Failure, Supervision, and Durable Execution

Part IV of *An Algebra of Agents*

Matthew Long
The YonedaAI Collaboration, YonedaAI Research Collective
Chicago, IL
matthew@yonedaai.com

September 2026

Abstract

Agent runtimes crash, retry, checkpoint and replay, and their behaviour is usually described by an implementation rather than by an algebra. This Part supplies the execution layer for the agent algebra of the preceding Parts. An execution model is a class of scheduler agents together with a fail-stop failure model in which a crash is an input that resets a component to a designated state, and fairness is a predicate on scheduler traces. Retry with backoff and timeout are terms over the recovery, race and delay operators of Part II. Supervisors with the three Erlang strategies and a restart budget, checkpointing, choice logs and deterministic replay are constructions on agents, and workflows are Kahn networks over the channels of Part III. Replay against a complete log of inputs and effect resolutions turns an agent over any of the three effect monads into a deterministic agent whose trajectory is the logged one, and replay of a modified agent is correct exactly when the log stays compatible with it. Supervision with a checkpoint at every tick refines the crash-free run in both directions once crash and restart markers are erased, and a counter child shows that supervision without checkpointing does not. Retry against an environment that deduplicates on a stated key gives effectively-once effects, and a counting environment shows duplicates. Deterministic acyclic workflows are schedule-independent under fairness, and the dependency order of every product and sequence term is series-parallel, so the N-shaped workflow is not such a term.

1 Introduction

An agent placed under a runtime is a different object from the agent. The runtime supplies a scheduler that decides which component steps, a failure model that decides which component is reset, a persistence layer that decides what survives a reset, and a log that decides what can be reproduced. Parts I through III fix the object and the wiring on the assumption that none of these intervene. In deployment they intervene constantly: language-model calls time out, worker processes are killed by the operating system, workflows resume on a different machine hours after they began, and a request is delivered twice because the first acknowledgement was lost. This Part puts those events inside the semantics and determines which laws of Parts I through III survive them.

This Part is written under the claim that governs the series.

> Part I fixes the object: discrete time, an effect monad T drawn from the list Id, P, D, and agents as effectful Mealy coalgebras with typed termination and failure as a terminal outcome. Parts II to IV introduce the operators under additional assumptions stated in

each Part and prove, for each operator, that it is well-defined on agents, which relation it respects, and which laws hold at that relation: Part II for ;, $\oplus$, $\otimes$, $\triangleright$ and μ; Part III for Tr, channels, $\|$ and Gate_H; Part IV for Sup, Persist and Replay; Part V introduces no operator and interprets policies, plans, options, shields and controllers as terms of Parts I to IV, proving properties of their closed loops. For each of sixteen fields the series gives an explicit translation of one named mechanism into a term of the algebra together with an adequacy result for that translation, which is a soundness theorem, a one-way simulation, or a proved limitation; no field is claimed to reduce to the algebra.

The clause about Sup, Persist and Replay holds under four assumptions that Parts I through III do not need: crashes occur only at tick boundaries, a crash tick is isolated so that components other than the ones being reset receive the idle input, the children of a supervisor are deterministic and do not fail internally, and the environment is quiescent on idle output. Each result below consumes some of the four, at the relation and over the effect monads that Table 1 records.

1.1 Contributions

A crash is an input drawn from a reserved control alphabet: it stops the component that receives it and emits a marker in place of an ordinary output. Fail-stop behaviour in the sense of Schlichting and Schneider then follows from the typing of the alphabet rather than from an assumption about hardware, because a struck component has no way to emit an ordinary output on the tick it is struck.

Supervision, checkpointing and the choice log are built over that failure model, and each is a construction on agents in the sense of Part I rather than a new primitive in the signature. The supervisor Sup_σ owns a vector of children, a restart function determined by a strategy, and a budget that fails the supervisor when restarts arrive faster than the budget allows. Checkpointing $\mathsf{Persist}_C$ extends the state space of an agent by a saved-state register and moves the restart target to the last saved state. The choice log records the choice component of Part I's policy and update factorization, which is where an effect monad meets the engineering requirement that a stochastic run be reproducible.

A supervised system emits crash and restart markers that an unsupervised system does not, so the two have disjoint trace sets. Erasing the markers makes them comparable, which is the reason Part I's observation policy provides for erasure. With every child checkpointed at every tick, the erased traces of the supervised system and the traces of the crash-free system agree under stuttering closure (Theorem 8.3). Without the checkpoint the supervised system admits a trace that no crash-free run produces (Theorem 8.5). The three Erlang restart strategies then have the same erased behaviour (Theorem 8.4), so a choice among them fixes the cost of a restart and not what survives one.

At-least-once delivery with an environment that deduplicates on a stated key is the usual construction behind what industrial systems call exactly-once semantics, and it is not exactly once in any literal sense: the duplicate requests are emitted and are visible on the wire. Theorem 9.2 states the property that does hold, a refinement of the environment's observable effects rather than of the full trace, and Theorem 9.3 gives the counting environment for which even that fails. The two properties have different subjects, which is why the result is named effectively once.

Workflow engines present a directed acyclic graph of tasks and promise that the outcome does not depend on how the engine interleaved them. Theorem 10.3 proves the promise for the fragment where it holds, deterministic nodes with blocking reads over unbounded channels on an acyclic graph, by a topological induction rather than by an appeal to a fixed-point theorem. Theorem 10.7

Table 1: The results of this Part, with the relation at which each is stated, the effect monads it covers and the hypotheses beyond those of Parts I to III.

Result	Kind	Relation	Instances and added hypotheses
Theorem 4.2	Proposition	halting time	Id, P, D; none
Theorem 5.3	Proposition	set inclusion	Id, P, D; none
Theorem 5.4	Proposition	one-way simulation	Id; fixed start order
Theorem 6.2	Proposition	$\sim_{\mathrm{er}}$	Id, P, D; synchronous cuts and synchronous crash schedules
Theorem 7.3	Theorem	equality of outputs	Id, P, D; a complete log of inputs and resolutions
Theorem 8.3	Theorem	$\sqsubseteq$ over $\sim_{\mathrm{er}}$	Id children that do not fail; a checkpoint at every tick; an isolating and bounded failure model; a replay-idempotent environment
Theorem 8.5	Counterexample	$\sqsubseteq$	Id; no checkpointing
Theorem 9.2	Theorem	$\sqsubseteq$ over $\sim_{\mathrm{er}}$	Id; crash points at tick boundaries; an environment idempotent on a stated key; the effect projection
Theorem 9.3	Counterexample	$\sqsubseteq$	Id; a counting environment
Theorem 10.3	Theorem	equality of outputs	Id; the Kahn fragment and a fair scheduler
Theorem 10.7	Proposition	graph property	Id; duration-flexible leaves
Theorem 10.9	Counterexample	$\sim$	Id; early start and duration-flexible leaves

and Theorem 10.9 then measure the distance between such graphs and the terms of Part II: the dependency order of a term built from $\otimes$ and ; is series-parallel, and the four-node N-shaped graph is a workflow whose dependency order is not, so a workflow graph expresses dependencies that the product and sequence operators cannot.

1.2 Relation to the rest of the series

The log is an instance of Part III's unbounded channel with a read cursor, the scheduler is Part III's scheduler agent with a crash port added, and the closed loop is Part I's with its output port relabelled by a function of the closed state, so no definition here reopens a definition of Parts I to III. Retry and timeout are terms over Part II's recovery, race and delay, and inherit their well-definedness from Part II's congruence theorem.

A language-model tool loop is an agent over the distribution monad in Part V, and the replay theorem below makes such a loop reproducible once its sampled tokens are recorded as effect resolutions. A human approval gate and a safety shield are the same guarded choice in Part V, and neither is a supervisor: a gate acts before an output, a supervisor after an outcome. Neither sentence carries a label and neither imports anything from a later Part.

2 Preliminaries

2.1 Imported objects

Every object of Table 2 enters this Part unchanged, under the label its owning Part gave it, abbreviated in the table to the name after the last colon.

An agent of Part I is a tuple $A = (S, \mathsf{init}, \mathsf{step}, \mathsf{halt})$ of type $X \Rightarrow Y$ over an interface (I, Σ) of pointed alphabets, with $\mathsf{init} : X \to S$, $\mathsf{step} : S \times I \to T(\Sigma \times S)$ and $\mathsf{halt} : S \to Y + E + 1$ for a finite failure alphabet E. The three outcomes are halted with a value, failed with a reason, and running. The effect monad T is one of Id, P and D, never a combination. The trace semantics is $[\![A]\!] : X \times I^* \to$

Table 2: The objects and results this Part imports, by owning Part. Full labels are formed by prefixing the kind and the owning Part's slug, so that *interface* in the first column is `def:agent-object:interface`.

Owner	Imported names
Part I, definitions	interface, effect-monad, lifting, agent, transition, policy-update, trace, trace-equiv, accepted-trace, stutter, refinement, bisim, erase, observation, invariant, reachable, environment, closed-loop
Part I, results	bisim-implies-trace, greatest-invariant, refinement-preorder, and the coinduction, homomorphism and lifting-bind lemmas
Part II, definitions	skip, seq, category, zero, fail, delay, guarded-choice, tensor, race, recover, context, guarded, mu
Part II, results	sequential-category, monoidal, congruence, unique-solutions, recover-laws, retry-guardedness, exchange-fails
Part III, definitions	process, quiescent, register, trace-op, channel, link, scheduler-agent, async-par, fragment
Part III, results	fifo-determinate, the two-node base case of Theorem 10.3, and the wiring congruence lemma

$T(\Sigma^* \times (Y + E + 1))$, and the step factors as $\mathsf{step}(s, i) = \mathbf{do}\ (\sigma, k) \leftarrow \pi(s, i);\ \eta(\sigma, \delta(s, i, \sigma, k))$ with k the choice made by the effect. That choice component is what the log of Section 7 records.

Two conventions of Part I recur throughout. A halted or failed state stutters: its step emits τ and returns the same state for every input. Halting and failing are evaluated between ticks and consume no tick, so a sequential composition passes control at a tick boundary and the composite's outcome after n ticks is read from the state that the handoff has already produced.

The five relations of Part I are used exactly as Part I defines them. Bisimilarity $\sim$ is defined by relation lifting of T and preserves the outcome map. Trace equivalence $\simeq_{\mathrm{tr}}$ is equality of $[\![\cdot]\!]$, and accepted-trace equivalence $\simeq_{\mathrm{acc}}$ compares only the runs whose outcome is a value. Refinement $\sqsubseteq$ is inclusion of stutter-closed trace sets, written with the refined agent on the right, so that $A \sqsubseteq B$ reads that B allows every stutter-closed trace of A. Marker erasure gives $\sim_{\mathrm{er}}$, and Theorem 2.1 fixes which outputs it erases.

2.2 The execution interface

Crashes and restarts have to be visible somewhere, because a semantics in which they are invisible cannot state that erasing them changes the comparison. Part I reserves a subset of the output alphabet for markers and defines $\sim_{\mathrm{er}}$ as bisimilarity after every marker has been replaced by τ. Theorem 2.1 fixes that subset concretely.

Definition 2.1 (Execution interface and markers)**.** Let N be a finite set of component names. The *marker set* over N is

$$M_N \;=\; \{\mathsf{crash}\} \;\cup\; \{\mathsf{restart}_J : \emptyset \neq J \subseteq N\},$$

a set of fresh symbols disjoint from Σ. For a base interface (I, Σ) the *execution interface* over N is

$$(I^\gamma, \Sigma^\gamma) \;=\; \big(I \times 2^N,\ \Sigma \uplus M_N\big),$$

pointed at $(\varepsilon, \emptyset)$ and τ. The second input component is the *crash port*: the set of components the failure model strikes on this tick. Markers are the reserved subset $M_N \subseteq \Sigma^\gamma$ of Part I's observation policy, and erasure sends each of them to τ and fixes everything else.

A marker occupies a whole tick, so a component that emits one produces no ordinary output on that tick. That is the modelling counterpart of fail-stop behaviour, and it is what makes Theorem 8.2 true. Part I's reserved subset also allows checkpoint and replay markers; the constructions below do not emit them, because a checkpoint is taken on a tick that also carries ordinary work and a replayed tick reproduces the output it is replaying. Both events are therefore silent here, and an implementation that wants them observable pays a tick for each.

The crash port is not an input the environment supplies. In a γ-execution it is driven by a source agent internal to the execution model and then hidden, in the sense of Part I's observation policy, so that the observable input alphabet of a running system is the base alphabet I and a crash tick carries the idle input. Theorem 3.2 makes the source explicit.

2.3 Recording the start value

Restart and retry both have to re-enter an agent where it began, and Part II's recovery operator starts its right branch at the failure reason rather than at the start value. The gap is closed once, here, by a lifting that keeps the start value in the state.

Definition 2.2 (Task lifting)**.** Let $A = (S, \mathsf{init}, \mathsf{step}, \mathsf{halt}) : X \Rightarrow Y$ have failure alphabet E. The *task lifting* $\widehat{A} : X \Rightarrow Y$ has failure alphabet $E \times X$, state space $S \times X$, and

$$\widehat{\mathsf{init}}(x) = (\mathsf{init}(x), x), \qquad \widehat{\mathsf{step}}((s,x),i) = T\big((\sigma, s') \mapsto (\sigma, (s', x))\big)\big(\mathsf{step}(s,i)\big),$$

with $\widehat{\mathsf{halt}}(s,x) = \mathsf{inl}(y)$ when $\mathsf{halt}(s) = \mathsf{inl}(y)$, $\widehat{\mathsf{halt}}(s,x) = \mathsf{inm}(e,x)$ when $\mathsf{halt}(s) = \mathsf{inm}(e)$, and $\widehat{\mathsf{halt}}(s,x) = \mathsf{inr}(*)$ otherwise. The *start reset* of $\widehat{A}$ is $\rho_{\mathsf{init}}(s,x) = (\mathsf{init}(x), x)$.

Lemma 2.3 (Task transparency)**.** *Let $p : E \times X \to E$ be the first projection and let p_* relabel failure reasons along p. Then $p_*\widehat{A} \sim A$ for every $T \in \{\mathrm{Id}, \mathrm{P}, \mathrm{D}\}$, and $\widehat{A}$ and A have the same accepted traces.*

Proof. Let $f : S \times X \to S$ be the first projection. Then $f(\widehat{\mathsf{init}}(x)) = \mathsf{init}(x)$; the outcome maps agree after relabelling along p, since $p(e,x) = e$; and $\widehat{\mathsf{step}}((s,x),i)$ is by construction the image of $\mathsf{step}(s,i)$ under $\mathrm{id}_\Sigma \times (s' \mapsto (s', x))$, so $\mathsf{step}(f(s,x),i) = T(\mathrm{id}_\Sigma \times f)(\widehat{\mathsf{step}}((s,x),i))$. The homomorphism lemma of Part I applies to f and gives $p_*\widehat{A} \sim A$. Accepted traces are unchanged because the first component of $\widehat{\mathsf{halt}}$ agrees with halt on halted states. □

From here on $\widehat{A}$ is written A whenever the start value plays no role, which is everywhere except in the definitions of restart and retry.

2.4 Observed closed systems

Part I's closed loop $\mathsf{Run}(A, E)$ has the unit output alphabet, so its traces carry no information beyond the halting outcome. Two of the results below compare closed systems, so the closed loop needs an output port. It is added by relabelling, not by a new wiring operator, and ruling 2 of the series prospectus, which forbids a third feedback construction, is respected.

Definition 2.4 (Effect projection and observed closed system)**.** Let A be an agent over $(I^\gamma, \Sigma^\gamma)$ and E an environment co-agent over $(\Sigma^\gamma, I^\gamma)$ with state space S_E. An *effect projection* is a partial function $\nu : S_E \times S_E \rightharpoonup V$ into a pointed set (V, τ_V) with $\nu(q,q) = \tau_V$. The *observed closed system* $\mathsf{Obs}_\nu(A, E)$ is Part I's $\mathsf{Run}(A, E)$ with its output port relabelled from the unit to $\Sigma^\gamma \times V$, emitting on tick t the pair $(\sigma_t, \nu(q_{t-1}, q_t))$ of the agent's output and the environment's observable effect. Writing $\mathsf{Obs}_E(A, E)$ for the same system with the first component dropped gives the *effect trace* of the run.

The condition $\nu(q, q) = \tau_V$ is what makes a tick on which the environment does not move an idle position for Part I's stutter closure, and it is the reason Theorem 9.2 can be stated as a refinement at all.

3 The execution model

An execution model fixes who steps and who is reset. Part III's scheduler agent settles the first; the failure model below settles the second and is new to this Part.

3.1 Fail-stop failure

Definition 3.1 (Crash liftings and failure model)**.** Let A be an agent over (I, Σ) with state space S. The *fail-stop lifting* $A^{\perp}$ is the agent over the execution interface with component set $\{A\}$, state space $S + \{\perp\}$, the outcome map of A on S and $\mathsf{halt}(\perp) = \mathsf{inm}(\mathsf{crashed})$, and

$$\mathsf{step}^{\perp}(s, (i, \kappa)) = \begin{cases} \eta(\mathsf{crash},\ \perp) & \text{if } \kappa \neq \emptyset \text{ and } \mathsf{halt}(s) = \mathsf{inr}(*), \\ \mathsf{step}(s, i) & \text{if } \kappa = \emptyset \text{ and } \mathsf{halt}(s) = \mathsf{inr}(*), \\ \eta(\tau, s) & \text{otherwise.} \end{cases}$$

For a *reset map* $\rho : S \to S$ the *reset lifting* A^{ρ} is the same agent with $\rho(s)$ in place of $\perp$ in the first clause, so that a crash returns A to $\rho(s)$ instead of failing.

A *failure model* on a component set N is a set F of *crash schedules* $\varphi : \mathbb{N}_{>0} \to 2^N$ that contains the empty schedule and is closed under prefixes. It is $(\max_r, \max_t)$-*bounded* when every $\varphi \in F$ and every n satisfy $\#\{t \in (n - \max_t,\ n] : \varphi(t) \neq \emptyset\} \leq \max_r$. It is *isolating* when, on a tick with $\varphi(t) \neq \emptyset$, every component outside $\varphi(t)$ receives the idle input.

The two liftings separate the primitive from the policy. The fail-stop lifting is the primitive: a struck component stops, and the decision about what happens next belongs to whatever encloses it, which is the retry wrapper of Section 4 or the supervisor of Section 5. The reset lifting is what an enclosing construct realizes once it has chosen a restart target, and Theorem 6.1 is the case where that target is a saved checkpoint.

A crash consumes the tick and produces a marker rather than an ordinary output. That is the modelling counterpart of the fail-stop condition of Schlichting and Schneider [13]: the component stops rather than performing an incorrect transition, and no observer sees a partial result. A crash delivered to a halted or a failed state does nothing, so an agent that has already produced its value cannot be made to unproduce it. The crash arrives at a tick boundary and never inside a step, which is Part I's observation policy and which makes the whole tick, rather than a fragment of it, the unit of loss.

Dropping isolation defeats supervision transparency even under checkpointing at every tick. A struck component loses a tick that its peers do not lose, so the environment advances past a request whose answer the lost tick was to consume, and the checkpoint restores the component's state without restoring that input. Theorem 8.6 gives a two-tick witness and the two designs that avoid it.

Definition 3.2 (Execution model and fairness)**.** An *execution model* is a pair $\gamma = (\Gamma, F)$ where Γ is a nonempty class of scheduler agents over the execution interface, each an instance of Part III's scheduler agent, and F is a failure model on the same component set. For $\varphi \in F$ the *crash source* Φ_{φ} is the agent over the interface $(1, 2^N)$ that emits $\varphi(t)$ at tick t and never halts. A γ-*execution* of

a system S is a run of S under some $\mathsf{Sched} \in \Gamma$ with the crash port linked to the output of some Φ_φ and then hidden. Hiding is Part I's observation policy applied to a linked port, so the observable interface of a γ-execution is the base interface of S, and a run driven by φ is written $S|_\varphi$.

Definition 3.3 (Fair scheduler trace)**.** A component is *enabled* at a tick when it has an available step, which for the channel-connected components of Part III means that every port it blocks on is nonempty. A scheduler trace is *fair* when every component enabled at infinitely many ticks is scheduled at infinitely many ticks. An execution model is fair when every trace of every scheduler in Γ is fair.

Fairness and boundedness are assumptions and not consequences. Fischer, Lynch and Paterson [6] prove, in their Theorem 1, that no deterministic protocol solves consensus in an asynchronous message-passing system in which one process may crash; nothing in Theorem 3.2 evades that, and no result in this Part asserts termination of an agreement task. What the assumptions buy is that a component that wants to run eventually runs and that a supervisor that is willing to restart eventually stops being asked to.

3.2 An execution model as an agent

The class Γ is a class of agents, so an execution model is itself an object of the algebra and can be composed, refined and compared like any other. The convenient consequence is that a statement of the form "the same system behaves the same under every scheduler in Γ" is a statement about a family of closed systems and is proved by exhibiting one relation, which is what the proof of Theorem 10.3 does.

Example 3.4 (A small execution model)**.** Take $N = \{1, 2\}$ and let Γ contain the round-robin scheduler, which emits the component index $t \bmod 2$ at tick t, and the class of all schedulers over P that emit a nonempty subset of N and are fair. Take F to be the set of crash schedules that are $(2, 10)$-bounded and isolating. Then $\gamma = (\Gamma, F)$ admits, among others, the crash-free execution, the execution with a single crash of component 1 at tick 7, and no execution with three crashes inside any ten consecutive ticks. A supervisor with budget $(2, 10)$ over these two components never reaches its dead state under γ, by Theorem 5.3, and that is exactly the condition Theorem 8.3 needs.

An execution model is a hypothesis about the deployment, not a property of the program. A runtime that restarts a worker without pausing its peers realizes a non-isolating F; a queue-based runtime that redelivers unacknowledged work realizes an isolating one at the level of channel contents. Which of the theorems below apply to a given deployment is therefore a question about the runtime, and the point of naming γ is to make that question answerable rather than implicit.

3.3 Symbols introduced here

Table 3 lists the symbols this Part adds to the shared notation of the series. Every one is a construction on agents of Part I, and none extends the signature of Part II or the wiring of Part III.

4 Retry, timeout and backoff

Timeout and retry are terms over Part II's operators rather than new agents, so their properties follow from Part II's congruence theorem. Both terms use one piece of notation: for a partial map $f : Z \rightharpoonup W$, write $\ulcorner f \urcorner : Z \Rightarrow W$ for the agent with state space W, initializer f, idle step and $\mathsf{halt} = \mathsf{inl}$, which halts at once with $f(z)$ and consumes no tick. It is an agent wherever f is defined, and it changes no timing.

Table 3: Symbols owned by this Part. Each denotes a construction on agents, defined in the section named in the last column.

Symbol	Reading	Defined in
$\gamma = (\Gamma, F)$	execution model: schedulers plus a failure model	Theorem 3.2
$A^{\perp}$, A^{ρ}	fail-stop lifting and reset lifting	Theorem 3.1
$\widehat{A}$	task lifting, which records the start value	Theorem 2.2
$\mathsf{retry}_{n,d}(A)$	retry with n attempts and backoff d	Theorem 4.3
$\mathsf{timeout}_t(A)$	timeout at t ticks	Theorem 4.1
$\mathsf{Sup}_{\sigma}(\vec{B})$	supervisor with strategy σ and a budget	Theorem 5.1
$\mathsf{Persist}_C(A)$	checkpointing at the cut set C	Theorem 6.1
$\mathsf{Log}(\ell)$, $\mathsf{Replay}(\ell)(A)$	choice log and replay against it	Theorem 7.1, Theorem 7.2
$\mathsf{Wf}_{\mathrm{ch}}(G)$	the same graph under the channel discipline	Theorem 10.1
$\mathsf{Wf}(G)$, $\mathsf{Wf}_{\mathrm{KPN}}$	workflow on a graph and its Kahn fragment	Theorem 10.1, Theorem 10.2
$\mathsf{Obs}_{\nu}(A, E)$	closed system observed through an effect projection	Theorem 2.4
$\prec_S$	dependency order of a system	Theorem 10.5

Definition 4.1 (Timeout)**.** Let $A : X \Rightarrow Y$ over (I, Σ), let $t \geq 0$, and let $\mathsf{to} \in E$ be a reserved reason. Part II's first-halt product $\mathsf{race}(A, \mathsf{delay}_t^X)$ has start type $X \times X$, value type $Y + X + Y \times X$ and interface $(I \times I, \Sigma \times \Sigma)$. Let b be the test on that value type which holds on the two summands that contain a value of Y, and let g be the partial map from that type to Y which is the identity on the summand Y and the first projection on the summand $Y \times X$, so that g is defined exactly where b holds. The *timeout* is

$$\mathsf{timeout}_t(A) \;=\; \mathsf{race}\big(A,\ \mathsf{delay}_t^X\big)\ ;\ \big(\ulcorner g \urcorner \oplus_b \mathsf{fail}_{\mathsf{to}}\big),$$

read at the diagonal start value $x \mapsto (x, x)$ and at the input relabelling $i \mapsto (i, \varepsilon)$, with the second output coordinate discarded. The relabellings are stateless functions on the alphabets and add no state, no tick and no operator.

Part II's race fails only when both factors have failed, so the shorter term $\mathsf{race}(A, \mathsf{delay}_t\ ;\mathsf{fail}_{\mathsf{to}})$ does not bound anything: if A runs forever the race runs forever, whatever the right branch does. Racing against delay_t, which halts rather than fails, and then converting the right injection into a failure is the term that bounds the outcome, and it does so under either convention for the failure clause of the race. Part II gives the same term with the timer arm tagged by a reserved value rather than carrying the start value; the two differ only in how the test reads the winning arm.

Proposition 4.2 (Timeout bound)**.** *For every $T \in \{\mathrm{Id}, \mathrm{P}, \mathrm{D}\}$, every agent $A : X \Rightarrow Y$, every start value x and every input word w of length $n \geq t + 1$, the outcome of $\mathsf{timeout}_t(A)$ after n ticks is not running. The bound is uniform in A.*

Proof. By Part II's definition, delay_t^X has state space $X \times \{0, \ldots, t\}$, starts at (x, t), decrements the counter on each tick while emitting τ, and halts with $\mathsf{inl}(x)$ at counter zero. After t ticks its counter is zero, so it is halted. Part II's race halts as soon as one factor has halted, so after t ticks the race is halted, with value in the Y summand if A halted no later than tick t and in the X summand otherwise. Halting is evaluated between ticks and consumes no tick, so the value is handed to $\ulcorner g \urcorner \oplus_b \mathsf{fail}_{\mathsf{to}}$ at that same boundary. The test b holds exactly on the summands where g is defined, so the left branch halts at once with a value of Y and the right branch fails at once, and the outcome of $\mathsf{timeout}_t(A)$ after t ticks is already halted or failed. By the stuttering rule for non-running states it stays so for every $n \geq t$, hence for every $n \geq t + 1$. □

The proof gives t where the statement claims $t+1$. The weaker bound is the one to quote, because it survives a convention in which the handoff of a sequential composition is observed at the start of the following tick rather than at the boundary that precedes it, and because an implementation measures the tick on which the composite is first observed to be non-running rather than the boundary at which it becomes so.

Definition 4.3 (Retry with backoff)**.** Let $A : X \Rightarrow Y$, let $n \geq 0$ and $d \geq 1$. Put $\mathsf{retry}_{0,d}(A) = \widehat{A}$ and

$$\mathsf{retry}_{k+1,d}(A) \;=\; \widehat{A} \;\triangleright\; (\ulcorner p_X \urcorner \,;\, \mathsf{delay}_d \,;\, \mathsf{retry}_{k,d}(A)),$$

where $p_X : E \times X \to X$ is the second projection. The *unbounded retry* is $\mathsf{retry}_{\infty,d}(A) = \mu Z.\, (\widehat{A} \triangleright (\ulcorner p_X \urcorner \,;\, \mathsf{delay}_d \,;\, Z))$.

The task lifting is what makes this typable. Part II's recovery starts its right branch at the failure reason, and the lifting arranges for the reason to carry the start value, so the retried attempt begins where the first one did.

Proposition 4.4 (Retry is well defined)**.** *For every T, $\mathsf{retry}_{n,d}(A)$ is an agent, and $A \sim A'$ implies $\mathsf{retry}_{n,d}(A) \sim \mathsf{retry}_{n,d}(A')$. For $d \geq 1$ the context defining $\mathsf{retry}_{\infty,d}$ is guarded, so it has a unique solution up to $\sim$; for $d = 0$ and an A that can fail without consuming a tick it is unguarded and has no unique solution.*

Proof. $\mathsf{retry}_{n,d}(A)$ is a finite term over $\triangleright$, ; and delay_d applied to $\widehat{A}$ and to reindexing agents, so it is an agent, and the congruence claim is Part II's congruence theorem applied one operator at a time together with Theorem 2.3. The guardedness claim is Part II's retry-guardedness proposition applied to the context $Z \mapsto \widehat{A} \triangleright (\ulcorner p_X \urcorner \,;\, \mathsf{delay}_d \,;\, Z)$, whose hole is reached only after delay_d has consumed d ticks, and uniqueness is Part II's unique-solutions theorem. □

Backoff is the condition under which unbounded retry denotes anything. An agent that fails immediately, retried immediately, gives the equation $Z = \mathsf{fail} \triangleright Z$, which every agent satisfies, so the equation picks out no agent in particular.

5 Supervision

Armstrong's supervision trees [1] put the recovery policy outside the component that fails. A supervisor owns a vector of children, watches their outcomes, restarts a set of them determined by a strategy when one of them stops, and gives up when restarts arrive faster than a stated budget. The three strategies of the Erlang/OTP supervisor behaviour differ only in that set.

Definition 5.1 (Supervisor)**.** Let $B_1, \ldots, B_n$ be crash liftings, with B_j over (I_j, Σ_j) of type $X_j \Rightarrow Y_j$ and reset map ρ_j. Let $\sigma \in \{\mathsf{one\_for\_one}, \mathsf{one\_for\_all}, \mathsf{rest\_for\_one}\}$ and let $\max_r \geq 0$, $\max_t \geq 1$. The *restart function* is

$$R_{\mathsf{one\_for\_one}}(j) = \{j\}, \qquad R_{\mathsf{one\_for\_all}}(j) = \{1, \ldots, n\}, \qquad R_{\mathsf{rest\_for\_one}}(j) = \{j, j+1, \ldots, n\},$$

extended to sets by $R_\sigma(K) = \bigcup_{j \in K} R_\sigma(j)$. Write $\mathsf{Sup}_\sigma(B_1, \ldots, B_n)$, with budget $(\max_r, \max_t)$, for the agent over the execution interface whose component set is $\{1, \ldots, n\}$ and whose state space is

$$\Big(\prod_j S_j\Big) \times \mathbb{N} \times \mathcal{H} \times \{\mathsf{up}, \mathsf{dead}\},$$

where $\mathcal{H}$ is the set of finite subsets of $\mathbb{N}$ of size at most $\max_r + 1$. Its value type is $\prod_j X_j \Rightarrow \prod_j Y_j$, its initial state is $((\mathsf{init}_j(x_j))_j,\ 0,\ \emptyset,\ \mathsf{up})$, and its step on input $((i_j)_j, \kappa)$ in a state $(\vec{s}, t, h, \mathsf{up})$ is the

following, where $\mathcal{F} = \kappa \cup \{j : \mathsf{halt}_j(s_j) \text{ is a failure}\}$ is the set of children that were struck by the crash port or have failed on their own.

- If $\mathcal{F} = \emptyset$, every running child steps on its own input, halted children stutter, the output is the tuple $(\sigma_j)_j$ of child outputs, and the tick counter advances.
- If $\mathcal{F} \neq \emptyset$, put $h' = \{t\} \cup \{u \in h : u > t - \max_t\}$. If $\#h' > \max_r$ the state becomes $(\vec{s}, t+1, h', \mathsf{dead})$ and the output is the marker crash. Otherwise every child in $J = R_\sigma(\mathcal{F})$ is replaced by $\rho_j(s_j)$, no child steps, the output is the marker $\mathsf{restart}_J$, and the state becomes $(\vec{s}', t+1, h', \mathsf{up})$.

The outcome map sends a dead state to $\mathsf{inm}(\mathsf{budget})$, a state in which every child has halted to inl of the tuple of values, and every other state to $\mathsf{inr}(*)$.

Definition 5.2 (Supervision tree)**.** A *supervision tree* is a finite tree whose leaves are crash liftings of agents and whose internal nodes are labelled by a strategy and a budget. Its denotation is defined by structural recursion: a leaf denotes its agent, and an internal node with children $t_1, \ldots, t_n$, strategy σ and budget $(\max_r, \max_t)$ denotes $\mathsf{Sup}_\sigma(\llbracket t_1 \rrbracket, \ldots, \llbracket t_n \rrbracket)$ with that budget, where $\llbracket t_i \rrbracket$ is crash-lifted with its own reset map.

Because the denotation of an internal node is again a crash lifting, a supervisor's failure with reason budget is exactly what makes its own parent restart it, which is the escalation discipline of the Erlang runtime.

Proposition 5.3 (Restart budget)**.** *Let* Sup_σ *have budget* $(\max_r, \max_t)$ *and let* φ *be a crash schedule. Call a tick* restart-triggering *when* $\mathcal{F} \neq \emptyset$ *at that tick. Then* Sup_σ *is in a* dead *state at the boundary after tick* n *if and only if there is an* $m \leq n$ *with*

$$\#\{t \in (m - \max_t,\ m] : t \text{ is restart-triggering}\} \ > \ \max_r .$$

The set of dead *states is closed under the step map, for every* $T \in \{\mathrm{Id}, \mathrm{P}, \mathrm{D}\}$*: it is an invariant, in the sense of Part I, of the agent obtained from* Sup_σ *by taking a* dead *state as the initial one.*

Proof. The set h maintained by the step is exactly the set of restart-triggering ticks in the trailing window, by induction on n: it is initially empty, entries older than $\max_t$ are discarded on every restart-triggering tick, and the current tick is added. The supervisor moves to dead exactly when $\#h' > \max_r$, which is the displayed condition at $m = t$. If the condition holds at some $m \leq n$ then the transition was taken at the least such m, and if it never holds then it was never taken. For the closure claim, a dead state carries the outcome $\mathsf{inm}(\mathsf{budget})$, so the stuttering rule for non-running states makes its step return the same state with weight η; the predicate lifting of T applied to the set of dead states therefore holds at every successor, which is Part I's inductiveness condition. The set of dead states does not contain the initial state of Sup_σ, so it is not an invariant of Sup_σ itself; the statement is that death is absorbing, and the greatest invariant theorem of Part I applies to the restarted agent. □

Proposition 5.4 (The Erlang restart sets)**.** *Let a supervised group consist of* n *children in a fixed start order, and consider the restart discipline of the Erlang/OTP supervisor behaviour described by Armstrong [1, pp. 118-123]: under* $\mathsf{one\_for\_one}$ *only the child that stopped is restarted, under* $\mathsf{one\_for\_all}$ *all children are terminated and restarted, and under* $\mathsf{rest\_for\_one}$ *the child that stopped and every child started after it are terminated and restarted. For every strategy* σ *and every sequence of stopping events, the sets restarted by that discipline are the sets* $R_\sigma(j)$ *of Theorem 5.1, and every finite restart sequence of the discipline is the restart sequence of a run of* Sup_σ *under a suitable crash schedule.*

Proof. The three set-valued clauses of the discipline are the three displayed definitions of R_σ, once the fixed start order is identified with $1 < \cdots < n$. For the simulation, let $e_1, \ldots, e_m$ be a finite sequence of stopping events with e_u naming the child that stopped at step u. Take the crash schedule that sets $\varphi(t_u) = \{e_u\}$ at a strictly increasing sequence of ticks $t_1 < \cdots < t_m$ chosen so that no window of length $\max_t$ contains more than $\max_r$ of them. By induction on u, the supervisor is in an up state at t_u by Theorem 5.3, and it therefore restarts $R_\sigma(\{e_u\})$ at that tick. The two sequences of restarted sets agree termwise. □

The simulation is one-way and deliberately narrow. A real supervisor also distinguishes permanent, transient and temporary children, runs an ordered shutdown protocol with timeouts before a restart, and permits children to be added and removed at runtime; Theorem 5.1 models none of these, and Section 13 says so.

6 Persistence

Checkpointing moves the restart target from the beginning of a component to its last saved state.

Definition 6.1 (Checkpointing)**.** Let $A = (S, \mathsf{init}, \mathsf{step}, \mathsf{halt})$ and let $C \subseteq S$ be a *cut set* containing $\mathsf{init}(X)$ and contained in the running states. Write A_C for the agent with state space $S \times C$, initializer $x \mapsto (\mathsf{init}(x), \mathsf{init}(x))$, outcome $\mathsf{halt}(s, c) = \mathsf{halt}(s)$, and step

$$\mathsf{step}_C((s, c), i) = T\Big((\sigma, s') \mapsto (\sigma,\ (s', c(s')))\Big)\big(\mathsf{step}(s, i)\big),$$

where $c(s') = s'$ when $s' \in C$ and $c(s') = c$ otherwise. The *checkpointed agent* $\mathsf{Persist}_C(A)$ is the reset lifting A^ρ_C with $\rho(s, c) = (c, c)$. Checkpointing *at every tick* is the case $C = S$. Taking a checkpoint is silent: the second component of the state changes and no marker is emitted, so A_C and A have the same output on every tick.

Proposition 6.2 (Checkpointing commutes with the two composition operators)**.** *Let $T \in \{\mathrm{Id}, \mathrm{P}, \mathrm{D}\}$.*

1. *Let $A : X \Rightarrow Y$ and $B : Y \Rightarrow Z$, let $C_A \subseteq S_A$ and $C_B \subseteq S_B$ be cut sets, and suppose C_A contains the halting boundary of A, meaning every state s with $\mathsf{halt}_A(s) = \mathsf{inl}(y)$ lies in C_A. Then*
$$\mathsf{Persist}_{C_A + C_B}(A\,;B) \ \sim_{\mathrm{er}}\ \mathsf{Persist}_{C_A}(A)\,;\mathsf{Persist}_{C_B}(B).$$
2. *Let A and B have cut sets C_A and C_B, let the cut of the product be the synchronous cut $C_A \times C_B$, and let the crash schedules of the product be synchronous, meaning the crash port delivers to both factors or to neither. Then*
$$\mathsf{Persist}_{C_A \times C_B}(A \otimes B) \ \sim_{\mathrm{er}}\ \mathsf{Persist}_{C_A}(A) \otimes \mathsf{Persist}_{C_B}(B).$$

Proof. For (1), define R to relate the state $(\mathsf{in}_1 s, \mathsf{in}_1 c)$ of the left-hand side to the state $\mathsf{in}_1(s, c)$ of the right-hand side, and $(\mathsf{in}_2 s, \mathsf{in}_2 c)$ to $\mathsf{in}_2(s, c)$. Every reachable state of the left-hand side has both components in the same summand: initially both are $\mathsf{in}_1\mathsf{init}_A(x)$; a step inside A can only move the saved component to a state of A; and at the handoff boundary the current state becomes $\mathsf{in}_2\mathsf{init}_B(y)$, which lies in C_B by the requirement that cut sets contain the initial states, so the saved component follows it into the second summand on the next tick, while the hypothesis that C_A contains the halting boundary makes the last saved state of the first summand the halting state itself. The step and reset clauses then match componentwise, and outcomes agree because halt ignores the saved

component. Both sides emit the same ordinary outputs, so the relation is a bisimulation already before erasure, and a fortiori after it.

For (2), the state spaces are $(S_A \times S_B) \times (C_A \times C_B)$ and $(S_A \times C_A) \times (S_B \times C_B)$, and the evident shuffle isomorphism matches the two on every tick that carries an ordinary output: under a synchronous cut, a joint state lies in $C_A \times C_B$ exactly when each component lies in its own cut, so the two sides save at the same ticks, and under a synchronous crash schedule both factors are reset together, so the two sides reset to the same pair. The crash ticks differ, and they differ in the alphabet rather than in the state. The left-hand side is a single component, so it emits the single marker crash of $(\Sigma_A \times \Sigma_B) \uplus M$. The right-hand side is a product of two components, so it emits the pair $(\mathsf{crash}, \mathsf{crash})$ of $\Sigma_A^\gamma \times \Sigma_B^\gamma$. Erasure sends the first to $\tau_{A\otimes B} = (\tau_A, \tau_B)$ and the second to (τ_A, τ_B), after which both live in $\Sigma_A \times \Sigma_B$ and agree, so the isomorphism is a bisimulation after erasure. Erasure is what makes the two sides comparable at all here, which is why the statement is at $\sim_{\mathrm{er}}$ and not at $\sim$. □

Remark 6.3 (Both hypotheses are load-bearing). Take A to be the agent that emits a for two ticks and halts, B the agent that emits b for two ticks and halts, and $C_A = \{\mathsf{init}_A(x)\}$, which omits A's halting state. A crash on the first tick of B rolls the left-hand side of (1) back into A and reruns it, producing the erased word $a\,a\,b\,\tau\,a\,a\,b\,b$, while the right-hand side rolls back only to init_B and produces $a\,a\,b\,\tau\,b\,b$. For (2), drop the synchrony of the crash schedule and crash only the first factor: the right-hand side resets A while B continues, reaching a pair of states that the left-hand side, which resets both, cannot reach. Recovering the product law for independent crash channels is the problem of consistent global cuts, for which Chandy and Lamport [4] give the standard marker algorithm; this Part does not treat it, and Section 13 records the omission.

7 Logs and replay

Durable execution engines re-run a workflow from the beginning after a failure and substitute recorded results for the steps that already completed, so that the re-run follows the original path without repeating its effects. Burckhardt and coauthors [3] give an operational semantics for this mechanism and prove two lower-level execution models, one with compute and storage separated and one with record and replay, equivalent to a high-level model. The construction below is the same mechanism expressed in the algebra, and the substituted quantity is the choice component of Part I's policy and update factorization.

Definition 7.1 (Resolutions, logs and compatibility)**.** Let A be an agent with factorization $\mathsf{step}(s,i) = \mathbf{do}\ (\sigma,k) \leftarrow \pi(s,i);\ \eta(\sigma, \delta(s,i,\sigma,k))$. A *resolution* at (s,i) is a pair $r = (\sigma,k) \in \mathrm{supp}(\pi(s,i))$, where the support is the singleton for Id, the set itself for P, and the set of positive-weight elements for D. A *log of length* N is a word $\ell = (i_1, r_1) \cdots (i_N, r_N)$ of inputs paired with resolutions. It is *A-compatible from x* when the states $s_0 = \mathsf{init}(x)$ and $s_t = \delta(s_{t-1}, i_t, \sigma_t, k_t)$ are defined and running for $t < N$ and satisfy $r_t \in \mathrm{supp}(\pi(s_{t-1}, i_t))$ for every $t \le N$. The *log agent* $\mathsf{Log}(\ell)$ is Part III's channel Ch_∞ preloaded with the records of ℓ together with a read cursor, which serves one record per dequeue request and reports exhaustion afterwards.

Definition 7.2 (Replay)**.** Let ℓ have length N and let A be an agent over T with the factorization above. The *replay agent* $\mathsf{Replay}(\ell)(A)$ is the Id agent with state space $S \times \{0, \ldots, N\}$, initializer $x \mapsto (\mathsf{init}(x), 0)$, outcome $\mathsf{halt}(s,t) = \mathsf{halt}_A(s)$, and, in a state (s,t) with $t < N$ and $\mathsf{halt}_A(s)$ running, the step on input i given by

- failure with reason $\mathsf{mismatch}$ if $i \neq i_{t+1}$;

- failure with reason $\mathsf{nondeterminism}$ if $i = i_{t+1}$ but $r_{t+1} \notin \mathrm{supp}(\pi(s, i_{t+1}))$;
- otherwise the output σ_{t+1} and the state $(\delta(s, i_{t+1}, \sigma_{t+1}, k_{t+1}),\ t+1)$.

In a state (s, N) the agent behaves as A does. A replayed tick emits the output it is replaying and no marker, so replay is silent and $\mathsf{Replay}(\ell)(A)$ is comparable with A without erasure. Writing $\mathsf{Replay}(\ell)(A)$ as $\mathsf{link}_{\log}(\mathsf{Log}(\ell), A)$ recovers the same agent up to $\sim$ when A's choice port is exposed, which is why the log is Part III's channel and not a new construct.

Theorem 7.3 (Replay determinism)**.** *Let $T \in \{\mathrm{Id}, \mathrm{P}, \mathrm{D}\}$, let A be an agent over T, let $x \in X$, and let $\ell = (i_1, r_1) \cdots (i_N, r_N)$ be A-compatible from x with induced trajectory $s_0, \ldots, s_N$ and output word $\sigma_1 \cdots \sigma_N$.*

1. *$\mathsf{Replay}(\ell)(A)$ is an agent over* Id*, and on the input word $i_1 \cdots i_N$ from x its state trajectory is $(s_0, 0), \ldots, (s_N, N)$, its output word is $\sigma_1 \cdots \sigma_N$, and its outcome after N ticks is $\mathsf{halt}_A(s_N)$.*
2. *Let A' be an agent over T with the same interface, value types and factorization shape. Then $\mathsf{Replay}(\ell)(A')$ completes the first N ticks without failing with reason* $\mathsf{nondeterminism}$ *if and only if ℓ is A'-compatible from x, and in that case its output word on $i_1 \cdots i_N$ is again $\sigma_1 \cdots \sigma_N$.*

Proof. (1) The step of Theorem 7.2 returns a single pair for every state and input, so the agent is over Id regardless of T. Induct on t. At $t = 0$ the state is $(\mathsf{init}(x), 0) = (s_0, 0)$. Assume the state after $t < N$ ticks is (s_t, t). The input i_{t+1} matches, so the mismatch clause does not fire; A-compatibility gives $r_{t+1} \in \mathrm{supp}(\pi(s_t, i_{t+1}))$, so the nondeterminism clause does not fire; the third clause emits σ_{t+1} and moves to $(\delta(s_t, i_{t+1}, \sigma_{t+1}, k_{t+1}), t+1) = (s_{t+1}, t+1)$. Outcomes agree because halt of the replay agent is halt_A of the first component.

(2) Run the same induction against A', writing s'_t for its states. Suppose the replay completes N ticks without a nondeterminism failure. Then at each t the guard $r_{t+1} \in \mathrm{supp}(\pi'(s'_t, i_{t+1}))$ held and the successor was $s'_{t+1} = \delta'(s'_t, i_{t+1}, \sigma_{t+1}, k_{t+1})$, which is precisely the definition of A'-compatibility of ℓ from x. Conversely, if ℓ is A'-compatible then the guard holds at every t by definition and the replay completes. The emitted outputs are the σ_t read from the log in both directions, because the third clause never consults A' for the output. □

Corollary 7.4 (Reproducible stochastic runs)**.** *Let A be an agent over* D *and let ℓ record the sampled pairs of a run of A. Then $\mathsf{Replay}(\ell)(A)$ is deterministic and reproduces that run exactly, and every agent obtained from A by a change that leaves* $\mathrm{supp}(\pi)$ *containing the logged resolutions replays correctly. A change that removes a logged resolution from the support is detected on the tick where it occurs.*

The corollary is the formal content of a practice that durable workflow engines enforce by convention: a workflow may be edited between a failure and its replay, but only in ways that keep the recorded history reachable, and an edit that violates this is reported rather than silently followed. The detection is exact because $\mathrm{supp}(\pi)$ is a decidable set on the finite models this Part considers.

8 Supervision refinement

A supervised system cannot behave identically to the unsupervised one, because it emits markers and spends ticks on restarts. Erasing the markers and closing under stuttering makes the two comparable, and under that comparison they refine each other in both directions provided every child is checkpointed at every tick.

Definition 8.1 (Replay-idempotent environment)**.** Let E be an environment co-agent over $(\Sigma^\gamma, I^\gamma)$ with state space S_E, and let $\mathsf{key} : \Sigma \to K + 1$ assign a deduplication key to outputs, with the right summand meaning "no key". Then E is *replay-idempotent for* key when

1. (quiescence) $\mathsf{step}_E(q, \sigma) = \eta(\varepsilon, q)$ for every q and every $\sigma \in \{\tau\} \cup M_N$, which is Part III's epsilon-quiescence read on the dual interface, extended to the markers; and

2. (memoization) if q is reachable by a history in which an output with key κ was already delivered and answered by i, then $\mathsf{step}_E(q, \sigma) = \eta(i, q)$ for every σ with $\mathsf{key}(\sigma) = \kappa$.

Clause (1) covers the markers as well as the idle output, and it has to. Markers are events of the runtime and not messages to the environment, and Part I's erasure is applied to a completed trace rather than on the wire, so nothing in the closed loop stops a marker from reaching E. An environment that reacted to a marker would see the crash it is not supposed to see, and Theorem 8.2 would fail. An implementation realizes the clause by erasing markers on the link that carries the system's output to its environment, which is the relabelling of Theorem 2.4 applied one step earlier.

Lemma 8.2 (Crash and restart ticks are stutter positions)**.** *Let the failure model be isolating, and let E satisfy clause (1) of Theorem 8.1. Let u be a configuration of an observed closed system whose agent component is a supervisor or a crash lifting, and let $u \to u'$ be a tick whose output is* crash *or* $\mathsf{restart}_J$*. Then the erased output of that tick is τ, the environment state is unchanged, and the closed-loop register is unchanged.*

Proof. By Theorem 3.1 and Theorem 5.1, the whole output of such a tick is a marker $m \in M_N$, so erasure gives τ. The environment receives m itself, since erasure acts on traces and not on the wire, and clause (1) of Theorem 8.1 covers M_N, so $\mathsf{step}_E(q, m) = \eta(\varepsilon, q)$ and the environment neither moves nor answers. Isolation supplies the idle input to every component of the system other than those being reset, so no sibling advances either. The crash port is hidden in a γ-execution, so the observable input of the tick is the environment's answer ε, and the position therefore carries the idle input and the idle output, which is Part I's condition for an idle position. The register of Part I's closed loop is written with the tick's output and the environment's answer, both idle after erasure, so it holds the same pair as before. □

Theorem 8.3 (Supervision refinement)**.** *Let $A_1, \dots, A_n$ be* Id *agents that never enter a failed state, let $P_j = \mathsf{Persist}_{S_j}(A_j)$ be checkpointed at every tick, let σ be any of the three strategies, and let $\gamma = (\Gamma, F)$ be an execution model whose failure model is isolating and $(\max_r, \max_t)$-bounded for the supervisor's budget. Let E be replay-idempotent and let ν be an effect projection. Then for every crash schedule $\varphi \in F$,*

$$\text{erase } \mathsf{Obs}_\nu(\mathsf{Sup}_\sigma(\vec{P}), E)\big|_\varphi \;\sqsubseteq\; \mathsf{Obs}_\nu\Big(\bigotimes_j A_j, E\Big),$$
$$\mathsf{Obs}_\nu\Big(\bigotimes_j A_j, E\Big) \;\sqsubseteq\; \text{erase } \mathsf{Obs}_\nu(\mathsf{Sup}_\sigma(\vec{P}), E)\big|_\varphi.$$

Proof. Write L for the supervised observed closed system under φ and R for the crash-free one.

Step 1: the checkpoint is the current state. At every tick boundary of L, each child's current state equals its saved state. Initially both are $\mathsf{init}_j(x_j)$. On a tick with $\mathcal{F} = \emptyset$ the child steps to s', and $C_j = S_j$ gives $s' \in C_j$, so s' becomes the saved state. On a tick with $\mathcal{F} \neq \emptyset$ each child in J is replaced by $\rho_j(s_j, c_j) = (c_j, c_j)$ and the others do not move, so the property persists.

Step 2: the core map. For a configuration u of L with supervisor state $(\vec{s}, t, h, \mathsf{up})$, environment state q and register reg, define $\mathrm{core}(u) = ((\pi_1 s_j)_j, q, \mathrm{reg})$, the vector of child current states together

with the environment and the register. Let $\mathcal{R}$ relate u to the configuration $\mathrm{core}(u)$ of R. The initial configurations are related, since $\mathcal{H}$ and the counters do not occur in core.

Step 3: non-crash ticks. Suppose $u \to u'$ with $\mathcal{F} = \emptyset$. Every running child of Sup_σ steps on its own input exactly as the corresponding factor of $\bigotimes_j A_j$ does, because $\mathsf{Persist}_{S_j}$ changes only the saved component and takes its checkpoints silently, and the children are deterministic, so the two systems emit the same output and no marker. The environment receives that output and answers identically. Hence $\mathrm{core}(u) \to \mathrm{core}(u')$ in R with the same erased output.

Step 4: crash and restart ticks. Suppose $u \to u'$ with $\mathcal{F} \neq \emptyset$. Boundedness of F and Theorem 5.3 give that the supervisor stays in up. Every child in J is reset to its saved state, which by Step 1 equals its current state, and no child steps, so the vector of current states is unchanged. Theorem 8.2 gives that the environment and the register are unchanged and that the erased output is the idle output. Hence $\mathrm{core}(u') = \mathrm{core}(u)$ and the tick contributes an idle position to the erased trace.

Step 5: outcomes. The supervisor halts exactly when every child has halted, and by Steps 3 and 4 the vector of child states of L at a boundary is the vector of child states of R at the corresponding boundary, so the two systems halt with the same tuple of values, and neither fails: the children do not fail by hypothesis, and the supervisor does not exhaust its budget.

Step 6: conclusion. Steps 3 and 4 exhibit the erased trace of L as the trace of R with idle positions inserted at the crash and restart ticks. Since $\sqsubseteq$ is inclusion of stutter-closed trace sets (Part I) and the stutter closure is closed under insertion and deletion of idle positions, the erased trace of L lies in the stutter closure of the trace set of R, which is the first inclusion. For the second, the empty crash schedule lies in F, and under it L has no crash or restart tick, so by Step 3 alone its erased trace is the trace of R; every trace of R is therefore an erased trace of L for some admissible schedule. □

Corollary 8.4 (Under checkpointing the strategy is invisible)**.** *Under the hypotheses of Theorem 8.3, the erased and stutter-closed traces of* $\mathsf{Sup}_{\mathsf{one\_for\_one}}(\vec{P})$, $\mathsf{Sup}_{\mathsf{one\_for\_all}}(\vec{P})$ *and* $\mathsf{Sup}_{\mathsf{rest\_for\_one}}(\vec{P})$ *coincide.*

Proof. Step 4 of the proof used only that every reset child is returned to its saved state, and the saved state is the current state by Step 1, so the identity of the set J does not enter. All three systems therefore have the same erased trace as $\bigotimes_j A_j$ under the same crash schedule. □

Under checkpointing at every tick the Erlang strategies differ in cost and not in behaviour: $\mathsf{one\_for\_all}$ discards and reloads work that $\mathsf{one\_for\_one}$ leaves alone, and the erased traces coincide. They differ in behaviour exactly when the persistence is coarser than the crash granularity, which is the quantity a deployment should measure.

Counterexample 8.5 (The counter child)**.** Let A be the Id agent with states $0, 1, 2, 3$, initial state 0, step $j \mapsto (j+1, j+1)$ emitting the numeral it enters, and $\mathsf{halt}(3) = \mathsf{inl}(*)$. Take one child, the strategy $\mathsf{one\_for\_one}$, the reset map ρ_{init} of Theorem 2.2 rather than a checkpoint, a budget that permits one restart, and the crash schedule $\varphi(3) = \{1\}$. The erased output word of the supervised run is $1\,2\,\tau\,1\,2\,3$, while the crash-free word is $1\,2\,3$. Deleting the idle position from the first word leaves $1\,2\,1\,2\,3$, which is not the second, so the first refinement of Theorem 8.3 fails. The failure is not repaired by a different strategy, a larger budget or a fair scheduler; it is repaired only by checkpointing.

Remark 8.6 (Isolation is not free)*.* Drop isolation and keep everything else. Let the child emit a request on tick 1, let the environment answer on tick 2, and let the crash occur on tick 2. The child rolls back to its state at the end of tick 1 and reads, on tick 3, whatever the environment produced on tick 2; but the environment has already consumed the request and moved on, so the answer

that the child needed was delivered into the tick that the crash destroyed. The erased trace of the supervised run then omits an input that every crash-free run consumes, and no stuttering closure repairs a deletion of a non-idle position. Two standard designs avoid this. One is isolation, which is what Theorem 3.1 assumes and what a runtime implements by pausing the group while a restart is in progress. The other is to wire the system with the channels of Part III instead of the one-tick register, so that an undelivered input remains in the channel across the crash; the argument is then the same with "the register is unchanged" replaced by "the channel is unchanged", at the cost of leaving Part I's closed loop for Part III's linked system.

Remark 8.7 (Where the deduplication key is used). Clause (2) of Theorem 8.1 is not used in the proof above. With $C_j = S_j$ the rollback target is the state at the end of the previous tick, so no output is ever re-emitted and there is nothing for the environment to deduplicate. The clause becomes necessary as soon as the cut set is coarser than every tick, because then a rollback re-executes the ticks between the checkpoint and the crash and re-emits their outputs; and it is necessary for Theorem 9.2, where the rollback target is the beginning of the whole attempt. The clause is therefore kept as a standing hypothesis, with the one theorem that does not consume it marked here.

9 Effectively-once execution

Retry is checkpointing at its coarsest: the rollback target is the beginning of the attempt. Everything the aborted attempt emitted is emitted again, so no equality of output words can hold. The environment can still end in the same state and pass through the same observable effects, which is what industrial systems promise when they offer exactly-once effects on top of at-least-once delivery. Helland [7] states the same discipline informally: messages may be retried, and idempotence means that this is acceptable.

Definition 9.1 (Keyed agent)**.** An agent A is *keyed* for key when along every run every non-idle output carries a key and no key is emitted twice.

Theorem 9.2 (Effectively-once)**.** *Let $A : X \Rightarrow Y$ be an* Id *agent, keyed for* key*, whose crash-free run from x halts after N ticks with value y. Let E be replay-idempotent for* key*, let ν be an effect projection, and let the failure model be isolating with crash points at tick boundaries only and with crashes delivered to the attempt and not to the backoff. Write $R = \mathsf{retry}_{n,d}(A^{\perp})$, the n-fold retry of the fail-stop lifting of A, so that a crash makes the current attempt fail and the recovery operator starts the next one. Then for every crash schedule φ with at most n crashes,*

$$\begin{aligned} \text{erase } \mathsf{Obs}_E(R,E)\big|_{\varphi} &\sqsubseteq \mathsf{Obs}_E(A,E), \\ \mathsf{Obs}_E(A,E) &\sqsubseteq \text{erase } \mathsf{Obs}_E(R,E)\big|_{\varphi}, \end{aligned}$$

and the retried system halts with the same value y.

Proof. Let $a_0, \dots, a_N$ and $q_0, \dots, q_N$ be the state sequences of A and E in the crash-free run, let $\sigma_1, \dots, \sigma_N$ and $i_1, \dots, i_N$ be its outputs and inputs, and let $\nu_t = \nu(q_{t-1}, q_t)$ be its effect trace.

The induction is on the number m of crashes in φ, with the following hypothesis, stated for every $0 \le j \le N$ and every $k \ge m$: if E is in the state q_j reached by delivering $\sigma_1, \dots, \sigma_j$, then the run of $\mathsf{retry}_{k,d}(A^{\perp})$ from x against E under a schedule with m crashes has erased effect trace equal to $\nu_{j+1} \cdots \nu_N$ with idle positions inserted, and halts with y.

For $m = 0$ the attempt is the crash-free run of A from x. Its first j ticks re-emit $\sigma_1, \dots, \sigma_j$, whose keys E has already answered, so by clause (2) of Theorem 8.1 the environment replies $i_1, \dots, i_j$

without changing state; determinism of A then makes the attempt retrace $a_0, \ldots, a_j$, and those j ticks contribute τ_V to the effect trace by the condition $\nu(q, q) = \tau_V$. From a_j the run continues as the crash-free run does from tick $j + 1$, since every remaining output carries a key E has not seen, so the remaining effect trace is $\nu_{j+1} \cdots \nu_N$ and the value is y.

For $m > 0$, let the first crash fall on tick c of the run. The ticks before it retrace $a_0, \ldots$ exactly as in the previous paragraph, so they contribute j idle positions and then $\nu_{j+1} \cdots \nu_{j'}$ for some $j' \geq j$, where $a_{j'}$ is the state the attempt had reached and $q_{j'}$ is the environment state. On tick c the fail-stop lifting emits the marker crash and enters its failed state. By Theorem 8.2 the environment does not move and the erased output is τ, so the tick is an idle position. The recovery operator of Part II then hands control, at that same boundary and without consuming a tick, to $\ulcorner p_X \urcorner \,;\, \mathsf{delay}_d \,;\, \mathsf{retry}_{k-1,d}(A^{\perp})$, whose first component halts at once and whose second consumes d ticks emitting τ; quiescence of E makes those d ticks idle positions as well.

What remains is a run of $\mathsf{retry}_{k-1,d}(A^{\perp})$ from x against E in the state $q_{j'}$, under a schedule with $m - 1$ crashes. Since $k - 1 \geq m - 1$, the induction hypothesis applies with j' in place of j and gives erased effect trace $\nu_{j'+1} \cdots \nu_N$ with idle positions inserted, and value y. Concatenating, the whole run has erased effect trace $\nu_{j+1} \cdots \nu_N$ with idle positions inserted, which is the hypothesis at m.

Taking $j = 0$ and q_0 gives that the erased effect trace of the retried run is the crash-free effect trace with idle positions inserted. Deleting those positions returns the crash-free trace exactly, so each system's effect trace lies in the stutter closure of the other's, which is the pair of refinements; the empty schedule lies in every failure model, which gives the second refinement for every trace of the crash-free system. The value is y in both. □

Counterexample 9.3 (The counting environment)**.** Let E have state $\mathbb{N}$, increment on every non-idle output, answer ε always, and let key be constant at the no-key value, so clause (2) of Theorem 8.1 is vacuous while clause (1) still holds. Let $\nu(q, q + 1) = \mathsf{inc}$ and $\nu(q, q) = \tau_V$. Take A to emit one keyed request per tick for $N = 3$ ticks, and let the crash fall on tick 3. The crash-free effect trace has three occurrences of inc; the retried one has five, since the second attempt re-emits the first two requests and the environment counts them. Deleting idle positions from the second gives a word of length five, so it is not in the stutter closure of a word of length three, and the first refinement of Theorem 9.2 fails. The failure is exactly the loss of clause (2).

Remark 9.4. The theorem is about the environment's observable effects and not about the wire. An observer that records every byte the agent sends sees the duplicates, and no hypothesis of Theorem 9.2 removes them. At-least-once delivery with a deduplicating receiver gives effectively-once effects, and it does not give exactly-once delivery.

10 Workflows

A workflow engine runs a directed acyclic graph of tasks, and two of its properties are settled below. Deterministic acyclic workflows with blocking reads produce the same histories under every fair interleaving, and their dependency orders exceed what the sequence and product operators of Part II reach.

10.1 The construction and its fragment

Definition 10.1 (Workflow)**.** Let $G = (V, \mathrm{Ed})$ be a finite directed acyclic graph. A *workflow assignment* gives each $v \in V$ an agent N_v whose input ports are the in-edges of v together with a private port, and whose output ports are the out-edges of v. Two disciplines are used below, and they answer the two different questions of this section.

Under the *channel discipline* the workflow $\mathsf{Wf}_{\mathrm{ch}}(G)$ is the interleaving composition, in the sense of Part III, of the agents N_v and one channel Ch_k per edge, linked so that the channel of $e = (u, v)$ carries u's output on e to v's input on e, and scheduled by a scheduler in Γ. This is the discipline of a real engine, and it is the one Theorem 10.3 is about.

Under the *early-start discipline* the workflow $\mathsf{Wf}(G)$ steps, on each tick, exactly those nodes that are *active*: a node is active from the first tick after all its predecessors have halted until it halts, and every source node is active from tick one. Active nodes step in lockstep and the output of a tick is the tuple of the outputs of all nodes, with τ in the coordinate of every inactive node. This discipline removes the scheduler from the picture and leaves only the dependency structure, which is what Theorem 10.7 and Theorem 10.9 are about.

Definition 10.2 (The Kahn fragment)**.** A workflow under the channel discipline lies in $\mathsf{Wf}_{\mathrm{KPN}}$ when every node agent is over Id and epsilon-quiescent; every node blocks on its reads, so that its state after any number of ticks is a function of its initializer and the sequence of values it has dequeued; every node is *productive*, meaning that a scheduled step either dequeues a value or writes one; every edge channel has capacity $k = \infty$; the graph is acyclic; and no node contains race, retry or timeout.

The last clause excludes the constructions whose behaviour depends on elapsed time. A node containing a timeout reacts to how many ticks passed while it waited, and the number of ticks between two dequeues is what a scheduler chooses.

Theorem 10.3 (Kahn determinism for acyclic workflows)**.** *Let $\mathsf{Wf}_{\mathrm{ch}}(G)$ lie in $\mathsf{Wf}_{\mathrm{KPN}}$ and let every scheduler in Γ be fair. Fix the histories of the source ports. Then for every edge e of G, the history written on e is the same, as a finite or infinite sequence, under every scheduler in Γ; in particular the histories of the sink ports do not depend on the interleaving.*

Proof. Fix a topological order $v_1, \ldots, v_m$ of V, which exists because G is acyclic. Prove by induction on i that the limit histories of all in-edges of v_i are the same under every scheduler.

For the base of the induction, the in-edges of a source node are the private ports, whose histories are fixed by hypothesis.

For the step, assume the in-edge histories $H_1, \ldots, H_p$ of v_i are scheduler-independent. Since the node blocks on its reads and is over Id, its state after any number of ticks is a function of init and of the sequence it has dequeued, so the sequence of values it writes on each out-edge is a function of the prefixes it has dequeued; this is the single-node instance of Part III's locality theorem, and for the two-node case it is Part III's determinacy proposition for a FIFO link, which assumes exactly the fairness hypothesis assumed here. It remains to show that v_i dequeues all of $H_1, \ldots, H_p$ and writes the whole induced output.

Suppose v_i stops after a finite prefix of its in-edge histories. Two cases arise. If every port v_i blocks on becomes nonempty, then v_i is enabled at infinitely many ticks; its out-edges have capacity ∞, so a write never blocks; fairness schedules a component enabled at infinitely many ticks at infinitely many ticks; and productivity makes each such step dequeue a value or write one, so v_i advances past any finite prefix, contradicting the assumption. If instead v_i blocks forever on a port that stays empty, then it stops at the point its own stream function assigns to the in-edge histories, and those histories are scheduler-independent by the induction hypothesis, so the stopping point and the out-edge histories are scheduler-independent as well. Both cases give out-edge histories determined by the in-edge histories. Hence the out-edge histories are the ones determined by the input histories, which are scheduler-independent by the induction hypothesis, and the induction goes through. □

Acyclicity replaces the least-fixed-point argument that Kahn [8] uses for networks with cycles by an induction along a topological order. For cyclic networks the statement is Kahn's, and the limit is the least fixed point of the continuous stream function induced by the nodes; Lee and Parks [10] survey the engineering form of the same result and the scheduling conditions under which an implementation realizes it.

Remark 10.4 (Three ways out of the fragment). A merge node that reads whichever of two inputs is available is not a function of the dequeued sequence, and Part III's tuple-space proposition supplies a two-consumer witness with at least two outcomes; the conclusion of Theorem 10.3 fails for it. A node containing $\mathsf{timeout}_t$ fails the last clause of Theorem 10.2 and its behaviour depends on the number of ticks the scheduler let pass. Finite channel capacity fails the third clause: with $k < \infty$ a producer can block on a full channel, so a fair schedule can reach a configuration in which every node is blocked, and which configuration is reached depends on the schedule. The code layer checks the fragment with k at least the tick budget, which is a bounded model of the unbounded statement and never a substitute for it.

10.2 Dependency shape

Expressiveness needs an observable that records when a leaf runs, in an alphabet shared by a workflow and a term, and leaves that can run for arbitrary lengths of time; otherwise a coincidence of durations makes two structurally different systems agree.

Definition 10.5 (Duration-flexible leaves, activity traces and the dependency order)**.** A leaf agent u is *duration-flexible* when it reads a private input port and, for every $k \geq 1$, there is an input word on that port for which u runs for exactly k ticks, emitting a letter ℓ_u on each of them and halting afterwards. Let S be a system built from a finite set L of such leaves by $;$, $\otimes$ and the early-start discipline, so that the output of S at a tick is a tuple of leaf letters and idle symbols. The *activity map* α sends such a tuple to the set of leaves whose letter occurs in it, and the *activity trace* of a run is the word over 2^L obtained by applying α tickwise. Write $\mathcal{A}(S)$ for the set of activity traces of S over all start values and input words, and write $u \prec_S v$ when every $w \in \mathcal{A}(S)$ has all its positions containing u strictly before all its positions containing v. The relation $\prec_S$ is the *dependency order* of S.

The activity map is a function of the output alphabet alone, so it commutes with every construction and $\mathcal{A}(S)$ depends only on the trace set of S. Two systems with different activity-trace sets are therefore not trace-equivalent, and by Part I's theorem that bisimilarity implies trace equivalence they are not bisimilar either, whatever their output alphabets. Comparing $\mathcal{A}$ rather than the raw traces is what lets a workflow on four nodes and a term whose product operator emits pairs be measured against each other.

Lemma 10.6 (The dependency order of a term)**.** *Let t be a term built from distinct duration-flexible leaves with private input ports, using only $;$ and $\otimes$. Then $u \prec_t v$ if and only if the least common ancestor of u and v in the syntax tree of t is a $;$ node with u in its left subtree and v in its right.*

Proof. If the least common ancestor is such a $;$ node, then by Part II's definition of sequential composition the whole left factor reaches a halting state before the right factor is initialized, so every tick on which u emits precedes every tick on which v emits, giving $u \prec_t v$.

Conversely, suppose the least common ancestor is a $\otimes$ node w, with u in the left factor t_L and v in the right factor t_R. Both factors of $\otimes$ are initialized on the same tick, by Part II's definition of the product. Give every leaf other than u the duration 1, which duration flexibility allows, and let

c_u and c_v be the resulting ticks, counted from the start of w, at which u and v become active; both are finite and determined by t. Now give u the duration D. Changing u's duration does not change c_u or c_v, because u lies in t_L and v in t_R and no leaf of t_R waits for a leaf of t_L under $\otimes$. Choosing $D > c_v - c_u + 1$ makes u active at tick c_v, which is a tick at which v is active, so some activity trace has a position containing both u and v. Hence $u \prec_t v$ fails, and by symmetry so does $v \prec_t u$. □

Proposition 10.7 (The dependency order of a term is series-parallel)**.** *For every term t over ; and $\otimes$ with distinct duration-flexible leaves, $\prec_t$ is a series-parallel partial order, that is, an order built from singletons by disjoint union and ordinal sum. Its Hasse diagram is a two-terminal series-parallel digraph.*

Proof. Induct on t using Theorem 10.6. For a leaf, $\prec_t$ is the one-element order, a singleton. For $t = t_1 \,;\, t_2$, the lemma gives $u \prec_t v$ exactly when $u \prec_{t_1} v$, or $u \prec_{t_2} v$, or $u \in L(t_1)$ and $v \in L(t_2)$, which is the ordinal sum $\prec_{t_1} \oplus \prec_{t_2}$. For $t = t_1 \otimes t_2$, the lemma gives $u \prec_t v$ exactly when u and v lie in the same factor and are related there, which is the disjoint union $\prec_{t_1} + \prec_{t_2}$. The two constructions are the two constructors of series-parallel orders, and the corresponding graph operations are series and parallel composition of two-terminal digraphs, which is the inductive definition of a two-terminal series-parallel digraph used by Valdes, Tarjan and Lawler [15, Section 1]. □

Lemma 10.8 (The dependency order of a workflow)**.** *Let $\mathsf{Wf}(G)$ have duration-flexible node agents, early start and a fair scheduler that runs every enabled node. Then $\prec_{\mathsf{Wf}(G)}$ is the reachability order of G.*

Proof. If u reaches v in G then the early-start discipline makes v wait for every node on a path from u to v, and in particular for u, so $u \prec v$. If u does not reach v, give every node other than u the duration 1 and let c_u and c_v be the resulting ticks at which u and v become active; both are finite. Give u the duration D. Since u is not an ancestor of v, changing u's duration leaves c_v unchanged, and it leaves c_u unchanged as well. Choosing $D > c_v - c_u + 1$ makes u active at tick c_v, so some activity trace has a position containing both, and $u \prec v$ fails. □

Counterexample 10.9 (A workflow that is not a term)**.** Let N be the four-node graph with nodes a, b, c, d and edges $a \to c$, $a \to d$ and $b \to d$, with duration-flexible node agents and the early-start discipline. Then no term t over ; and $\otimes$ with leaf set $\{a, b, c, d\}$ satisfies $\mathcal{A}(t) = \mathcal{A}(\mathsf{Wf}(N))$. In particular $\mathsf{Wf}(N)$ is not bisimilar to any such term.

Proof. Suppose $\mathcal{A}(t) = \mathcal{A}(\mathsf{Wf}(N))$ for such a term. The dependency order is defined from the activity-trace set, so $\prec_{\mathsf{Wf}(N)} = \prec_t$. By Theorem 10.8 the left side is the order $a < c$, $a < d$, $b < d$ with the three pairs $\{a, b\}$, $\{b, c\}$ and $\{c, d\}$ incomparable. By Theorem 10.7 the right side is series-parallel, so this order would be either a disjoint union of two nonempty orders or an ordinal sum of two nonempty orders. It is not a disjoint union, because its comparability graph contains the path $c - a - d - b$ and is therefore connected. It is not an ordinal sum $P \oplus Q$ with both parts nonempty, because in an ordinal sum any two incomparable elements lie in the same part: a and b force a, b into one part, a and d force a, d into one part, and c and d force c, d into one part, so all four elements lie in one part and the other is empty. The contradiction proves the first claim, and the second follows because bisimilar agents are trace-equivalent by Part I and the activity-trace set is a function of the trace set. □

The four-element order used above is the obstruction that Valdes, Tarjan and Lawler [15, Section 2] identify: a finite order is series-parallel exactly when it contains no induced copy of it, and their

recognition algorithm decides the property in linear time. The connection to Part II is direct. Part II shows that the exchange law fails for the synchronous product, which is the algebraic form of the observation that $\otimes$ cannot express a dependency between a leaf of one factor and a leaf of the other; Theorem 10.9 is the shape-level form of the same fact.

11 Worked examples

11.1 A durable pipeline

Consider a document pipeline with three stages: $\mathsf{Fetch} : \mathsf{Url} \Rightarrow \mathsf{Doc}$, which calls an external service; $\mathsf{Summarize} : \mathsf{Doc} \Rightarrow \mathsf{Text}$, which calls a language model and is therefore an agent over D; and $\mathsf{Publish} : \mathsf{Text} \Rightarrow \mathsf{Id}$, which writes to a store. The runtime term is

$$\mathsf{Pipeline} \;=\; \mathsf{Persist}_C\Big(\mathsf{retry}_{3,2}\big(\mathsf{timeout}_{30}(\mathsf{Fetch})\big)\,;\,\mathsf{retry}_{2,5}\big(\mathsf{timeout}_{60}(\mathsf{Summarize})\big)\,;\,\mathsf{Publish}\Big),$$

with C containing the two handoff boundaries and the initial states, so that clause (1) of Theorem 6.2 applies and the checkpointed pipeline is the sequential composition of its checkpointed stages.

Theorem 4.2 bounds each stage: Fetch is not running after tick 31 of its attempt, so the whole first stage occupies at most $4 \times 31 + 3 \times 2 = 130$ ticks and the pipeline has a finite worst-case duration, which is what makes a budget on the enclosing supervisor meaningful. Theorem 4.4 makes the retries well defined because both backoffs are positive. Theorem 9.2 applies to the first stage when the fetch carries an idempotency key that the external service honours, and fails for the third stage if the store counts writes rather than deduplicating them, which is Theorem 9.3 with the store as the counting environment. Reading the term with the fail-stop lifting inside each retry, as Theorem 9.2 requires, is what makes a crash consume one of the three permitted attempts rather than restarting the count. Theorem 7.3 applies to the second stage: recording the sampled token at each step of Summarize makes the stage an Id agent on replay, so a failure in Publish can be recovered without paying for the model call again, and Theorem 7.4 says which edits to the prompt template are safe.

11.2 A supervised worker group

Let $W_1, \ldots, W_4$ be crawler workers, each checkpointed at every tick, under $\mathsf{Sup}_{\mathsf{one\_for\_one}}$ with budget $(3, 60)$. Theorem 5.3 gives the exact death condition: the group dies at the first tick m at which four restart-triggering ticks fall in the window $(m - 60, m]$. Under Theorem 8.3 the group's erased behaviour up to that point is the behaviour of $W_1 \otimes W_2 \otimes W_3 \otimes W_4$, and by Theorem 8.4 switching to one_for_all changes nothing observable, only the amount of work discarded on each restart. If the workers are not checkpointed, Theorem 8.5 applies to any worker with a nontrivial counter, and the group's output is not a stuttering variant of the crash-free output.

11.3 An N-shaped workflow

Let a produce a manifest, b produce a schema, c validate the manifest and d join the manifest with the schema. The dependencies are $a \to c$, $a \to d$ and $b \to d$. By Theorem 10.9 no arrangement of these four tasks using only sequential composition and the synchronous product has this dependency order, so a workflow engine that offers only those two combinators must either serialize b before c, which delays c for no reason, or start d before a has finished, which is wrong. The graph form of the workflow expresses the intended order exactly, and by Theorem 10.3 its output does not depend

on which of the two admissible interleavings the engine picks, since all four nodes are deterministic and read blocking.

11.4 A replayed model call

Let Ask be an agent over D that emits a prompt, samples a completion from a bounded vocabulary with rational weights, and halts with the parsed result. A log for one run records, on each tick, the input the agent received and the pair consisting of the emitted output and the sampled token. Theorem 7.3(1) makes the replay an Id agent reproducing that run. If the prompt template is then edited so that the sampled token still lies in the support of the new policy, part (2) says the replay is still correct; if the edit removes the token from the support, the replay fails with reason nondeterminism on exactly the tick where the histories diverge, which is the diagnostic a durable engine reports.

12 Prior art

The material of this Part exists in four literatures that rarely cite each other, and the contribution here is a common algebraic setting rather than a new result inside any of them.

Failure models. Schlichting and Schneider [13] introduce the fail-stop processor, whose defining properties are that it halts rather than performing an incorrect transition, that its volatile state is lost while its stable state survives, and that its halt is detectable. Theorem 3.1 is a discrete-time reading of exactly these three properties, with the crash port supplying detectability by construction. Schneider [14] shows how deterministic replicas plus agreement on inputs yield fault-tolerant services, which is the reason Theorem 8.3 needs deterministic children; the theorem is the single-replica shadow of that argument. Fischer, Lynch and Paterson [6] bound what any such model can promise, and their Theorem 1 is why fairness is a hypothesis in Theorem 3.3 rather than a lemma. Byzantine models are outside the scope of this Part entirely, and Lynch [11] is the reference for what changes when they are admitted.

Supervision. Armstrong [1, 2] developed supervision trees, the escalation discipline and the restart budget, and the design has been reimplemented in most agent orchestration frameworks without a semantics. Theorem 5.4 makes the correspondence of the three restart sets precise as a one-way simulation, and Theorem 8.4 isolates the condition, checkpointing at every tick, under which the choice among them is invisible.

Checkpointing and rollback. Elnozahy, Alvisi, Wang and Johnson [5] classify rollback-recovery protocols into checkpoint-based and log-based families, with coordinated, uncoordinated and communication-induced variants. Theorem 6.1 is the coordinated case with a synchronous cut, and Theorem 6.3 marks the boundary at which the uncoordinated case begins, where the marker algorithm of Chandy and Lamport [4] and the happens-before order of Lamport [9] are the standard tools. Nothing in this Part addresses the domino effect of uncoordinated checkpointing.

Durable execution and dataflow. Burckhardt and coauthors [3] give the operational semantics of Azure Durable Functions and prove their record-and-replay execution model equivalent to a high-level model; Theorem 7.3 is the same statement over an arbitrary commutative effect monad, obtained by substituting the choice component of Part I's factorization rather than the result of an external call. Kahn [8] proves determinacy of process networks with unbounded channels and blocking reads, and Lee and Parks [10] give the engineering account, including the scheduling conditions that a bounded-memory implementation must satisfy. Van der Aalst and coauthors [16] catalogue the control-flow patterns that workflow systems support, and Theorem 10.9 explains why the catalogue cannot be generated by sequence and parallel split alone. Murata [12] surveys Petri

nets, the main alternative foundation for workflow semantics; the choice of Kahn semantics here buys determinism from the acyclicity and determinism of nodes, where a Petri net formulation would need separate liveness and safety side conditions. Valdes, Tarjan and Lawler [15] supply the series-parallel classification. Helland [7] states the idempotence discipline that Theorem 8.1 formalizes.

13 Limitations

The central claim of the series is quoted again here, because a limitations section that does not restate what is being claimed cannot limit it.

> Part I fixes the object: discrete time, an effect monad T drawn from the list Id, P, D, and agents as effectful Mealy coalgebras with typed termination and failure as a terminal outcome. Parts II to IV introduce the operators under additional assumptions stated in each Part and prove, for each operator, that it is well-defined on agents, which relation it respects, and which laws hold at that relation: Part II for ;, $\oplus$, $\otimes$, $\triangleright$ and μ; Part III for Tr, channels, $\|$ and Gate_H; Part IV for Sup, Persist and Replay; Part V introduces no operator and interprets policies, plans, options, shields and controllers as terms of Parts I to IV, proving properties of their closed loops. For each of sixteen fields the series gives an explicit translation of one named mechanism into a term of the algebra together with an adequacy result for that translation, which is a soundness theorem, a one-way simulation, or a proved limitation; no field is claimed to reduce to the algebra.

This Part proves the clause about Sup, Persist and Replay under four assumptions absent from Parts I to III: crashes occur only at tick boundaries, a crash tick is isolated so that components other than the ones being reset receive the idle input, the children of a supervisor are deterministic and do not fail internally, and the environment is quiescent on idle output. The results are refinement statements over marker erasure rather than congruence statements, and none of Sup, Persist and Replay is claimed to respect bisimilarity exactly.

The remaining limitations are specific.

The failure model is fail-stop. A component that emits a wrong output before crashing, or that continues to emit after it should have stopped, is outside Theorem 3.1, and the supervision results say nothing about it. Byzantine faults, network partitions and clock drift are all excluded.

Isolation is a strong assumption. Theorem 8.6 shows that removing it breaks Theorem 8.3 even with checkpointing at every tick, and describes the channel-based alternative. Whether an unisolated version of the theorem holds under a weaker hypothesis, for example that all inter-component traffic goes through channels whose contents survive a crash, is open.

Checkpointing is treated only at synchronous cuts. The product law of Theorem 6.2 needs the crash schedule to hit both factors together, and Theorem 6.3 shows the law fails otherwise. Consistent global cuts in the sense of Chandy and Lamport are not modelled, and neither is the question of which cut set is the cheapest one that preserves Theorem 8.3.

Supervision is proved for deterministic children that do not fail internally. A child over P or D makes the crash-free reference system itself nondeterministic, and the two-sided refinement of Theorem 8.3 would have to be restated as an inclusion of trace sets with a matching argument for the choices, which this Part does not attempt. A child that fails internally is restarted by the supervisor while the reference product fails, so the two systems differ by construction.

The Erlang correspondence is narrow. Theorem 5.4 matches restart sets and nothing else. Permanent, transient and temporary child specifications, the ordered shutdown with per-child

timeouts, dynamic addition and removal of children, and the significant-child rules of recent releases are all unmodelled, and a run of a real supervisor is simulated by Sup_σ only in the restart dimension.

The Kahn theorem is proved for unbounded channels and checked, in the code layer, only for capacities at least the tick budget. A finite-capacity test cannot validate the unbounded statement, and Theorem 10.4 records that finite capacity leaves the fragment. Cyclic networks are Kahn's theorem and are not reproved here.

The dependency-shape results assume duration-flexible leaves with private input ports. Without that hypothesis particular durations can make an N-shaped workflow agree with a product-and-sequence term, as the pipeline in Section 11 would if every task took one tick. The results also cover only terms over ; and $\otimes$ under the early-start discipline; whether adding race and $\triangleright$ enlarges the class of expressible dependency orders is open, and so is the corresponding question for the channel discipline, where a node's activity depends on the scheduler as well as on the graph.

The two workflow disciplines are not compared. Theorem 10.3 is about the channel discipline and the shape results are about the early-start discipline, and this Part proves nothing that relates the two. A theorem saying that the channel discipline refines the early-start one inside $\mathsf{Wf}_{\mathrm{KPN}}$ looks plausible and is not attempted here.

Time is discrete and global. Every result counts ticks, and no result survives translation to a model with independent local clocks without an argument that this Part does not give. The log is assumed durable, which pushes the hardest part of a real durable execution system, the atomicity of appending to the log and acknowledging an effect, outside the model.

14 Conclusion

The execution layer of this Part rests on four assumptions: crashes land between ticks, a crash tick is isolated, supervised children are deterministic, and the environment does not move when nothing is said to it. Under them, replay against a complete choice log is deterministic for every one of the three effect monads and is correct for a modified agent exactly when the log stays compatible with it; supervision with a checkpoint at every tick refines the crash-free run in both directions after marker erasure, and loses that property without the checkpoint; retry against a deduplicating environment gives effectively-once effects, and gives duplicates against a counting one; and deterministic acyclic workflows are schedule-independent under fairness, while their dependency orders exceed what sequence and product express.

The three negative results locate one failure in three places. Supervision without persistence is not transparent, retry without deduplication is not effectively once, and the N-shaped dependency is not a term. In each case a design that looks compositional is not, and in each case the separating run was found by stating the comparison at the relation the operators respect and then searching for a witness.

Section 13 lists what a fifth assumption might settle: an unisolated supervision theorem, checkpointing at asynchronous cuts, supervision of nondeterministic children, and the dependency orders reachable once race and recovery are allowed inside a workflow node.

A Code evidence

The accompanying code layer is a finite model, not a proof assistant. Every row below is a finite-model check over generated finite terms and agents, or an exhaustive bounded check over all inputs, schedules and crash sequences up to a stated bound. State spaces are finite, tick budgets are bounded, recursion depth is bounded, channel capacities are finite and at least the tick budget, and

the distribution monad uses rational weights. Each row states the fragment its check covers, and every part of a claim that lies outside that fragment rests on its proof; a finite-capacity check never validates an unbounded-channel statement, and a check over one effect monad never validates a statement about three. The checks rule out the cheap ways of being wrong. They are not verification, and none of the theorems above depends on them.

Table 4: Code evidence for the results of Part IV. The claim column gives the short label; the full label is the kind followed by `execution-semantics:` and the short name. Files are under `code/agent-algebra/src` and tests under `code/agent-algebra/test`.

Claim (label)	File and identifier	Test (file › describe › it)	Runs	What the test shows
`thm:` replay-determinism	`ops.ts` `replay`; `rules.ts` `stepReplay`; `sim.ts` `run`	`part4-execution-semantics.test.ts` › replay-determinism › replay of a logged stochastic run reproduces the state trajectory and outputs on 1000 seeds, and › replay-determinism › a run is a pure function of the seed, the scheduler and the fault schedule	1000 seeds	Finite-model check on 1000 seeds that the replay of a logged distribution run reproduces the logged output word and the final configuration, on 200 seeds for the powerset monad, and that a run is a function of the seed, the scheduler and the fault schedule. The per-tick trajectory statement and the compatibility clause of Theorem 7.3 rest on its proof.
`thm:` kahn-determinism	`kahn.ts` `randomDag`, `withMergeNode`; `ops.ts` `workflow`; `sim.ts` `allFair`	`part4-execution-semantics.test.ts` › kahn-determinism › random DAG workflows give identical outputs under 100 fair schedulers	100 schedules	Finite-model check of the bounded fragment only: channels are finite with capacity at least the tick budget, so this check does not validate the unbounded-channel theorem, which rests on its proof.
`thm:` kahn-determinism, boundary	`kahn.ts` `withMergeNode`; `sim.ts` `allFair`	`part4-execution-semantics.test.ts` › kahn-determinism › a nondeterministic merge node yields at least two outputs	sampled schedules	Finite-model check on the sampled fair schedules that a merge node yields at least two outputs, so that leaving the Kahn fragment loses schedule independence. This is the boundary recorded in Theorem 10.4 and not an exhaustive search over schedules.

Claim (label)	File and identifier	Test (file › describe › it)	Runs	What the test shows
`thm:` supervision-refinement	`ops.ts sup`, `persist`; `semantics.ts recoverT`, `restartT`; `equiv.ts eraseMarkers`, `refinesTraces`	`part4-execution-semantics.test.ts` › supervision-refinement › supervision with Persist every tick refines the crash-free run up to stuttering after erasing crash and restart markers	all faults to tick 5	Exhaustive bounded check over every fault schedule on the first five ticks that the erased supervised word refines the crash-free word up to stuttering. The code appends the fault marker to the tick that carries it rather than replacing the tick's output, so the check covers the refinement and not the marker-only crash tick of Theorem 3.1.
`cex:` restart-without-persist	`ops.ts sup`; `sim.ts run`	`part4-execution-semantics.test.ts` › supervision-refinement › a counter child under one_for_one without Persist diverges from the crash-free run	fixed witness	Exhaustive bounded check that the counter child of Theorem 8.5 repeats its first output after the restart and produces a word outside the stutter closure of the crash-free word.
`thm:` effectively-once	`ops.ts retry`, `recover`, `delay`; `envs.ts idempotentEnv`	`part4-execution-semantics.test.ts` › effectively-once › retry against an environment idempotent on the dedup key ends in the crash-free environment state	5 counts, 3 backoffs	Exhaustive bounded check over retry counts one to five and backoffs one to three that the environment state reached by the retried run is the state reached by the crash-free run. The code drives the retries by internal failures of the attempt rather than by injected crashes, and the effect-trace refinement of Theorem 9.2 rests on its proof.
`cex:` duplicate-effects	`ops.ts retry`; `envs.ts counterEnv`	`part4-execution-semantics.test.ts` › effectively-once › retry against a counter environment shows duplicates	4 counts	Exhaustive bounded check over retry counts two to five that dropping the deduplication key makes the counter environment record the reissued requests, which is Theorem 9.3.
`prop:` timeout	`ops.ts timeout`, `race`, `choiceB`, `failT`; `leaves.ts timerLeaf`; `rules.ts statusRace`	`part4-execution-semantics.test.ts` › timeout-bound › timeout_t(A) halts or fails by tick t+1 for every A	100 terms	Finite-model check on 100 generated Id terms that the derived timeout term is not running after tick $t+1$. The code layer uses the same guarded-choice form as Theorem 4.1 and for the same reason. The P and D instances of Theorem 4.2 rest on its proof.

Claim (label)	File and identifier	Test (file › describe › it)	Runs	What the test shows
`prop:` persist-commutes	`ops.ts` `persist`, `seq`; `rules.ts` `stepPersist`, `replayFrom Checkpoint`; `equiv.ts` `relabel`, `eraseMarkers`	`part4-execution-semantics. test.ts` › persist-commutes › Persist commutes with seq up to bisimilarity on small agents	100 pairs	Finite-model check of clause (1) of Theorem 6.2 on 100 random crash-free pairs, together with one fixed pair in which a crash inside the first phase gives the same erased word on both sides. Crashes after the handoff, the product clause and the asynchronous case are not checked, and the hypothesis that the cut set contains the halting boundary is the reason, as Theorem 6.3 records.
`prop:` series-parallel-shape and `cex:` n-dag	`kahn.ts` `seriesParallel`, `termDag`, `nDag`	`part4-execution-semantics. test.ts` › series-parallel › the recognizer accepts every tensor/seq term DAG and rejects the N-DAG	100 terms	Finite-model check on 100 generated shapes of depth three that every product and sequence term has a series-parallel dependency graph, with a fixed check that the N graph and two further graphs do not. The converse direction, and Theorem 10.9 itself, rest on their proofs.
`prop:` restart-budget	`ops.ts` `sup`; `rules.ts` `restartSet`, `statusSup`; `semantics.ts` `statusT`	`part4-execution-semantics. test.ts` › restart-budget › dead-state entry coincides with budget exhaustion under random crash sequences	100 triples	Finite-model check on 100 random triples of budget and failure period, with a single periodically failing child, that the observed dead tick equals the condition of Theorem 5.3 computed from the observed failure ticks. Simultaneous failures of several children are not exercised.
`prop:` otp-strategies	`ops.ts` `sup`; `rules.ts` `restartSet`	`part4-execution-semantics. test.ts` › restart-budget › one_for_one, one_for_all and rest_for_one restart exactly the OTP restart sets	5 events	Finite-model check on five fixed stopping events, with one supervisor witness, that the three restart functions of Theorem 5.1 return the three Erlang restart sets. The simulation over all stopping sequences claimed by Theorem 5.4 rests on its proof.

References

[1] Armstrong, J. (2003). Making reliable distributed systems in the presence of software errors. PhD thesis, KTH Royal Institute of Technology, Stockholm.

[2] Armstrong, J. (2010). Erlang. Communications of the ACM 53(9):68-75.

[3] Burckhardt, S., Gillum, C., Justo, D., Kallas, K., McMahon, C., Meiklejohn, C. (2021). Durable functions: semantics for stateful serverless. Proceedings of the ACM on Programming Languages 5(OOPSLA), article 133. DOI: 10.1145/3485510.

[4] Chandy, K.M., Lamport, L. (1985). Distributed snapshots: determining global states of distributed systems. ACM Transactions on Computer Systems 3(1):63-75. DOI: 10.1145/214451.214456.

[5] Elnozahy, E.N., Alvisi, L., Wang, Y.M., Johnson, D.B. (2002). A survey of rollback-recovery protocols in message-passing systems. ACM Computing Surveys 34(3):375-408.

[6] Fischer, M.J., Lynch, N.A., Paterson, M.S. (1985). Impossibility of distributed consensus with one faulty process. Journal of the ACM 32(2):374-382. DOI: 10.1145/3149.214121.

[7] Helland, P. (2012). Idempotence is not a medical condition. ACM Queue 10(4). DOI: 10.1145/2160718.2160734.

[8] Kahn, G. (1974). The semantics of a simple language for parallel programming. Information Processing 74, Proceedings of the IFIP Congress, pages 471-475, North-Holland.

[9] Lamport, L. (1978). Time, clocks, and the ordering of events in a distributed system. Communications of the ACM 21(7):558-565. DOI: 10.1145/359545.359563.

[10] Lee, E.A., Parks, T.M. (1995). Dataflow process networks. Proceedings of the IEEE 83(5):773-801.

[11] Lynch, N.A. (1996). Distributed Algorithms. Morgan Kaufmann, San Francisco.

[12] Murata, T. (1989). Petri nets: properties, analysis and applications. Proceedings of the IEEE 77(4):541-580.

[13] Schlichting, R.D., Schneider, F.B. (1983). Fail-stop processors: an approach to designing fault-tolerant computing systems. ACM Transactions on Computer Systems 1(3):222-238.

[14] Schneider, F.B. (1990). Implementing fault-tolerant services using the state machine approach: a tutorial. ACM Computing Surveys 22(4):299-319. DOI: 10.1145/98163.98167.

[15] Valdes, J., Tarjan, R.E., Lawler, E.L. (1982). The recognition of series parallel digraphs. SIAM Journal on Computing 11(2):298-313.

[16] van der Aalst, W.M.P., ter Hofstede, A.H.M., Kiepuszewski, B., Barros, A.P. (2003). Workflow patterns. Distributed and Parallel Databases 14(1):5-51. DOI: 10.1023/A:1022883727209.