1 Algebraic Foundations of Staged Durable Execution

1.1 Abstract

We develop an algebraic account of durable workflow execution organized as a staged reduction: commutative fact accumulation, idempotent failure closure, and deterministic state classification. The core object is not a workflow program trace but a topology-indexed graph state together with a validity predicate for normalized, closed states. We separate three layers of obligation: algebraic facts about accumulation, closure, and classification; structural recovery theorems for fixed-topology execution; and external assumptions about worker I/O reproducibility. The result is a sharper statement of what the runtime proves, what it merely assumes, and what is deferred to mechanization. We also characterize the extension boundary: which changes preserve the staged reduction and which require a different machine. Topology-changing graph rewriting is treated as one such boundary and is excluded from the core theory of this paper.

1.2 1. Core Model

1.2.1 1.1 Graphs, States, and Validity

Let $G = (V, E)$ be a finite DAG. A graph state consists of:

- a total status map $status : V \to NodeStatus$
- a partial output map $output : V \rightharpoonup Value$

For the fixed-topology core:

$$NodeStatus = \text{Pending} \mid \text{Running} \mid \text{Completed} \mid \text{Failed} \mid \text{Skipped} \mid \text{Waiting}\ (signal)$$

We distinguish three kinds of states:

- **mid-accumulation states**, which may still contain Running
- **normalized states**, after `resetRunningToPending`
- **closed states**, after `propagateFailure`

The main validity predicate is defined only on normalized, closed states.

Definition 1 (`wellFormedGraphState`). $\text{wellFormedGraphState}\ (G, s)$ holds iff:

1. $\text{dom}\ (status) = V$
2. $\text{dom}\ (output) \subseteq V$
3. outputs appear exactly on statuses that semantically carry outputs
4. no node is Running or Interrupted
5. every Pending or Waiting descendant of a Failed node is itself Failed
6. $\text{readyNodes}\ (G, s)$ is extensionally equal to the readiness predicate: nodes that are Pending and whose predecessors are all Completed, Skipped, or Rewritten

This definition is intentionally relative to fixed topology. Once topology changes, the validity contract must be enriched with materialization and identity invariants.

1.2.2 1.2 The Staged Machine

The fixed-topology runtime factors as:

$$\text{reduce}\ (G, R, s) = \text{classify}\ (G, \text{close}\ (G, \text{accumulate}\ (R, s)))$$

Where:

- `accumulate = foldl (flip applyNodeFact)`
- `close = propagateFailure`
- `classify = classifyGraphState`

The execution model uses barriered frontier waves, but the algebraic core only sees a set of node-local facts and a topology-indexed state.

1.3 2. Algebraic Structure

1.3.1 2.1 Accumulation as a Commutative Action

Let `F` be the free commutative monoid of `NodeResult` values with distinct `NodeIds`. Distinctness is justified by the frontier antichain property.

`applyNodeFact` induces a monoid action:

$$\text{act} : F \times GraphState \to GraphState$$

$$\text{act } (S, s) = \text{foldl } (\text{flip } applyNodeFact, s, S)$$

The load-bearing property is not a deep join law on shared state. It is the more modest disjoint-key commutativity of point updates.

Lemma 1 (disjoint-key commutativity). If $r_1.nid \neq r_2.nid$, then:

2 $$ applyNodeFact\ r_1\ (applyNodeFact\ r_2\ s)

applyNodeFact\ r_2\ (applyNodeFact\ r_1\ s) $$

This is the precise theorem. References to LVars, CRDTs, and related work should be read as analogies in design style, not as direct reuse of their stronger shared-key commutativity results.

2.0.1 2.2 Failure Closure

`propagateFailure` is treated as a closure operator on the preorder "has at least as many failed consequences as":

- extensive
- monotone
- idempotent

These laws are what make partial persistence structurally safe. In this draft they are stated as semantic obligations of the implementation and deferred to the Lean mechanization plan.

2.0.2 2.3 Classification

`classifyGraphState` is a deterministic decision procedure on normalized, closed states:

- `Progressing` if some node is ready
- `Settled` if all nodes are terminal
- `Suspended` if some node is waiting and no node is ready
- `Stuck` otherwise

The case split is by construction once readiness, terminality, and waiting are fixed.

2.0.3 2.4 Determinism of the Staged Reduction

Theorem 2 (staged reduction determinism). For any permutation π of a frontier result set:

$$\text{reduce } (G, \pi(results), s) = \text{reduce } (G, results, s)$$

The theorem is conditional on the identity of `NodeResult` values. It is a theorem about reduction order, not about worker I/O behavior.

2.1 3. Structural Recovery Safety

2.1.1 3.1 Theorem Statement

Let $S = S_1 \cup S_2$ be a frontier result set, where S_1 is the subset whose local facts were durably persisted before a crash and S_2 is the subset that must be replayed. Define:

$$s_{persisted} = \text{accumulate } (S_1, s_0)$$

$$s_{recovered} = \text{close } (G, \text{resetRunningToPending } (s_{persisted}))$$

Theorem 3 (structural recovery safety). $s_{recovered}$ satisfies $\text{wellFormedGraphState } (G, s_{recovered})$.

Moreover:

- $\text{readyNodes } (G, s_{recovered})$ is a correct resume frontier
- re-closing after replayed facts are accumulated preserves already derived failures

2.1.2 3.2 What the Theorem Does and Does Not Say

The theorem guarantees:

- topology-indexed domain consistency
- removal of transient `Running`
- closure completeness
- frontier correctness
- schedulable recovery from any persisted accumulation prefix

The theorem does **not** guarantee:

- identical business outputs after replay
- identical worker exceptions or timing
- end-to-end correctness of the whole workflow system

Those stronger properties require assumptions or proof obligations outside the staged reduction itself.

2.1.3 3.3 Dependency Structure

The theorem depends on four obligations:

1. `applyNodeFact` preserves topology-indexed domains
2. `resetRunningToPending` preserves domains and removes all `Running`
3. `propagateFailure` is a closure operator and preserves domain consistency
4. `readyNodes` is sound and complete for the readiness predicate

Only after these are stated explicitly does the recovery theorem stop being circular.

2.1.4 3.4 Proof Status

Claim	Status in this draft	Intended home
Frontier antichain	sketchable from DAG readiness	Paper 1 + Paper 2
Disjoint-key commutativity	direct proof	Paper 1 + Paper 2
Closure extensiveness / monotonicity / idempotence	stated as obligations, not fully proved here	Lean mechanization plan
Structural recovery safety	stated sharply, proof outline only	Lean mechanization plan + Paper 2
Outcome reproducibility under replay	external assumption	outside the algebra

The point of the table is discipline: this paper should never silently upgrade "tested in implementation" into "proved in theory."

2.2 4. The Forward Fragment and the Reset Boundary

2.2.1 4.1 Forward Fragment

The forward statuses form the local monotone fragment:

$$\text{Pending} \leq \text{Running} \leq \text{Completed}$$
$$\text{Pending} \leq \text{Running} \leq \text{Failed}$$
$$\text{Pending} \leq \text{Running} \leq \text{Skipped}$$
$$\text{Pending} \leq \text{Running} \leq \text{Waiting } (signal)$$

Forward outcomes move upward in this partial order. This fragment supports the cleanest algebraic reading.

2.2.2 4.2 Reset Boundary

Cancellation, shutdown, and crash normalization are explicitly non-monotone. They do not invalidate the staged reduction, but they do invalidate any claim that the runtime is globally monotone.

This separation should remain explicit:

- the forward fragment supports the order-theoretic story
- reset behavior is handled constitutionally by keeping resets node-local and coordinator-controlled

---

2.3 5. Extension Boundary

The staged reduction has three load-bearing invariants:

- **I.** accumulation commutativity for frontier facts
- **II.** closure idempotence
- **III.** frontier antichain / readiness soundness

2.3.1 5.1 Extensions That Preserve the Core

Extension	I	II	III	Reason
new node-local forward outcome	yes	yes	yes	still a point update
richer classification output	yes	yes	yes	classification is observational
additional closure commuting with failure closure	yes	conditionally	yes	composition of commuting closures
edge annotations preserving DAG readiness semantics	yes	yes	conditionally	if readiness stays predecessor-based

2.3.2 5.2 Extensions That Cross the Boundary

Extension	Broken invariant	Why
inter-node mutation in accumulation	I	facts no longer commute by disjoint key
compensating or oscillating propagation	II	closure may stop being idempotent
topology mutation during accumulation	III	frontier and closure are computed over moving topology
shared-key accumulation without a merge algebra	I	update order becomes observable

Topology-changing execution is therefore not "just another extension." It requires a different machine and a different safety theorem.

2.4 6. Relation to Other Models

2.4.1 6.1 CALM

The CALM theorem remains a useful interpretation:

- accumulation is coordination-free
- classification is a threshold observation
- frontier barriers are the coordination boundary

This is an architectural reading, not a formal reduction of the runtime to CALM.

2.4.2 6.2 CKA and BSP

The runtime resembles a BSP-style fragment of concurrent Kleene algebra:

$$(\text{wave}_1 \text{ workers}); \text{close}; \text{classify}; (\text{wave}_2 \text{ workers}); \text{close}; \text{classify}; \cdots$$

This is useful for intuition about barriered parallel composition, but we do not claim a full CKA embedding here.

2.4.3 6.3 Actor Confluence Work

Henrio et al.'s layered confluence results for actors are relevant because our system is also a worker-coordinator architecture. The key difference is that our workers are constitutionally restricted to node-local facts, which gives a narrower and stronger commutativity condition than arbitrary actor messaging.

2.5 7. Open Problems

1. **Mechanize the closure laws.** This is the main proof debt and the central purpose of the Lean mechanization plan.
2. **Characterize the closed-state subspace.** The closure operator suggests a useful sub-poset of normalized states, but its exact algebraic structure is still only partly understood.
3. **Weaken replay assumptions.** Structural safety needs less than outcome identity. The weakest useful replay-compatibility condition is still open.
4. **Separate theory for topology-changing execution.** Rewrite-capable execution requires a materialization boundary, durable identity scheme, and different recovery theorem. That is the subject of the rewrite materialization and recovery plan.

2.6 8. Takeaway

The point of the algebraic foundations paper is not to inflate an implementation sketch into a fake theorem. It is to isolate the exact mathematical kernel of the fixed-topology runtime:

- node-local facts commute by disjoint key
- failure closure is the load-bearing closure operator
- classification is a deterministic read on normalized, closed states
- structural recovery safety follows only after validity is defined explicitly

Everything stronger must either be proved elsewhere, or named honestly as an assumption.

2.7 References

1. Kuper, L., Newton, R. (2013). *LVars: Lattice-based Data Structures for Deterministic Parallelism.* FHPC 2013.
2. Shapiro, M. et al. (2011). *Conflict-free Replicated Data Types.* SSS 2011.
3. Conway, N. et al. (2012). *Logic and Lattices for Distributed Programming.* SoCC 2012.

4. Hoare, C. A. R., Möller, B., Struth, G., Wehrman, I. (2009). *Foundations of Concurrent Kleene Algebra*. RelMiCS 2009.
5. Valiant, L. G. (1990). *A Bridging Model for Parallel Computation*. Communications of the ACM 33(8):103–111.
6. Henrio, L. et al. (2026). *Layers of Confluence for Actors*. CPP 2026.