for stateless edits. Over $S_{\log}$ every cocycle is trivial, so the condition says nothing. What survives is a decomposition:

1. *Transport* must be evaluated at the authoring state. A merge that has the shared history has that state, so excluding conditional edits on these grounds is stronger than necessary.
2. *Re-evaluation* is a read/write dependency, including reads of absence, and belongs with the conflict analysis, not with transport.
3. *Replay invariance* is holonomy-freeness of transport over whatever state replay is defined on. It holds for stable identifiers and fails for names.

None of this is specific to Sweedler theory, but the runner reading places the transport in the residual, where Proposition 4.2-style comparisons can carry it across implementations.

8 Certified squares versus directed reconciliation

Denicek's local reconciliation is asymmetrical. When an edit is moved past an earlier edit, it can be rewritten into a *sequence* of edits. Appendix B describes three mechanisms: transporting references through structure changes; focusing an existing transformation on material newly introduced by Add, Append, or Copy; and copying an incoming operation to the destination of a prior Copy [1]. The system's conflict analysis uses conservative approximations to structure, value, and tag effects, and may flag operations that would in fact execute compatibly.

For instance, let N act on every current list item and let $A_{j,x}$ insert a new item at fresh identifier j. A focused transformation N_j gives the elementary law

$$N_* \circ A_{j,x} = N_j \circ A_{j,x} \circ N_*, \tag{10}$$

on the common domain of validity. This is ordinary naturality of insertion. Denicek's extra achievement is to *construct syntactically* the requisite N_j by refining a wildcard selector to an index.

Now suppose a saved interaction inserts j and then applies another transformation M_j. It is important not to infer arbitrary commutation:

$$N(M(x)) \quad \text{need not equal} \quad M(N(x)).$$

For example, appending `!` and appending `?` in different orders yields different strings. One merge order may deliberately place an inherited transformation before a later saved edit. The opposite order need not give the same result. This is consistent with Denicek's explicitly asymmetric merging and its separate conflict policy (§4.2 of [1]).

We therefore need to keep two kinds of claims separate: (i) *certified commuting squares*, justified by actual semantic equations on stated admissibility domains, and (ii) *directed reconciliations*, justified by a chosen conflict or priority policy. A mathematical treatment should not identify both kinds of transformation as equations.

9 Replay and the observable identity of a system

Denicek's programming-by-demonstration mechanism makes the distinction between current document and historical state essential. A saved interaction is associated with the hash of its original history checkpoint. Replay attaches the saved edits at that checkpoint and merges them through subsequent changes, so a demonstration can survive a schema refactoring (§5.2 of [1]).

Two different histories may evaluate to the same document but provide different saved checkpoints, different intermediate audit information, or different opportunities to inspect and replay past interactions. Equality of current documents is therefore not the same as equivalence of programming systems.

This suggests fixing a visible interface T_{sys} with operations such as Edit, Save, Replay, and perhaps InspectHistory. Its implementations are runners whose persistent state retains whatever historical information is necessary to realise that interface. The relevant behavioural comparison is relative to the operations one allows users or tools to perform. Omitting InspectHistory can make more implementation states equivalent than exposing raw historical structure.

An ambitious continuation would use the behaviour/comodel perspective and Garner's explicitly constructed behaviour category for accessible monads [4] to organise such distinctions.

However, it would require a precise algebraic theory for history-aware interaction; the document-edit theory alone is not sufficient.

A parallel continuation is a double-categorical treatment of *certified reconciliation squares*. Horizontal arrows could represent executable interactions; vertical comparisons could translate schema or history presentations; squares would express replay-compatible transport. This is a proposed construction. It is not yet established that Denicek's actual rules satisfy the required interchange or path-independence laws.

10 An agenda for a small joint investigation

A useful first joint effort would not attempt to categorify all of Denicek. It could proceed through the following specific deliverables.

1. **A verified core edit fragment.** Formalise exact selector matching, reference-behaviour annotations, admissibility, and the Add/Append/WrapRecord/Copy reconciliation clauses. Preserve ordered-list behaviour rather than replacing it silently by unordered maps.
2. **Local correctness certificates.** For each reconciliation clause, specify the exact observable effect to be preserved, including definedness and reference integrity. Keep intentional conflict-resolution choices distinct from semantic equalities.
3. **A second runner comparison.** Extend the direct-versus-history proof in Proposition 4.1 to a small interface containing a genuinely saved and replayable interaction. Prove preservation of replay outcomes under a representation change.
4. **Composition and counterexamples.** Test critical pairs with node creation, non-idempotent operations, Copy, and two insertions before the same successor. A reported port-level counterexample concerns insertion before the same successor (where a successor link can overwrite its predecessor), not insertion after the same predecessor (where the examined representation breaks ties by key). This needs confirmation in the source version being discussed.
5. **Generality.** Apply the resulting method to a different substrate, perhaps one with explicit schema types or a different dependency-tracking mechanism. This is the step that would demonstrate the value of Sweedler theory beyond Denicek itself.

The certified toy merge of Section 6 is a first instance of the second deliverable, but its certificates cover only flat reference records. The first two deliverables for Denicek itself are largely operational-semantics work, and must be grounded in the source implementation as well as the paper. The last three are where measuring constructions, relative behavioural equivalence, and changes of residual theory may offer genuine leverage.

11 What Sweedler theory offers—and what it has yet to earn

The current argument supports a limited but concrete claim: an operational interface can admit different stateful implementations, and runner comparisons can transport equations and safety invariants across them. A further certified-square construction shows how the same guarantees can survive compatible history merges, while conflicting postconditions must be reported or resolved by policy. Section 5 shows a corresponding audit: one implementation violates the intended invariant, while an explicit validating adapter restores a certified comparison. Propositions 4.1 and 4.2 provide a worked comparison with value-returning operations and nontrivial residual effects, though the simulation itself is elementary. This part requires neither conjectural double-category axioms nor any assumption that all merges commute.

The stronger claim remains a research hypothesis: the interaction of measuring comonads, residual theories, and behavioural comparison may provide a modular correctness discipline for evolving computational substrates. Denicek is a valuable test because it exercises precisely

the features that make the hypothesis difficult—references, partial edits, saved demonstrations, generated operations, history-sensitive replay, and conservative conflict analysis.

For a prospective collaboration, the most useful question is therefore not "Can Denicek be expressed categorically?" but:

> *Can we give programming-system builders reusable correctness principles for changing how an operational interface is implemented—while being precise about which properties of editing, replay, observation, and incremental computation must survive?*

The Sweedler formalism suggests one way to organise this effort. Whether it improves on a more conventional operational account should be judged by the comparisons and preservation theorems it ultimately makes possible.

12 Towards an auditable design methodology

The examples above suggest a more demanding ambition than verifying particular Denicek edits after the fact. A mature theory should help a designer decide *before implementation* which representations and operations make a proposed guarantee available, which combinations of requirements are incompatible, and which changes of substrate preserve an established guarantee. The strict/lenient reference audit and its guarded repair are a first, very small instance. Extending the argument to Denicek's richer capabilities would require three connected stages.

1. Validate a faithful semantic core. Begin with Denicek's actual tree representation, stable ordered identifiers, Field/Index/All/Parent selectors, reference-behaviour annotations, and partial execution rules. Compare an executable specification with the source on carefully chosen successful and failing edits; the reported lenient-upsert and ordered-insertion counterexamples are test obligations, not settled claims about every implementation. The first theorem should assert that, on an explicitly stated domain, transporting references through a structural edit preserves their resolution and definedness. Since All can denote several locations and WrapList changes selector generality, this needs a typed or relational account of reference resolution, not simply a total function on paths. The outcome should be both a positive certificate for permitted schema changes and a counterexample when its hypotheses fail.

2. Certify reconciliation, not arbitrary convergence. Add the Appendix B rules for focusing earlier transformations on newly created nodes and propagating edits through Copy; retain the exact ordering semantics of Append. Each generated square should carry its admissibility predicate, the guarantee it preserves (reference integrity, definedness, selected effects), and a proof or checkable certificate. Noncommuting edits and incompatible postconditions must instead receive explicit *directed* conflict resolutions. A composition theorem would then allow locally certified squares to be pasted into safe history transformations. The design output would distinguish (i) a merge safely accepted, (ii) a genuine conflict requiring policy, and (iii) an implementation defect for which a different guard or data representation supplies a certified repair. Our flat-record theorem establishes only the prototype of this result.

3. Extend the guarantee to saved interactions and alternative substrates. In Denicek, a programme recorded at a historical checkpoint is replayed by transporting its edits through later changes. We therefore need a history-aware interface with Save and Replay, an explicit correspondence of checkpoints under reconciliation, and a law expressing preservation of replay behaviour, schematically

$$f^{+}(\mathrm{Replay}(H, u)) \;\simeq\; \mathrm{Replay}(fH, f_{*}u).$$

This equivalence must be defined relative to specified observations and future interactions; equality of current documents alone is insufficient. The critical Sweedler test would then compare two genuinely different residual implementations—for example, a history-based substrate and a typed or schema-indexed variant—and transport certified safety and replay properties along a runner simulation. That, rather than the mere existence of a runner presentation, would be evidence of added mathematical leverage.

At each stage the proposed *audit report* has the same form: state the desired observable guarantee; exhibit either a proof or a minimal counterexample for the existing design; identify a change of interface, guard, transaction boundary, or residual representation that makes the guarantee attainable; and certify that the improvement survives the relevant implementation comparisons. One need not assume that every desirable property is compatible with Denicek's permissive editing experience. Indeed, a useful design tool should make such trade-offs explicit.

A first collaboration could stop after the reference/selector core and one nontrivial certified reconciliation rule. Copy, ordering and history-sensitive replay would form subsequent extensions, rather than prerequisites for an initial useful result. The eventual question is whether the resulting proof obligations and preservation theorems can be *reused* when designing a second computational substrate. That is the point at which Sweedler theory would have earned the description of an auditing *and* design methodology.

Questions for discussion

Three questions would particularly benefit from experience with Denicek's implementation. First, which visible guarantees should count as invariant when the substrate changes: final document equality, reference validity, preservation of saved demonstrations, or agreement of dependency-driven invalidation? The answer determines the operational theory we ought to specify, and avoids asking for a stronger equivalence than the user interface actually needs.

Second, could the reference-update and node-creation rules in Appendix B be given a small executable specification that separates admissibility, generated edit provenance, and conflict-resolution choices? A restricted correctness theorem would already be useful if its conditions could be checked locally; a counterexample to a stronger proposed coherence law would be equally informative.

Third, what would be a worthwhile *second* substrate against which to test this semantics? A typed or schema-indexed document model, a different ordered-list implementation, or a system with explicit provenance would let us assess whether runner comparisons provide reusable correctness proofs rather than merely new terminology for Denicek's existing rules.

A productive first conversation need not resolve these questions. It would be enough to identify one interface, one implementation change, and one guarantee whose preservation is worth proving.

References

[1] T. Petricek and J. Edwards. *Denicek: Computational Substrate for Document-Oriented End-User Programming*. UIST 2025. DOI: 10.1145/3746059.3747646.

[2] J. Jakubovic, J. Edwards, and T. Petricek. *Technical Dimensions of Programming Systems*. The Art, Science, and Engineering of Programming 7 (2023), Article 13. DOI link.

[3] D. McDermott, E. Rivas, and T. Uustalu. *Sweedler Theory of Monads*. In FoSSaCS 2022, LNCS 13242, pp. 428–448. DOI link.

[4] R. Garner. *The costructure–cosemantics adjunction for comodels for computational effects*. Mathematical Structures in Computer Science 32(4) (2022), 374–419. DOI link.

[5] S. Katsumata, E. Rivas, and T. Uustalu. *Interaction Laws of Monads and Comonads*. LICS 2020, pp. 604–618. DOI link.