Channels, Actors, and Shared State

Part III of *An Algebra of Agents*

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

September 2026

Abstract

Agents that interact do so through wires: feedback loops, message queues, shared stores, and the approval channel that connects a system to a person. This Part fixes the wiring layer of an algebra of agents whose object and operators are settled in Parts I and II. A single feedback operator with a one-tick register generates the rest. We prove that agents with the synchronous product and this operator satisfy naturality, vanishing, superposing and a delayed form of yanking, so the structure is a delayed trace rather than a traced monoidal category, and we exhibit the run that separates the two. The closed loop of Part I is the trace of the product of an agent and its environment. Channels, links, a scheduler and asynchronous interleaving are then derived, and two fragments are kept apart. In the interleaving fragment, where components are coupled only by channels, the accepted-trace set of a parallel composition is the shuffle of the component sets and the concurrent Kleene algebra exchange inclusion holds, strictly, where the synchronous product fails it. In the shared-object fragment, blackboards and tuple spaces are linearizable with tick order, replication breaks linearizability, and tuple-space competition is nondeterministic while a single channel between deterministic agents is not. Binary session typing with fidelity and progress gives deadlock freedom under a fair scheduler, and the human approval gate is a session-typed link that is transparent exactly when approval is automatic.

1 Introduction

An agent that runs alone is a state machine. The engineering interest starts when agents are wired together: an output becomes an input one tick later, a message waits in a queue until its recipient is scheduled, several writers contend for one shared record, and a person is asked to approve an action before it reaches the world. Each of these arrangements has a mature theory of its own. Actors, tuple spaces, blackboards, session types and traced monoidal categories were each developed for a different purpose and are usually presented in different vocabularies. What is missing is a single object on which all of them can be defined, so that a claim proved about one arrangement can be compared with a claim proved about another.

This series supplies that object: Part I fixes an agent as an effectful Mealy coalgebra with typed termination, Part II supplies the composition operators and their laws, and this Part supplies the wiring. The claim the series makes, quoted in every Part, is this.

> Part I fixes the object: discrete time, an effect monad T drawn from the list Id, $\mathcal{P}$, $\mathcal{D}$, 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 ;, $\oplus$, $\otimes$, $\rhd$ 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 claim for Tr, for channels and the derived interleaving operator $\|$, and for the approval gate Gate_H, under three assumptions added here: the feedback register holds its value for exactly one tick, the traced ports are hidden from the interface, and every result about $\|$ is stated either for components coupled only by channels or for a shared object that serves one operation per tick, never for both at once.

1.1 Results

Agents with the synchronous product of Part II and the feedback operator of Section 3 form a delayed-trace structure and not a traced monoidal category. Feedback satisfies naturality in the ports that are not fed back, vanishing and superposing, while the yanking axiom of Joyal, Street and Verity [13] fails on any alphabet with two letters: closing a symmetry wire on itself gives a one-tick register. The register is the reason feedback is defined at all for machines that consume an input before producing an output, and it makes the structure the delayed trace of Sprunger and Katsumata [20].

The closed loop of Part I is that same operator applied to the product of an agent and its environment, with the register initialised at the idle pair. The series has one feedback construction and one latency convention, not two.

Coupling by channels and coupling by a shared object obey different laws, and the two are kept in separate fragments throughout. For components coupled only by channels, the accepted-trace set of $A \| B$ is the shuffle of the accepted-trace sets of A and B, and $\|$, ;, $\oplus$, 0 and skip satisfy the axioms of a concurrent semiring, including the exchange inclusion of Hoare, Möller, Struth and Wehrman [9]. The synchronous product violates that inclusion, by a counterexample of Part II, and the derived interleaving operator satisfies it strictly.

For components that share an object the shuffle law fails and linearizability takes its place. Every history of a blackboard under $\|$ is linearizable with tick order; a replicated blackboard with asynchronous propagation has a history that is not; a tuple space with two consumers competing for one tuple has at least two outcomes, where a single channel between deterministic agents has exactly one.

Two agents whose session types are dual and whose only coupling is the channel between them never reach a state in which every party is blocked, and the linked system halts exactly when the session reaches its end. Typing has two clauses, fidelity and progress, and both are properties of the port traces of a single agent.

The approval gate is a channel to a co-agent that answers ok or no, with the agent frozen while it waits. No action leaves the gate before an approval of that action, and under an oracle that always approves, the gate and the agent it wraps refine each other, so approval costs exactly the idle ticks it introduces.

1.2 Relation to prior work

Feedback in a symmetric monoidal category is axiomatised by a trace operator satisfying naturality, dinaturality, vanishing, superposing and yanking [13]. Causal stream functions carry a delayed trace instead, a weaker operator that misses the yanking and dinaturality axioms of the usual trace, and that weakening is what makes the feedback guarded [20]. Transporting the delayed trace to effectful Mealy coalgebras with halting requires proving the axioms one at a time rather than inheriting them, since neither the halting outcome nor the effect monad appears in the stream-function setting. Katis, Sabadini and Walters [15] treat feedback bicategorically, an alternative route not taken here.

The coordination constructs are older than the categorical ones. Actors are due to Hewitt, Bishop and Steiger [8] and were given their standard operational treatment by Agha [1] and by Agha, Mason, Smith and Talcott [2]; the property this Part proves for them, that an actor's state depends only on its initial value and on the sequence of messages it dequeued, is the formal content of the isolation that the actor literature states informally. Generative communication in Linda is due to Gelernter [5]; blackboard architectures to Erman, Hayes-Roth, Lesser and Reddy [4] and Hayes-Roth [6]. Linearizability is the correctness condition of Herlihy and Wing [7], and it is what survives when the shuffle law does not. Session types are due to Honda [10] and Honda, Vasconcelos and Kubo [11], with the logical reading given by Caires and Pfenning [3] and Wadler [21]; the multiparty theory of Honda, Yoshida and Carbone [12] is outside the scope of this Part and is discussed in Section 10.

What this Part adds to those sources is not a new coordination primitive. It is a single object on which all of them are terms, together with the observation that the two coupling disciplines obey different laws: the shuffle and the exchange inclusion in the channel fragment, linearizability with tick order in the shared-object fragment, and no theorem that mixes them.

2 Setting

2.1 Imported foundations

This Part defines no agent, no equivalence and no composition operator. It uses the objects listed in Table 1 exactly as the earlier Parts fix them.

Three conventions of the frozen foundations do most of the work below. Halting and failing consume no tick and are visible in $\sim$, because halt is part of what a bisimulation preserves. A state that is not running emits τ on every input and stays where it is, so a halted component inside a wired system contributes idle ticks and nothing else. The idle symbols ε and τ are the ones the stutter closure inserts and deletes, so a tick that consumes ε and emits τ is invisible to $\sqsubseteq$.

2.2 Processes and the wiring category

Composition in Part II moves values along a handoff: $A\,;B$ starts B at the value A halted with. Wiring moves symbols along a port within one tick, a different operation on a different dimension of the same object, and the trace axioms are statements about that second dimension. The category $\mathcal{W}_T$ below has the ports as its objects and the port-preserving processes as its morphisms.

Definition 2.1 (Process)**.** A *process* over an interface (I, Σ) is an agent of sequential type $1 \Rightarrow 1$ whose halt map is constantly $\mathrm{inr}(*)$: it never halts and never fails. Processes are written $A : I \to \Sigma$.

Channels, registers, shared objects and routing wires are processes. Agents that compute a value and stop are not; they enter a wired system as components whose halting is observed by the components around them.

Table 1: Objects imported from Parts I and II. This Part cites them and does not restate their definitions; the numbers refer to the sibling manuscripts.

Object	Source	What it fixes
Interface (I, Σ)	Part I, Definition 2.1	Pointed input and output alphabets with idle symbols ε and τ, and the product interface.
Effect monad T	Part I, Definition 2.2	One of Id, $\mathcal{P}$ (finite nonempty subsets), $\mathcal{D}$ (finite-support rational distributions); commutative and weak-pullback preserving.
Agent $A : X \Rightarrow Y$	Part I, Definition 2.10	The tuple $(S, \mathrm{init}, \mathrm{step}, \mathrm{halt})$ with $\mathrm{step} : S \times I \to T(\Sigma \times S)$ and $\mathrm{halt} : S \to Y + E + 1$, together with the rule that a non-running state stutters.
Trace semantics $[\![A]\!]$	Part I, Definition 3.2	The map $X \times I^* \to T(\Sigma^* \times (Y + E + 1))$ and its prefix compatibility.
Accepted traces and $\simeq_{\mathrm{acc}}$	Part I, Definition 3.5	The runs whose outcome is $\mathrm{inl}(y)$, and equality of the resulting sets or distributions.
Stutter closure and $\sqsubseteq$	Part I, Definition 5.3, Part I, Definition 5.4	Insertion and deletion of idle ticks, and inclusion of stutter-closed trace sets.
Bisimilarity $\sim$	Part I, Definition 4.1	Relation lifting of T, preservation of halt, initial states related.
Observation policy	Part I, Definition 5.2	Reserved marker outputs, hiding of consumed ports, and fairness of a scheduler.
Environment and closed loop	Part I, Definition 9.1, Part I, Definition 9.2	The co-agent E over (Σ, I) and the closed system $\mathrm{Run}(A, E)$ on $S_A \times S_E \times I \times \Sigma$ with register initialised at (ε, τ).
$\otimes$, ;, $\oplus$, 0, skip, μ	Part II, Definition 4.14, Part II, Definition 4.6, Part II, Definition 4.12, Part II, Definition 4.2, Part II, Definition 4.1, Part II, Definition 4.20	The synchronous product and its unit, sequential composition, the choice family, divergence, the identity of ;, and guarded recursion.

Definition 2.2 (Pipelining)**.** For processes $A : I \to \Sigma$ and $B : \Sigma \to \Theta$, the process $A \gg B : I \to \Theta$ has state $S_A \times S_B$, initial state $(\mathrm{init}_A, \mathrm{init}_B)$, and

$$\mathrm{step}_{A \gg B}\big((a, b), i\big) \;=\; \mathrm{step}_A(a, i) \gg\!= \lambda(\sigma, a').\, \mathrm{step}_B(b, \sigma) \gg\!= \lambda(\theta, b').\, \eta(\theta, (a', b')),$$

where $\gg\!=$ is the bind of T and η its unit. The *copy wire* $\mathrm{id}_I : I \to I$ has a one-element state and $\mathrm{step}(*, i) = \eta(i, *)$.

Pipelining is composition inside a tick: B sees at tick t what A emitted at tick t. It is not ; of Part II, and the copy wire is not skip of Part II, which halts immediately with its start value and is therefore not a process.

Definition 2.3 (Wiring category)**.** $\mathcal{W}_T$ has as objects the pointed sets, as morphisms $I \to \Sigma$ the processes over (I, Σ) modulo $\sim$, composition $\gg$, identities the copy wires, and monoidal product the restriction of $\otimes$ of Part II to processes, with unit the one-element alphabet 1 and symmetry the stateless wire $\mathrm{swap}_{I,J} : I \times J \to J \times I$ that exchanges its two input components each tick.

Lemma 2.4 (Wiring is symmetric monoidal)**.** *For every $T \in \{\mathrm{Id}, \mathcal{P}, \mathcal{D}\}$, $\mathcal{W}_T$ is a symmetric monoidal category, and $\sim$ is a congruence for $\gg$ and for $\otimes$ on processes.*

Proof. Associativity of $\gg$ up to $\sim$ is the associativity of Kleisli binding together with the evident bijection $(S_A \times S_B) \times S_C \cong S_A \times (S_B \times S_C)$, which is a bisimulation because it commutes with step and both sides never halt. The unit laws use $S \times 1 \cong S$ in the same way. Congruence for $\gg$: if R relates A and A' and R' relates B and B', then $R \times R'$ is a bisimulation for the composites, since relation lifting of T is compatible with binding (Part I, Lemma 4.2 and the lifting lemma cited there): the first bind sends R-related states to $\mathrm{Rel}(T)(R)$-related distributions of pairs, and the second bind preserves that. For $\otimes$, associativity, commutativity and unit are Part II, Theorem 10.1, and congruence is the $\otimes$ case of Part II, Theorem 5.1. Bifunctoriality, that is $(A \gg B) \otimes (C \gg D) \sim (A \otimes C) \gg (B \otimes D)$, holds because both sides perform the same four leaf steps on the same inputs and differ only in the order in which the two independent binds are taken, which commutativity of T makes irrelevant. Symmetry: swap is stateless and self-inverse, and the hexagon and naturality squares reduce to equalities of tuples of symbols. □

Remark 2.5. The wiring category has no halting, so it has no value dimension. This is deliberate. A statement such as "feedback is a trace" is about ports, and the halting behaviour of the components involved plays no part in it. The two dimensions meet again in Section 4, where the components of an interleaved system halt and the system halts when they all do.

3 Delayed feedback

Feedback is the operation that turns an output port into an input port. For machines that read before they write, it cannot be instantaneous: an agent whose output at tick t depended on its own output at tick t would not be a Mealy coalgebra. The construction that makes feedback total inserts a register.

Definition 3.1 (Register)**.** Let C be a pointed set and $c_0 \in C$. The *one-tick register* $\mathrm{R}_{C,c_0} : C \to C$ is the process with state C, initial state c_0, and $\mathrm{step}(c, c_{\mathrm{in}}) = \eta(c, c_{\mathrm{in}})$: it emits the value it holds and stores the value it receives.

The register emits c_0 at the first tick and, at every later tick t, the symbol it received at tick $t-1$. We write delay_1 for R_{C,c_0} when the alphabet and the initial value are clear. This is a wire and is not the agent delay_d of Part II, Definition 4.4, which halts after d ticks with its start value.

Definition 3.2 (Delayed trace)**.** Let A be an agent of type $X \Rightarrow Y$ over the interface $(I \times C, \Sigma \times C)$ and let $c_0 \in C$. The agent $\mathrm{Tr}^{C,c_0}(A)$ of type $X \Rightarrow Y$ over (I, Σ) has state $S_A \times C$, initial state $\mathrm{init}_{\mathrm{Tr}(A)}(x) = (\mathrm{init}_A(x), c_0)$, halting map $\mathrm{halt}_{\mathrm{Tr}(A)}(s, c) = \mathrm{halt}_A(s)$, and

$$\mathrm{step}_{\mathrm{Tr}^{C,c_0}(A)}\big((s,c),i\big) \;=\; \mathrm{step}_A\big(s,(i,c)\big) \gg\!= \lambda\big((\sigma,c'),s'\big).\ \eta\big(\sigma,(s',c')\big).$$

The port C is consumed and, by the observation policy of Part I, hidden: it appears in neither the interface nor the traces of $\mathrm{Tr}^{C,c_0}(A)$.

The C-symbol produced at tick t is the C-symbol consumed at tick $t+1$, and the symbol consumed at the first tick is c_0. Nothing in the definition requires A to be a process; Tr acts on agents of any sequential type and leaves halting untouched. The axioms below are stated in $\mathcal{W}_T$ because they involve composition in the port dimension, which is $\gg$.

Lemma 3.3 (Congruence)**.** *If $A \sim A'$ as agents over $(I \times C, \Sigma \times C)$ then $\mathrm{Tr}^{C,c_0}(A) \sim \mathrm{Tr}^{C,c_0}(A')$. Consequently $\sim$ is a congruence for every operator built from $\otimes$, $\gg$ and* Tr*, which includes* Ch_k*-linking,* link_c*, $\|$ and* Act *below.*

Proof. Let $R \subseteq S_A \times S_{A'}$ be a bisimulation with $(\mathrm{init}_A x, \mathrm{init}_{A'} x) \in R$ for every x. Put $\widehat{R} := \{((s,c),(s',c)) : (s,s') \in R,\ c \in C\}$. Initial states are $\widehat{R}$-related by construction, and halt is preserved because $\mathrm{halt}_{\mathrm{Tr}(A)}$ ignores the register. For the transfer condition, fix $((s,c),(s',c)) \in \widehat{R}$ and an input i. Since $(s,s') \in R$, the two distributions $\mathrm{step}_A(s,(i,c))$ and $\mathrm{step}_{A'}(s',(i,c))$ are related by $\mathrm{Rel}(T)$ applied to the relation "equal output, R-related successor" on $(\Sigma \times C) \times S$. Relation lifting commutes with binding along the function $((\sigma,c'),s') \mapsto (\sigma,(s',c'))$, which is the graph collapse lemma of Part I, so the images under the bind of Definition 3.2 are related by $\mathrm{Rel}(T)(\widehat{R})$. The second claim follows from Lemma 2.4 by induction on the term. □

Lemma 3.4 (Sliding along a relabelling)**.** *Let $A : I \otimes U \to \Sigma \otimes V$ be a process, let $h : V \to U$ be a pointed function, and write h also for the stateless wire it induces. If $u_0 = h(v_0)$ then*

$$\mathrm{Tr}^{U,u_0}\big(A \gg (\mathrm{id}_\Sigma \otimes h)\big) \ \sim\ \mathrm{Tr}^{V,v_0}\big((\mathrm{id}_I \otimes h) \gg A\big).$$

Proof. Relate (s,u) on the left to (s,v) on the right whenever $u = h(v)$; the initial states are related because $u_0 = h(v_0)$. On input i the left side runs A on (i,u) and stores $h(v')$ where v' is the V-component of A's output; the right side maps its register through h, runs A on $(i,h(v)) = (i,u)$, and stores v'. The two runs of A are the same computation with the same T-weights, the emitted Σ-symbols agree, and the successors are again related. □

The lemma is the only form of sliding claimed here. Sprunger and Katsumata [20] state that the delayed trace misses the dinaturality axiom of the usual trace, and nothing below repairs it: sliding a stateful component across the register changes where the delay sits.

Theorem 3.5 (Delayed-trace axioms)**.** *Let $T \in \{\mathrm{Id}, \mathcal{P}, \mathcal{D}\}$. In $\mathcal{W}_T$ the operator* Tr *of Definition 3.2 satisfies the following, in each case up to $\sim$.*

(i) Naturality in the ports that are not fed back: for processes $f : I' \to I$, $g : \Sigma \to \Sigma'$ and $A : I \otimes C \to \Sigma \otimes C$,

$$\mathrm{Tr}^{C,c_0}\big((f \otimes \mathrm{id}_C) \gg A \gg (g \otimes \mathrm{id}_C)\big) \ \sim\ f \gg \mathrm{Tr}^{C,c_0}(A) \gg g.$$

(ii) Vanishing: $\mathrm{Tr}^{1,*}(A) \sim A$ for $A : I \otimes 1 \to \Sigma \otimes 1$, and

$$\mathrm{Tr}^{C \times D,\,(c_0,d_0)}(A) \ \sim\ \mathrm{Tr}^{C,c_0}\big(\mathrm{Tr}^{D,d_0}(A)\big)$$

for $A : I \otimes C \otimes D \to \Sigma \otimes C \otimes D$, where the left side reads $C \times D$ as one register.

(iii) Superposing: for $A : I_1 \otimes C \to \Sigma_1 \otimes C$ and $B : I_2 \to \Sigma_2$,

$$\mathrm{Tr}^{C,c_0}(A) \otimes B \ \sim\ \mathrm{Tr}^{C,c_0}\big(\alpha \gg (A \otimes B) \gg \beta\big),$$

where α and β are the stateless symmetry wires that move the C port to the right of the I_2 and Σ_2 ports.

(iv) Delayed yanking: $\mathrm{Tr}^{C,c_0}(\mathrm{swap}_{C,C}) \sim \mathrm{R}_{C,c_0}$.

Proof. Each clause is an explicit bisimulation. Throughout, halt is constant on processes, so only the transfer condition needs checking, and by Lemma 3.3 it suffices to exhibit a relation on states closed under step.

(i) The left side has state $(S_f \times S_A \times S_g) \times C$ and the right side $S_f \times (S_A \times C) \times S_g$. Relate $((\varphi, s, \gamma), c)$ to $(\varphi, (s, c), \gamma)$. On input i, both sides run f on i, obtaining i' and φ'; then A on (i', c), obtaining (σ, c') and s'; then g on σ, obtaining σ' and γ'. The three binds occur in the same order on both sides, so no commutativity of T is used; the emitted symbol is σ' on both sides and the successors are again related.

(ii) For the unit, $S_A \times 1 \cong S_A$ and the register carries no information. For the second equation, relate $(s, (c, d))$ on the left to $((s, d), c)$ on the right. On input i, the left side runs A on (i, c, d) and stores the pair (c', d') that A emitted; the right side runs the inner trace, which supplies d from its register and stores d', inside the outer trace, which supplies c and stores c'. Both compute $\mathrm{step}_A(s, (i, c, d))$ once and store the same pair.

(iii) Relate $((s, c), b)$ to $((s, b), c)$. On input (i_1, i_2), the left side runs A on (i_1, c) and B on i_2 and forms the pair of outputs; the right side runs $A \otimes B$ on $\alpha(i_1, i_2, c) = ((i_1, c), i_2)$ and reassociates the outputs by β. Both sides therefore take the same two independent binds in opposite orders, and commutativity of T makes the resulting elements of $T(\Sigma_1 \times \Sigma_2 \times C \times S_A \times S_B)$ equal; this is the bifunctoriality step of Lemma 2.4.

(iv) $\mathrm{swap}_{C,C}$ is stateless, so $\mathrm{Tr}^{C,c_0}(\mathrm{swap})$ has state $1 \times C \cong C$ with initial value c_0. On input i at register value c, the wire receives (i, c) and emits (c, i), so the visible output is c and the new register value is i. That is exactly $\mathrm{step}_{\mathrm{R}_{C,c_0}}(c, i) = \eta(c, i)$, and the initial values agree. □

Counterexample 3.6 (Ordinary yanking fails). Let $C = \{c_0, c_1\}$ with $c_0 \neq c_1$ and $T = \mathrm{Id}$. Then $\mathrm{Tr}^{C,c_0}(\mathrm{swap}_{C,C})$ is not bisimilar to the copy wire id_C, so the yanking axiom of Joyal, Street and Verity [13] fails in $\mathcal{W}_T$ for every T.

Proof. On the one-letter input word c_1 the copy wire emits c_1 and $\mathrm{Tr}^{C,c_0}(\mathrm{swap})$ emits c_0 by clause (iv) of Theorem 3.5. The two agents therefore have different trace semantics, and by Part I, Theorem 4.4 bisimilar agents have equal trace semantics; so they are not bisimilar. The same word separates them for $\mathcal{P}$ and $\mathcal{D}$, since both embed the deterministic wires by η. □

Remark 3.7. Theorem 3.5 and Counterexample 3.6 place $(\mathcal{W}_T, \otimes, \mathrm{Tr})$ among the delayed-trace structures of Sprunger and Katsumata [20] and outside the traced monoidal categories of Joyal, Street and Verity [13]. The gap between the two is exactly one tick, and it is not removable: an agent consumes an input before it emits an output, so a value fed back to the same tick would have to be known before it was computed. Every construction in the rest of this Part inherits that tick. A message crosses a channel with latency, an actor sees a message no earlier than the tick after it was sent, and a shared object answers a client no earlier than the client's next scheduled tick.

Proposition 3.8 (The closed loop is a trace)**.** *Let A be an agent of type $X \Rightarrow Y$ over (I, Σ) and let E be an environment co-agent of type $X_E \Rightarrow Y_E$ over (Σ, I). Let $\mathrm{Run}(A, E)$ be the closed system of Part I, Definition 9.2, with state space $S_A \times S_E \times I \times \Sigma$, register initialised at (ε, τ), and both components stepped in lockstep. Then*

$$\mathrm{Run}(A, E) \;\sim\; \mathrm{Tr}^{I \times \Sigma,\ (\varepsilon,\tau)}\big((A \otimes E) \gg \mathrm{swap}_{\Sigma,I}\big),$$

an agent of type $X \times X_E \Rightarrow Y \times Y_E$ over the trivial interface $(1, 1)$, for every $T \in \{\mathrm{Id}, \mathcal{P}, \mathcal{D}\}$.

Proof. $A \otimes E$ is an agent over $(I \times \Sigma, \Sigma \times I)$, and $\mathrm{swap}_{\Sigma,I}$ carries its output alphabet to $I \times \Sigma$, so the composite is an endo-wire on $I \times \Sigma$ and the trace over the whole alphabet leaves the trivial interface.

Its state space is $(S_A \times S_E) \times (I \times \Sigma)$, which is the state space of $\mathrm{Run}(A, E)$ up to the associativity of the product; let θ be that bijection, which carries the initial register value (ε, τ) to the same pair in Part I's presentation. For the transfer condition, fix a state (s, e, i, σ). Both systems run $\mathrm{step}_A(s, i)$ and $\mathrm{step}_E(e, \sigma)$, take their T-product, which is well defined and order-independent because T is commutative, emit nothing on the trivial alphabet, and store the pair of emitted symbols as the next register value, A's output going to E's input port and E's output going to A's input port. The swap wire performs exactly that crossing and is stateless. Halting agrees: Tr inherits halt from its argument, and $\mathrm{halt}_{A \otimes E}$ is inl when both components have halted and $\mathrm{inm}(e)$ when either has failed, which is the halting rule of the closed system. Hence θ is a bisimulation. □

The proposition settles a bookkeeping question for the series: there is one feedback construction, not two, and results proved about Tr apply to every closed loop. It also fixes the latency of a closed loop at one tick, which is the reason an agent cannot react to its own effect on the environment within the tick in which it acts.

4 Channels, objects and interleaving

4.1 Channels

Definition 4.1 (Channel)**.** Let M be a finite message set and $k \in \mathbb{N} \cup \{\infty\}$ a capacity. The channel $\mathrm{Ch}_k(M)$ is the process with input alphabet $I_{\mathrm{Ch}} = (M + \{-\}) \times \{\mathsf{req}, -\}$, idle symbol $(-, -)$, output alphabet $\Sigma_{\mathrm{Ch}} = (M + \{-\}) \times \{\mathsf{ack}, \mathsf{full}, -\}$, idle symbol $(-, -)$, state the words $M^{\leq k}$, initial state the empty word, and

$$\mathrm{step}(q, (m, r)) \;=\; \eta((d, u),\, q''),$$

with the dequeue served from the queue as it stood at the start of the tick and the enqueue appended afterwards. Write $q = m_1 \cdots m_l$. If $r = \mathsf{req}$ and $l \geq 1$ then $d = m_1$ and $q' = m_2 \cdots m_l$; otherwise $d = -$ and $q' = q$. If $m \neq -$ and $|q'| < k$ then $u = \mathsf{ack}$ and $q'' = q'm$; if $m \neq -$ and $|q'| = k$ then $u = \mathsf{full}$ and $q'' = q'$; if $m = -$ then $u = -$ and $q'' = q'$. A dequeue request against an empty queue returns $d = -$, which is the blocking case: the requesting agent learns that nothing was available and may request again.

The channel is deterministic for every T, since its step is η of a function. It never halts, so it is a process in the sense of Definition 2.1 and can be wired.

Definition 4.2 (Linking through an object)**.** Let O be a process over (I_O, Σ_O), let $A_1, \ldots, A_n$ be agents each of whose interface carries an O-port, and let $\odot$ be either $\otimes$ or the interleaving $\parallel$ of Definition 4.5. The system obtained by *linking the family through* O is

$$\mathrm{Tr}^{\Sigma_O,\, \tau_O}\Big((\bigodot_j A_j) \gg \rho \gg (\mathrm{id} \otimes O) \gg \rho' \Big),$$

where ρ is the stateless routing wire that gathers the components' O-port outputs into one input of O and ρ' is the stateless wire that returns O's output to the traced port. For $n = 2$, $\odot = \otimes$ and $O = \mathrm{Ch}_k(M)$, with A writing and B reading, we write $\mathrm{link}_c(A, B)$ and call it a *link*. Its interface is $(I_A \times I_B, \Sigma_A \times \Sigma_B)$; the channel ports are consumed by Tr and hidden.

Unfolding the definitions, $\mathrm{link}_c(A, B)$ has state $S_A \times S_B \times M^{\leq k} \times \Sigma_{\mathrm{Ch}}$ and, writing the register as (d, u),

$$\begin{aligned} \mathrm{step}((a, b, q, (d, u)),\ (i_A, i_B)) \;= {} & \mathrm{step}_A(a, (i_A, u)) \gg\!= \lambda((\sigma_A, m), a'). \\ & \mathrm{step}_B(b, (i_B, d)) \gg\!= \lambda((\sigma_B, r), b'). \\ & \mathrm{step}_{\mathrm{Ch}}(q, (m, r)) \gg\!= \lambda((d', u'), q').\ \eta((\sigma_A, \sigma_B),\ (a', b', q', (d', u'))), \end{aligned}$$

with halt that of $A \otimes B$: the link halts when both components halt and fails when either fails. A message written by A at tick t enters the queue at tick t, can be dequeued at tick $t+1$, and reaches B at tick $t+2$. Nothing in the algebra removes those two ticks; they are the register of Theorem 3.5 counted once for the queue and once for the response path.

4.2 Scheduling

Definition 4.3 (Scheduler agent)**.** For $T = \mathcal{P}$ and $n \geq 1$, the scheduler Sched_n is the process over $(1, \{1, \ldots, n\})$ with a one-element state and $\mathrm{step}(*, \varepsilon) = \{(j, *) : 1 \leq j \leq n\}$: at every tick it emits an index, and every index is available at every tick. A schedule is a trace of Sched_n, and a schedule is fair in the sense fixed by the observation policy of Part I when every component that is enabled infinitely often is scheduled infinitely often. Part IV develops the classes of schedulers and the failure model; this Part uses only the two conditions just named.

Definition 4.4 (Idle quiescence)**.** An agent A is *ε-quiescent* when $\mathrm{step}_A(s, \varepsilon) = \eta(\tau, s)$ for every state s: an idle input produces an idle output and no state change.

Quiescence is what makes "not scheduled" equal to "frozen". It is a property of leaf agents that holds for every construction in this Part, because a component learns about the world only through its ports, and an idle tick carries nothing on any port.

Definition 4.5 (Asynchronous interleaving)**.** Call an agent *schedulable* when it is ε-quiescent, has sequential type $1 \Rightarrow 1$, and has an interface of the form $\big(\{\varepsilon\} \cup (\{\mathsf{run}\} \times P),\ \Sigma\big)$, where run is the scheduling token and P is the agent's port bundle, trivial when the agent has no ports. For schedulable $A_1, \ldots, A_n$ the interleaving $\|_j\, A_j$ is the $T = \mathcal{P}$ agent with port bundle $\prod_j P_j$ and output alphabet $\biguplus_\tau \Sigma_j$, the coproduct of the Σ_j with all idle symbols identified, with state $\prod_j S_j$, with $\mathrm{step}(\vec{s}, \varepsilon) = \eta(\tau, \vec{s})$, and with

$$\mathrm{step}\big((s_1, \ldots, s_n),\ (\mathsf{run}, (p_1, \ldots, p_n))\big) = \bigcup_{j=1}^{n} \Big\{ \big(\iota_j(\sigma),\ (s_1, \ldots, s'_j, \ldots, s_n)\big)\ :\ (\sigma, s'_j) \in \mathrm{step}_j\big(s_j, (\mathsf{run}, p_j)\big) \Big\},$$

where ι_j is the coproduct injection. The composite halts when every component has halted and fails when some component has failed; it is again schedulable, so interleavings nest. For $n = 2$ we write $A \parallel B$.

Exactly one component advances per tick; the others receive ε and, being quiescent, stand still, so the union above is the whole of the composite step. The nondeterminism is the scheduler's, which is why $\|$ is defined for $T = \mathcal{P}$ and not for $\mathcal{D}$: choosing whom to run is not a probabilistic experiment with a distinguished distribution. A component that has already halted contributes τ when it is scheduled, by the stuttering rule of Part I, so no special case is needed for termination.

Definition 4.6 (Object system)**.** A *shared object* is a process O over $(Op_\perp, Res_\perp)$ with $\mathrm{step}(m, \perp) = \eta(\perp, m)$, which applies at most one operation per tick. Given ε-quiescent clients $A_1, \ldots, A_n$ whose port bundle is $P_j = Res_\perp$ and whose output carries an operation slot, the *object system* $\mathrm{Sys}(O; A_1, \ldots, A_n)$ is the linking of the family through O with $\odot =\!\|$ (Definition 4.2). Its state is $\big(\prod_j S_j\big) \times S_O \times Res_\perp^n$, written $(\vec{s}, m, \vec{r})$, and one tick chooses a client j, advances it on the response

r_j held for it, applies the operation it emits to O, and stores the result as the new r_j:

$$\mathrm{step}((\vec{s}, m, \vec{r}), \varepsilon) = \bigcup_j \Big\{ \big(\iota_j(\sigma),\ (\vec{s}[j \mapsto s'_j],\ m',\ \vec{r}[j \mapsto res])\big)\ :\ \big((\sigma, o), s'_j\big) \in \mathrm{step}_j(s_j, (\mathsf{run}, r_j)),\ (res, m') \in \mathrm{step}_O(m, o) \Big\}.$$

A client issues at most one operation before consuming its response, so one response slot per client suffices. A finite family of objects is treated as one object, their tensor, whose operation slot carries the name of the object addressed; at most one object receives a non-idle operation per tick, because at most one client is scheduled.

The operation is applied in the tick in which it is issued, and its result reaches the client at the client's next scheduled tick. That two-part timing is what gives the linearizability statement of Section 6 its content: the interval of an operation spans several ticks, while its effect happens at one of them.

Definition 4.7 (Interleaving fragment)**.** The *interleaving fragment* consists of the schedulable agents built from schedulable leaves by ;, $\oplus$, $\|$ and linking through channel objects, in which every channel has exactly one writing port and one reading port and no two components share any other port. The *shared-object fragment* is the same with channels replaced by objects that serve more than one client. The two fragments are kept apart throughout: the shuffle and exchange laws below are stated for the first, linearizability for the second, and no result is stated for both.

4.3 The shuffle law

A schedulable agent with no ports consumes only ε and run, and its runs are indexed by the ticks at which it was scheduled.

Definition 4.8 (Accepted-trace set of a schedulable agent)**.** For a schedulable agent A with no ports and $T = \mathcal{P}$, let $L_{\mathrm{acc}}(A) \subseteq (\Sigma \setminus \{\tau\})^*$ be the set of words obtained by taking the accepted traces of A on the input words run^n, in the sense of Part I, Definition 3.5, and deleting the letters equal to τ. Deleting a τ letter is the stutter closure of Part I applied to the closed system that drives A with run at every tick, in which the driving component supplies the token and the tick consumes ε from outside. Two agents are identified when their sets agree, and $L_{\mathrm{acc}}(A) \subseteq L_{\mathrm{acc}}(B)$ is the order used below.

Definition 4.9 (Shuffle)**.** For languages L_1, L_2 over an alphabet Δ, $\mathrm{Sh}(L_1, L_2)$ is the set of words w that admit a factorisation $w = w_1 w_2 \cdots w_{2r}$ with $w_1 w_3 \cdots \in L_1$ and $w_2 w_4 \cdots \in L_2$ after deleting empty factors; equivalently, the words obtained by interleaving a word of L_1 with a word of L_2 without reordering either.

Theorem 4.10 (Shuffle)**.** *Let A and B be schedulable agents with no ports, in the interleaving fragment, over output alphabets Σ_A and Σ_B, with $T = \mathcal{P}$. Then*

$$L_{\mathrm{acc}}(A \parallel B) \;=\; \mathrm{Sh}\big(\iota_A L_{\mathrm{acc}}(A),\ \iota_B L_{\mathrm{acc}}(B)\big),$$

where ι_A and ι_B are the injections into $\Sigma_A \uplus_\tau \Sigma_B$ extended to words. If A and B emit over one common action alphabet and the injections are forgotten, the same equation holds with ι the identity.

Proof. For the inclusion from left to right, take an accepted run of $A \parallel B$ on the input word run^n. By Definition 4.5 each tick advances exactly one component and leaves the other where it is, so the run determines a function $c : \{1, \ldots, n\} \to \{A, B\}$ and, by restriction to $c^{-1}(A)$ and $c^{-1}(B)$, two runs of A and of B: the A-restriction is a legal run because the A-component of the state changes only at A-ticks and changes there by step_A, and likewise for B. The composite is accepted only when both components have halted, so both restrictions are accepted runs. Deleting the τ letters commutes with the restriction, so the visible word of the composite is an interleaving of the two visible words.

For the inclusion from right to left, let $u \in L_{\mathrm{acc}}(A)$ and $v \in L_{\mathrm{acc}}(B)$ be witnessed by accepted runs π_A of length n_A and π_B of length n_B, and let w be an interleaving of $\iota_A u$ and $\iota_B v$. Choose any $c : \{1, \ldots, n_A + n_B\} \to \{A, B\}$ with $|c^{-1}(A)| = n_A$ whose induced order on the non-idle ticks realises w; such a c exists because the idle ticks of π_A and π_B may be placed anywhere. The schedule c is available at every tick, since the union in Definition 4.5 ranges over all components at every state. The run of $A \parallel B$ that follows c and, at each A-tick, takes the transition π_A takes at its next position, is a run of the composite; it ends with both components halted, so it is accepted, and its visible word is w. □

Remark 4.11. The theorem is about $\parallel$ before any channel is linked in. Linking constrains the schedule: a component blocked on an empty channel emits τ until the writer has run, so the traces of a linked system form a subset of the shuffle of its components' traces, and the inclusion is usually strict. That is the price of communication, and it is the reason the deadlock theorem of Section 7 has to be proved rather than read off from Theorem 4.10.

4.4 The concurrent semiring

Part II proves that the synchronous product fails the interchange law: with a taking one tick and b taking two, $(a; c) \otimes (b; d)$ and $(a \otimes b); (c \otimes d)$ are related in neither direction by $\sqsubseteq$ (Part II, Counterexample 10.3). The failure is a timing artefact of lockstep, and the derived interleaving operator does not have it.

Theorem 4.12 (Concurrent semiring with exchange)**.** *Fix an action alphabet and let $\mathcal{K}$ be the schedulable agents with no ports in the interleaving fragment over that alphabet with $T = \mathcal{P}$, identified when their accepted-trace sets of Definition 4.8 agree, ordered by inclusion of those sets. Then*

(i) $(\mathcal{K}, \oplus, ;, 0, \mathrm{skip})$ *is an idempotent semiring;*

(ii) $(\mathcal{K}, \oplus, \parallel, 0, \mathrm{skip})$ *is a commutative idempotent semiring;*

(iii) the exchange inclusion holds: $(A \parallel B) ; (C \parallel D) \sqsubseteq (A ; C) \parallel (B ; D)$*;*

(iv) the small exchange laws $(A \parallel B) ; C \sqsubseteq A \parallel (B ; C)$ *and* $A ; (B \parallel C) \sqsubseteq (A ; B) \parallel C$ *hold, and hence* $A ; B \sqsubseteq A \parallel B$*.*

The inclusion in (iii) is strict for some A, B, C, D.

Remark 4.13. Part II proves the same semiring identities for a finer relation, equality of accepted-trace sets with the halting tick observable, and there the carrier has to exclude terms in which an argument of $\oplus$ has already halted at its initial state (Part II, Theorem 9.4): $\mathrm{skip} \oplus 0$ halts one tick later than skip, since the choice state of Part II, Definition 4.12 is running by construction. The reading used here deletes idle ticks, which is exactly the difference, so skip may appear as an argument of $\oplus$ and is the unit of both products. The price is that halting time is not observable in $\mathcal{K}$, which is what makes the shuffle law available in the first place.

Proof. The five operators are computed by L_{acc} as follows. $L_{\mathrm{acc}}(A \oplus B) = L_{\mathrm{acc}}(A) \cup L_{\mathrm{acc}}(B)$, because a nondeterministic choice contributes the accepted runs of both branches. $L_{\mathrm{acc}}(A\,;B) = L_{\mathrm{acc}}(A) \cdot L_{\mathrm{acc}}(B)$, because an accepted run of $A\,;B$ is an accepted run of A followed by an accepted run of B started at the unit value, the handoff consumes no tick, and deletion of idle letters is a monoid homomorphism on words. $L_{\mathrm{acc}}(0) = \emptyset$, because 0 never halts and therefore has no accepted run. $L_{\mathrm{acc}}(\mathrm{skip}) = \{\lambda\}$, because skip halts before its first tick. $L_{\mathrm{acc}}(A \parallel B) = \mathrm{Sh}(L_{\mathrm{acc}}(A), L_{\mathrm{acc}}(B))$ is Theorem 4.10 in its untagged reading. The first equation holds for arguments that have halted at their initial state as well: the choice state of Part II, Definition 4.12 runs for one tick before it enters the chosen branch, and that tick emits τ and is deleted.

Under these equations the four claims are statements about languages. (i) is the language model of an idempotent semiring, and it is the image of the accepted-trace theorem of Part II (Part II, Theorem 9.4) under the deletion of idle ticks, which is why no iteration axiom appears in either statement. (ii) holds because the shuffle is associative and commutative, has $\{\lambda\}$ as unit, distributes over union in each argument, and satisfies $\mathrm{Sh}(L, \emptyset) = \emptyset$. (iii) Let $w \in \mathrm{Sh}(L_A, L_B) \cdot \mathrm{Sh}(L_C, L_D)$, say $w = w_1 w_2$ with w_1 an interleaving of $u_A \in L_A$ and $u_B \in L_B$ and w_2 an interleaving of $u_C \in L_C$ and $u_D \in L_D$. Reading w from left to right and recording for each letter whether it came from the A or C side or from the B or D side exhibits w as an interleaving of $u_A u_C \in L_A \cdot L_C$ and $u_B u_D \in L_B \cdot L_D$, which is the required membership. (iv) is (iii) with D or with A replaced by skip; taking $B = \mathrm{skip}$ in the first small law and using the unit law of (ii) gives $A\,;C \sqsubseteq A \parallel C$.

For strictness, let a, b, c, d be four distinct actions and let A, B, C, D be the agents that emit a, b, c, d respectively at their first tick and halt at their second. Then the left side of (iii) denotes $\{ab, ba\} \cdot \{cd, dc\} = \{abcd, abdc, bacd, badc\}$ and the right side denotes $\mathrm{Sh}(ac, bd)$, which contains $acbd$. Since $acbd$ is not in the left side, the inclusion is strict. □

Remark 4.14. Iteration is not claimed. Concurrent Kleene algebra in the sense of Hoare, Möller, Struth and Wehrman [9] carries two star operations, and the star is an unguarded least fixed point, while the recursion operator of Part II is guarded and has unique solutions for that reason. The structure proved here is the concurrent semiring with exchange, which is the part of the axiomatisation the exchange law belongs to.

5 Actors

An actor is a behaviour with a mailbox and an address. The isolation that the actor literature states informally, that an actor's state is affected by nothing except the messages it has taken from its own mailbox, becomes a theorem once the mailbox is the only shared component.

Definition 5.1 (Actor)**.** Let $Addr$ be a finite set of addresses and M a message set. A *behaviour* is an ε-quiescent agent A of type $1 \Rightarrow 1$ whose port bundle is $M + \{\bot\}$, whose output carries a slot in $(Addr \times M) + \{\bot\}$ for at most one send per tick, and which is *message-driven*: $\mathrm{step}_A(s, (\mathsf{run}, \bot)) = \eta(\tau, s)$ for every s. The actor $\mathrm{Act}_a(A)$ at address a is the linking of A through the mailbox object $\mathrm{Ch}_\infty(M)$, with the reading port bound to A and the writing port left open, so that any component may enqueue to a.

Definition 5.2 (Actor system)**.** An *actor system* over $Addr$ is the object system

$$\mathrm{Sys}\big(\textstyle\bigotimes_{a \in Addr} \mathrm{Ch}_\infty(M);\ A_{a_1}, \ldots, A_{a_n}\big)$$

of Definition 4.6, in which the routing wire sends a message emitted with destination b to the mailbox of b and returns to each scheduled actor the response of its own mailbox. The family is fixed: $Addr$ is finite and every address carries a component from the first tick.

A fixed family cannot create actors, and no construction below adds one. An address whose behaviour ignores every input until an activation message arrives imitates a created actor within the bound $|Addr|$, but that device is not part of the simulation result and creation stays outside the fragment; see Section 10.

Proposition 5.3 (Simulation of the actor transition rules)**.** *Let a configuration be a pair $\langle \alpha \mid \mu \rangle$ with α a finite map from addresses to behaviour states and μ a finite multiset of messages in transit, and let the transition rules be those of Agha, Mason, Smith and Talcott [2] in the following three-rule form:* receive*, an actor at address a that is ready removes a message $\langle a \Leftarrow m \rangle$ from μ and continues with m;* send*, an actor emits $\langle b \Leftarrow m \rangle$ into μ;* become*, an actor replaces its state by the successor its behaviour prescribes. Creation is not covered, since the family of components is fixed. Define $\mathcal{S}$ to relate a configuration to a state of the actor system when α agrees with the components' states and μ agrees with the multiset union of the mailbox queues and the responses held for the addressed actors. Then for $T = \mathcal{P}$, whenever $\langle \alpha \mid \mu \rangle \to \langle \alpha' \mid \mu' \rangle$ by one of the three rules and $\mathcal{S}$ relates $\langle \alpha \mid \mu \rangle$ to a system state ς, there is a run of at most two ticks of the actor system from ς to some ς' with $\mathcal{S}$ relating $\langle \alpha' \mid \mu' \rangle$ to ς'.*

Proof. Each rule is matched by scheduling the acting address, which is always available because the union in Definition 4.5 ranges over every component. Receive takes two ticks. At the first, scheduling a issues a dequeue on a's mailbox, which by Definition 4.1 returns the head of the queue and removes it; the message moves from the queue to the response held for a, and $\mathcal{S}$ counts it in μ in both places, so the related configuration has not changed. At the second, a is scheduled again and its behaviour consumes the message, which is the transition to α'. The other two rules take one tick each, since the behaviour's own step performs the state change and carries at most one send in its output slot. Send: the scheduled actor emits a message with destination b, which the routing wire delivers to b's mailbox in the same tick, so the message enters the transit multiset. Become: the behaviour's own step changes the component state and is exactly what step_A does; no object is touched. The simulation is one way: ticks that schedule an actor with an empty mailbox correspond to no configuration transition, and they are idle by message-drivenness, so the correspondence is a simulation up to stuttering and not a bisimulation. □

Theorem 5.4 (Locality)**.** *Let* Sys *be an actor system, let a be an address with behaviour A, and consider any run of* Sys *of n ticks. Let $m_1, \ldots, m_r$ be the messages that A consumed from its mailbox during the run, in order. Then the state of the A-component after the run is $\mathrm{step}_A^{\dagger}(\mathrm{init}_A, m_1 \cdots m_r)$, the iterated step of A on that sequence alone: it does not depend on n, on the schedule, on the states of the other actors, or on the contents of any other mailbox. Consequently, if $A \sim A'$ then replacing A by A' at address a yields a bisimilar system.*

Proof. Induction on n. At a tick that does not schedule a, the A-component receives ε and, by ε-quiescence, does not move. At a tick that schedules a with response $\bot$, the component does not move either, by message-drivenness. At a tick that schedules a with a message m, the component moves by $\mathrm{step}_A(\cdot, (\mathsf{run}, m))$, and m is the next entry of the consumed sequence. The three cases exhaust the possibilities, because the only port of A is its own mailbox port and the routing wire feeds it only from the mailbox of a. For $T \neq \mathrm{Id}$ read the conclusion in T: the T-object of A-components after the run is the iterated Kleisli composite on the same sequence, since the binds for distinct components are independent and T is commutative.

For the substitution claim, Sys is a term built from $\otimes$, $\gg$ and Tr with A as one leaf, so Lemma 3.3 applies. Locality is what makes the substitution meaningful rather than merely formal: it says that the part of the system that could distinguish A from A' observes them only through the mailbox interface, which is the interface at which $A \sim A'$ was assumed. □

Table 2: The two coupling disciplines and the results proved for each. No result in this Part ranges over both fragments.

Fragment	Coupling	Results
Interleaving	Components share only channels, each with one writer and one reader.	Shuffle law for accepted traces, concurrent semiring with the exchange inclusion, determinacy of a single link between deterministic agents, binary session deadlock freedom.
Shared object	Components share an object that serves several clients, one operation per tick.	Linearizability with tick order, failure of linearizability under asynchronous replication, nondeterminism of tuple-space competition.

6 Shared state

When two components read and write one object, the shuffle law fails: the object's state records the order in which the operations arrived, so not every interleaving of the components' behaviours is available. What survives is a weaker and older correctness condition, linearizability. Table 2 records which results belong to which fragment.

6.1 Blackboards

Definition 6.1 (Blackboard)**.** Let K be a finite set of keys and V a set of values with a distinguished element. The blackboard BB is the shared object with state V^K, initial state constantly the distinguished value, and operations $\mathrm{read}(k)$, $\mathrm{write}(k, v)$ and $\mathrm{cas}(k, v, v')$ with the responses $m(k)$, ack, and the Boolean $[m(k) = v]$ respectively, the last operation writing v' exactly when that Boolean is true.

Definition 6.2 (Knowledge source and control agent)**.** A *knowledge source* is a client of an object system whose object is BB. A *control agent* is an agent C over $(Res_\perp, \{1, \ldots, n\})$ that replaces the scheduler: in the controlled system, the union over j in Definition 4.6 is restricted to the index C emits at that tick. Control is therefore not a new construct but a replacement of one component, and every result proved for all schedules holds for every control agent.

This is the arrangement of Hearsay-II [4] and of the control architecture of Hayes-Roth [6]: independent sources that inspect a shared partial solution, a scheduling component that decides which source runs next, and no direct communication between sources.

Definition 6.3 (History)**.** Fix a run of an object system. Its *history* is the sequence of events, ordered by tick, that contains $\mathrm{inv}_j(o)$ at each tick where client j is scheduled and emits the operation $o \neq \perp$, and $\mathrm{res}_j(r)$ at each tick where client j is scheduled and consumes a held response $r \neq \perp$; at a tick carrying both, the response event precedes the invocation event. An operation o *precedes* an operation o' in real time when the response event of o occurs at an earlier tick than the invocation event of o'. The history is *linearizable* when there is a total order on its completed operations that extends the real-time order and under which the responses are those the sequential object produces when the operations are applied in that order from the initial state.

Proposition 6.4 (Blackboard linearizability)**.** *Every history of an object system whose object serves one operation per tick, and in particular every history of a blackboard under $\|$, is linearizable for $T \in \{\mathrm{Id}, \mathcal{P}, \mathcal{D}\}$, with the order by application tick as a witness.*

Table 3: A run of the replicated blackboard whose history has no linearization. Each row is one tick; responses are consumed at the client's next scheduled tick, which is omitted from the table.

Tick	Scheduled client	Operation and response	State after the tick
1	w_1	write$(x, 1)$, ack	BB_1 holds 1; a forwarding message for BB_2 is queued.
2	w_2	write$(x, 2)$, ack	BB_2 holds 2; a forwarding message for BB_1 is queued.
3	r_1	read(x), 1	Unchanged.
4	r_2	read(x), 2	Unchanged.
5	forwarder	write$(x, 2)$ at BB_1	BB_1 holds 2.
6	forwarder	write$(x, 1)$ at BB_2	BB_2 holds 1.
7	r_1	read(x), 2	Unchanged.
8	r_2	read(x), 1	Unchanged.

Proof. By Definition 4.6 each tick applies at most one operation to the object, and the object's state evolves by applying those operations one at a time from its initial state. Order the completed operations by the tick at which they were applied; the order is total because at most one operation is applied per tick. It extends the real-time order: an operation is applied at the tick of its invocation event, and if o precedes o' in real time then the invocation of o' is at a strictly later tick than the response of o, which is at or after the invocation of o. The responses agree with the sequential object because the response recorded for a client is exactly the value the object returned at the application tick, and the object was in the state produced by applying the earlier operations in the same order. Per-client program order is a special case of the real-time order, since a client issues its next operation only after consuming the response to the previous one. □

Linearizability is a consequence of the one-operation-per-tick discipline of Definition 4.6 rather than an axiom of the algebra, and it fails as soon as the object is replicated.

Counterexample 6.5 (Replicated blackboard). Let the object be replaced by two blackboards BB_1 and BB_2 over one key x, each serving its own clients, with a forwarding component that carries every write applied at one replica to the other through a channel. Let w_1 and r_1 be clients of BB_1 and w_2 and r_2 clients of BB_2. The run of Table 3 produces a history that is not linearizable, for $T = \mathrm{Id}$.

Proof. Write w_1, w_2 for the two write operations and $r_1^a, r_1^b, r_2^a, r_2^b$ for the four reads, in the order of Table 3. Suppose a linearization exists. The response of r_1^a is 1, so the last write before it is w_1; the response of r_1^b is 2 and r_1^a precedes r_1^b in real time, so the last write before r_1^b is w_2, and therefore w_1 precedes w_2. The same argument applied to r_2^a and r_2^b, whose responses are 2 and 1, gives that w_2 precedes w_1. A total order cannot contain both, so no linearization exists. The history is produced by the run of Table 3, in which each replica applies one operation per tick and the forwarding channel delivers with the latency of Definition 4.2. □

The witness is the classical one for non-atomic registers [7]. It is recorded here because the algebra makes the two designs, one object and two replicas, terms of the same signature: the second is the first with an extra channel, and the extra channel is exactly what costs the correctness condition.

6.2 Tuple spaces

Definition 6.6 (Tuple space)**.** Fix a set of field values. A *tuple* is a finite sequence of values and a *template* is a finite sequence each of whose entries is a value or a wildcard; a template matches a

tuple when they have equal length and agree at every entry that is not a wildcard. The tuple space TS is the shared object with state a finite multiset of tuples, initial state the empty multiset, and, for $T = \mathcal{P}$, the operations $\mathrm{out}(t)$, which adds t and answers ack; $\mathrm{in}(p)$, which nondeterministically selects one tuple matching p, removes it and answers it, and answers $\perp$ when nothing matches; and $\mathrm{rd}(p)$, which answers a matching tuple without removing it, and $\perp$ when nothing matches.

Definition 6.7 (Outcome)**.** For a closed system with components $A_1, \ldots, A_n$, the *outcome* of a run is the tuple whose jth entry is the visible word of component j, that is the projection of the run's trace to the symbols emitted by A_j with idle symbols deleted, paired with A_j's halting outcome. Out is the set of outcomes over all runs, and over all fair schedules where fairness is assumed.

Proposition 6.8 (Simulation of the Linda primitives)**.** *Let a Linda configuration be a pair $\langle \theta \mid P_1, \ldots, P_n \rangle$ with θ a finite multiset of tuples and P_j processes, and let its transitions be those of Gelernter [5]:* $\mathrm{out}(t)$ *adds t to θ;* $\mathrm{in}(p)$ *removes some tuple of θ matching p and binds it, and is enabled only when such a tuple exists;* $\mathrm{rd}(p)$ *binds a matching tuple without removing it, under the same enabling condition. Relate a configuration to a state of the object system* $\mathrm{Sys}(\mathrm{TS}; A_1, \ldots, A_n)$ *when θ is the object's multiset and each P_j agrees with the client's state. Then every Linda transition is matched by one tick of the object system, with $T = \mathcal{P}$ and clients that reissue a blocked operation unchanged.*

Proof. Scheduling the acting client is always available. For out, the object appends the tuple in the tick the operation is issued and answers ack. For in and rd, the enabling condition of the Linda rule is that a matching tuple exists, which is exactly the case in which Definition 6.6 answers with a tuple rather than $\perp$; the nondeterministic selection among matching tuples is the union in the object's step. The blocked case of the Linda semantics has no transition, and the corresponding tick of the object system answers $\perp$ and leaves the client's state unchanged, so it is a stutter. The simulation is one way for that reason. Dynamic process creation, the eval primitive, is not covered; the dormant pool of Definition 5.2 is the bounded substitute. □

Proposition 6.9 (Competition is nondeterministic)**.** *Let* TS *hold the single tuple $\langle v \rangle$ and let C_1 and C_2 be clients that each issue* $\mathrm{in}(\langle ? \rangle)$ *once, emit* got *if the response is a tuple and* none *if it is $\perp$, and halt. Then $|\mathrm{Out}| \geq 2$ for $T = \mathcal{P}$.*

Proof. The schedule that runs C_1 first gives C_1 the tuple, since the object answers with the unique matching tuple and removes it, and leaves C_2 with $\perp$; the outcome has got in the first entry and none in the second. The schedule that runs C_2 first gives the opposite outcome. Both schedules are available at every tick by Definition 4.5, and both are fair, since every component is scheduled in each. The two outcomes differ in their first entry. □

Proposition 6.10 (A single link is determinate)**.** *Let $T = \mathrm{Id}$. Let P be a producer with no ports other than its own scheduling, which emits a fixed finite sequence of messages followed by an end marker and halts, and let C be a consumer whose only port is the reading end of* Ch_∞*, which performs blocking reads, meaning its state is unchanged when the response is $\perp$, and which halts on the end marker. In the system in which P and C are linked by that one channel, $|\mathrm{Out}| = 1$ over all fair schedules.*

Proof. P has no input port, so its sequence of emitted messages is a function of its own state alone and is the same in every run; call it $m_1 \cdots m_r \mathsf{end}$. The channel is first in, first out and never full, so the sequence of messages the consumer can dequeue is a prefix of that sequence, in order. By the locality argument of Theorem 5.4, applied with the channel port in place of the mailbox, the state

of C after any run is a function of the sequence of messages it dequeued; the responses equal to $\perp$ leave it unchanged by the blocking-read hypothesis. Under a fair schedule P is scheduled until it halts, so every message is enqueued, and C is scheduled infinitely often until it halts, so every message is dequeued in order and the end marker is reached. Hence in every fair run C consumes exactly $m_1 \cdots m_r\mathsf{end}$, both components halt, and both visible words are determined. The outcome does not record the interleaving, so it is the same for every fair schedule. □

The two propositions are the same system with one component changed. A channel with one reader delivers its messages in order to that reader; an object that several clients may take from does not, and no amount of scheduling discipline recovers determinacy without removing the competition. Proposition 6.10 is the two-node case of the determinacy of Kahn process networks [14]; Part IV proves the general acyclic case and cites this proposition as its base case.

7 Session types and deadlock freedom

A session type describes what a component is allowed to do on a channel. In this Part it is a predicate on the port traces of an agent rather than a new field of the agent tuple, so typing is a property of an agent already defined and adds nothing to the signature of Part I.

Definition 7.1 (Session type)**.** Fix a message set M and a finite label set Lab. Session types are the possibly infinite trees generated coinductively by

$$S \;::=\; \mathsf{end} \;\mid !m.S \;\mid ?m.S \;\mid\; \oplus\{l_i : S_i\}_{i\in J} \;\mid\; \&\{l_i : S_i\}_{i\in J},$$

with $m \in M$, the $l_i \in Lab$ distinct, and J finite and nonempty. The port action alphabet is $\mathcal{A} = \{!m, ?m : m \in M\} \cup \{!l, ?l : l \in Lab\}$, and $L(S) \subseteq \mathcal{A}^*$ is the prefix-closed language

$$\begin{aligned} L(\mathsf{end}) &= \{\lambda\}, & L(!m.S) &= \{\lambda\} \cup !m \cdot L(S), \\ L(?m.S) &= \{\lambda\} \cup ?m \cdot L(S), & L(\oplus\{l_i : S_i\}) &= \{\lambda\} \cup \textstyle\bigcup_i !l_i \cdot L(S_i), \\ L(\&\{l_i : S_i\}) &= \{\lambda\} \cup \textstyle\bigcup_i ?l_i \cdot L(S_i). \end{aligned}$$

Because the labels of a branch are distinct, a word $w \in L(S)$ determines a unique residual type S/w.

Definition 7.2 (Duality)**.** $\mathsf{end}^\perp = \mathsf{end}$, $(!m.S)^\perp = ?m.S^\perp$, $(?m.S)^\perp = !m.S^\perp$, $(\oplus\{l_i : S_i\})^\perp = \&\{l_i : {S_i}^\perp\}$ and $(\&\{l_i : S_i\})^\perp = \oplus\{l_i : {S_i}^\perp\}$, defined coinductively. Duality is an involution.

Definition 7.3 (Typing)**.** Let A be an agent with a single channel port. Its *port trace* in a run is the word over $\mathcal{A}$ recording, in order, the messages and labels it sent and received on that port. Then $A : S$ holds when both of the following do.

Fidelity. Every port trace of every run of A lies in $L(S)$.

Progress. For every reachable non-halted state s with port trace w so far, write $S' = S/w$. If S' is a send or a select, A emits a message or label allowed by S' after finitely many of the ticks at which it is scheduled, and its port trace does not change before it does. If S' is a receive or a branch, A issues a dequeue request at every tick at which it is scheduled, leaves its state unchanged while the response is $\perp$, and advances on a response allowed by S'. If $S' = \mathsf{end}$ then A halts, and A halts only when $S' = \mathsf{end}$.

Fidelity alone permits an agent that stops in the middle of a protocol and does nothing forever, which is why typing has the second clause. Progress is a liveness condition on a single agent; the theorem below turns two such conditions into a liveness property of the pair. The bound on the number of scheduled ticks before an action is finite rather than one, so an agent that consults another component before acting, such as the gated agent of Section 8, still satisfies it.

Definition 7.4 (Deadlock)**.** A reachable configuration of a closed system is a *deadlock* when some component has not halted and, for every component j and every step available to j from that configuration, the resulting configuration is the same one. A deadlocked system stutters forever without halting.

Lemma 7.5 (Session invariant)**.** *Let $A : S$ and $B : S^{\perp}$ be linked by two unbounded channels, one in each direction, which are their only coupling, and consider the path through S that a given run follows, written as a sequence $a_1 a_2 \cdots$ of directed actions, where a_k is a send by A if the kth constructor of S on that path is $!$ or $\oplus$, and a send by B otherwise. In every reachable configuration there are numbers $n_A, n_B \geq 0$ such that A has performed exactly $a_1 \cdots a_{n_A}$ on its port, B has performed the duals of $a_1 \cdots a_{n_B}$, the messages in transit from A to B are those a_k with $n_B < k \leq n_A$ that are sends by A, in order, the messages in transit from B to A are those a_k with $n_A < k \leq n_B$ that are sends by B, in order, and a_k is a send by A whenever $n_B < k \leq n_A$ and a send by B whenever $n_A < k \leq n_B$.*

Proof. Induction on the number of ticks. Initially $n_A = n_B = 0$ and both queues are empty. A tick either changes nothing, or advances one component by one action, since one component is scheduled per tick and a component advances only on a conforming action by Definition 7.3. If A sends, n_A increases by one and the message enters the queue, which preserves the description of the queues; fidelity guarantees the action is a_{n_A+1}, and it must be a send by A because A can only send what $S/a_1 \cdots a_{n_A}$ allows. If A receives, the message it takes is the head of the queue from B, which by the induction hypothesis is a_{n_A+1}; then n_A increases and the queue shortens. The two cases for B are symmetric, using $S^{\perp}/\cdot$ and the involutivity of duality. The last clause holds because a component never overtakes an action it must receive: if $n_B < k \leq n_A$ and a_k were a send by B, then A would have received a_k before B sent it. □

Theorem 7.6 (Binary deadlock freedom)**.** *Let $A : S$ and $B : S^{\perp}$ be linked by their only channels, one in each direction and both unbounded, under a fair schedule, with $T \in \{\mathrm{Id}, \mathcal{P}, \mathcal{D}\}$. Then no reachable configuration of the linked system is a deadlock, and the system halts exactly when the path followed through S reaches* end*: if that path is finite the system halts with both components halted, and if it is infinite the system never halts and performs infinitely many actions.*

Proof. Fix a reachable configuration and let n_A, n_B be as in Lemma 7.5. Suppose the configuration is not fully halted; we show some component can change it.

Assume without loss of generality $n_A \leq n_B$. If n_A is the length of the path, then by the last clause of the invariant $n_B = n_A$, both residuals are end, and by the halting clause of progress each component halts, so the configuration is fully halted, contrary to assumption. So the path has an action a_{n_A+1}, and we treat the three cases in which some component is committed to act.

If a_{n_A+1} is a send by A, progress makes A perform it after finitely many of A's scheduled ticks. Then not every step of A from this configuration can leave the configuration unchanged: if every one did, the system would return to the same configuration whenever A was scheduled, so A would never perform the send, contradicting progress. If a_{n_A+1} is a send by B and $n_A + 1 \leq n_B$, then B has already sent it and the invariant places it at the head of the queue from B to A; scheduling A then returns it from the channel, which changes both the queue and A's state. Otherwise a_{n_A+1} is

a send by B with $n_A + 1 > n_B$, so $n_B = n_A$ and a_{n_B+1} is the action B is committed to send; the first argument applied to B gives a step that changes the configuration. In every case a component can change the configuration, so it is not a deadlock.

For the second claim, the argument above exhibits, in every non-halted reachable configuration, a component that is committed to the next action and cannot be blocked from performing it. A fair schedule schedules that component infinitely often, so by the finite bound in the progress clause the action is performed and $\min(n_A, n_B)$ increases; iterating, it increases without bound unless the path is finite. If the path is finite of length n, then after finitely many advances $n_A = n_B = n$ and both components halt by the progress clause, so the system halts. If the path is infinite the system performs infinitely many actions and no component ever reaches end, so it never halts. □

Remark 7.7. The theorem is about two parties. With three or more, dual typing is replaced by projection from a global type and deadlock freedom needs the well-formedness conditions of Honda, Yoshida and Carbone [12]; nothing in the proof above extends to that case, since Lemma 7.5 uses a single linear order of actions that a multiparty protocol does not have. Section 10 records this as a limitation rather than a conjecture.

8 The approval gate

A human approval step is a channel to a co-agent that answers questions about proposed actions. Giving it the same status as any other component is what makes it possible to state, and prove, what approval costs and what it guarantees.

Definition 8.1 (Human co-agent)**.** A *human co-agent* H is an agent over the interface $(\Sigma_{\mathsf{ask}}, I_{ans})$ with $\Sigma_{\mathsf{ask}} = \Sigma_A + \{-\}$ and $I_{ans} = \{\mathsf{ok}, \mathsf{no}, \mathsf{pending}\}$: it receives proposed actions and answers approval, refusal, or nothing yet. H is an *oracle* when its answer is a function of the proposed action alone; it is *live* when from every reachable state it answers ok or no within finitely many of its scheduled ticks.

Definition 8.2 (Approval gate)**.** Let A be an agent over (I_A, Σ_A). The *gate skeleton* $G(A)$ is the agent over $(I_A \times I_{ans},\ \Sigma_A \times \Sigma_{\mathsf{ask}})$ with state $S_A \times (\{\mathsf{run}\} \cup \{\mathsf{wait}(\sigma) : \sigma \in \Sigma_A\})$ and the following two rules, in which the second component of the output is the question put to H.

Running. In state (s, run) with input $(i, \cdot)$, the gate steps A once. If A emits τ and moves to s', the gate emits $(\tau, -)$ and moves to (s', run). If A emits $\sigma \neq \tau$ and moves to s', the gate emits (τ, σ), asking about σ, and moves to $(s', \mathsf{wait}(\sigma))$.

Waiting. In state $(s, \mathsf{wait}(\sigma))$ the gate does not step A. On the answer $\mathsf{pending}$ it emits $(\tau, -)$ and stays. On ok it emits $(\sigma, -)$ and moves to (s, run). On no it emits $(\tau, -)$ and moves to (s, run).

$G(A)$ halts when A has halted and the skeleton is not waiting, with A's halting value, and fails when A fails. The *approval gate* $\mathrm{Gate}_H(A)$ is the linking of $G(A)$ and H through the question and answer ports, in the sense of Definition 4.2: those ports are consumed by Tr and hidden, so $\mathrm{Gate}_H(A)$ is an agent over (I_A, Σ_A) of the same sequential type as A, and every question spends at least one tick in the register before H sees it.

The skeleton freezes A while it waits rather than feeding it an idle input, so no assumption about A's reaction to idle input is needed. It also does not undo the step that produced a refused action: A has already moved to s', and only the emission is suppressed. Undoing the step would require the checkpointing of Part IV, and the difference is visible in the algebra rather than hidden in an implementation.

Theorem 8.3 (Gate refinement)**.** *Let* $T \in \{\mathrm{Id}, \mathcal{P}, \mathcal{D}\}$.

(i) In every run of $\mathrm{Gate}_H(A)$*, a tick at which the action port emits* $\sigma \neq \tau$ *is a tick at which the hidden answer port carried* ok*, and the question answered was* σ*, asked at an earlier tick. Under an oracle that refuses an action, that action never appears on the action port.*

(ii) Let H *be the oracle that answers* ok *at its first opportunity. For every trace of* A *on an input word* w *there is a trace of* $\mathrm{Gate}_H(A)$ *on a word obtained from* w *by inserting idle letters after the positions at which* A *emits a non-idle symbol, whose output word is that of* A *with* τ *inserted at the same positions; conversely every trace of* $\mathrm{Gate}_H(A)$ *whose inserted positions carry idle input arises this way. Hence, on input words that are idle while the gate waits,* $\mathrm{Gate}_H(A) \sqsubseteq A$ *and* $A \sqsubseteq \mathrm{Gate}_H(A)$.

Proof. (i) The action port emits a non-idle symbol only in the waiting rule on the answer ok, by inspection of Definition 8.2: the running rule emits τ on the action port in both of its cases. The symbol emitted is the one recorded in the state $\mathsf{wait}(\sigma)$, which is entered only at the tick at which σ was asked, and the skeleton leaves that state at the tick the answer arrives. If the oracle answers no for σ, the skeleton leaves $\mathsf{wait}(\sigma)$ by the third case, which emits τ, and σ is never held again unless A proposes it again.

(ii) Relate the state s of A to the states of $\mathrm{Gate}_H(A)$ whose skeleton component is (s, run). A tick of A from s emitting τ is matched by one tick of the gate, which emits τ and returns to a running skeleton state. A tick of A from s emitting $\sigma \neq \tau$ and reaching s' is matched by a block of ticks: the first consumes the same input i, emits τ on the action port and asks σ; the intermediate ticks carry the question to H and the answer back through the register, emitting τ; the last receives ok and emits σ, reaching a state whose skeleton component is (s', run). The intermediate ticks consume ε on the agent port and emit τ, so they are idle positions in the sense of the stutter closure of Part I. Halting is matched because the gate halts exactly when A has halted and no question is outstanding. Both inclusions of stutter-closed trace sets follow, which is mutual refinement in the sense of Part I, Definition 5.4, the relation being a preorder by Part I, Proposition 5.5. □

Corollary 8.4. *If* H *is live and the schedule is fair,* $\mathrm{Gate}_H(A)$ *has no reachable deadlock, and it halts whenever* A *would halt.*

Proof. Inside $\mathrm{Gate}_H(A)$ the skeleton and H are the two parties of a binary session over one channel in each direction, typed by $\mu X.\,!\mathsf{ask}.\,?\mathsf{ans}.X$ and its dual: the skeleton asks and then waits, and H receives and then answers. Liveness of H is the progress clause of Definition 7.3 for H, and the skeleton satisfies its own progress clause by Definition 8.2, since it asks at the first tick after A proposes and emits at the tick the answer arrives. Theorem 7.6 applies to that pair, so no configuration of the pair is a deadlock; A itself is frozen while the session is in progress and is therefore not part of it. □

9 Worked examples

The three examples below are small enough to check by hand and large enough to exercise the results. They are also the patterns that appear in current multi-agent language-model systems: conversational message passing in AutoGen [22], and a shared memory read and written by many agents in the generative-agent architecture of Park et al. [19]. Those systems are cited as instances of the patterns, not as sources of theorems.

9.1 A writer and a reviewer

Let W produce drafts and R review them. Their protocol, from W's side, is the session type

$$S \;=\; \mu X.\ !\mathsf{draft}.\ ?\mathsf{verdict}.\ \oplus\{\mathsf{revise} : X,\ \mathsf{accept} : \mathsf{end}\},$$

read coinductively as the infinite tree it unfolds to, and R is typed by $S^{\perp}$, which receives a draft, sends a verdict, and branches on the label. Take W to be the agent that emits a draft when scheduled, waits for a verdict, and selects accept once the verdict is positive or after n rounds, and take R to be the agent that emits a verdict computed from the draft it received. Both satisfy fidelity by construction, since neither has a transition that emits an action the type does not allow, and both satisfy progress, since each acts at its next scheduled tick and neither halts before end.

Theorem 7.6 then gives that the linked system has no reachable deadlock and halts exactly when W selects accept, which it does after at most n rounds. The result is worth contrasting with what happens without the type discipline: if R is replaced by an agent that waits for two drafts before answering, fidelity fails at the second ?draft, and the linked system reaches a configuration in which W waits for a verdict and R waits for a draft, with both queues empty. That configuration is a deadlock by Definition 7.4, and it is reachable, so the hypothesis of the theorem is not decorative.

Now put the writer behind an approval gate, $\mathrm{Gate}_H(W)$, with H a human co-agent that answers ok or no for each proposed draft. By clause (i) of Theorem 8.3 no draft reaches the channel without an ok for that draft. By clause (ii), if H approves everything, the gated system and the ungated one have the same stutter-closed traces, so the discipline costs idle ticks and changes nothing else. If H is live, Corollary 8.4 keeps the system deadlock-free. If H stops answering, the gated writer stops sending, and the linked system stalls; that is the intended behaviour of an approval gate and the theorem locates it exactly, in the liveness of the co-agent rather than in the protocol between the two agents.

9.2 A blackboard with two knowledge sources

Let BB hold two keys, *hyp* and *score*, and let K_1 and K_2 be knowledge sources: K_1 writes a hypothesis when the board is empty, and K_2 reads a hypothesis and writes a score. Let the control agent C alternate between them. A run produces the history

$$\mathrm{inv}_1(\mathrm{write}(\mathit{hyp}, h)),\ \mathrm{res}_1(\mathsf{ack}),\ \mathrm{inv}_2(\mathrm{read}(\mathit{hyp})),\ \mathrm{res}_2(h),\ \mathrm{inv}_2(\mathrm{write}(\mathit{score}, v)),\ \mathrm{res}_2(\mathsf{ack}),$$

whose linearization by application tick is the sequential history in the same order, so Proposition 6.4 applies and the responses are those of the sequential object. Replacing C by the nondeterministic scheduler adds runs in which K_2 reads before K_1 has written and receives the initial value; those histories are linearizable as well, with the read placed before the write. Linearizability is a statement about consistency, not about usefulness, and the control agent is what makes the schedule useful. Two knowledge sources that update the same key with cas show the difference: the failed compare-and-swap returns false, and the history remains linearizable, while the source that lost has to retry.

9.3 A work queue, twice

Let two workers take tasks from a tuple space that holds one task tuple. By Proposition 6.9 the system has at least two outcomes, one for each worker that wins the race, and no scheduling discipline internal to the tuple space removes the difference, since both schedules are fair. Now give each worker its own channel from a dispatcher that decides which worker receives which task. By

Proposition 6.10 each channel delivers its messages in order to its single reader, and if the dispatcher is deterministic the outcome is unique under every fair schedule.

The two designs differ in one component. Competition for a shared object buys load balancing and pays with nondeterminism; a dispatcher with private channels buys determinacy and pays with a component that must decide. The algebra makes the trade explicit, because the two systems are terms over the same signature that differ in which object sits between the workers.

10 Limitations

The series claim, quoted again, is the following.

> Part I fixes the object: discrete time, an effect monad T drawn from the list Id, $\mathcal{P}$, $\mathcal{D}$, 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.

What this Part proves of that claim is the wiring half: Tr, the channel constructions built from it, the interleaving operator $\|$ and the gate Gate_H are well-defined on agents and respect the relations named in each statement, under the added assumptions that the feedback register holds for exactly one tick, that traced ports are hidden, and that every parallel statement is made in one of the two fragments and never across both. The following are the specific things not proved.

The trace structure is weaker than a traced monoidal category and is not repaired here. Yanking fails by Counterexample 3.6, and dinaturality is not claimed in the general form; Lemma 3.4 covers only sliding along a stateless relabelling. Whether a useful class of stateful components can be slid across the register is open.

Session typing is binary. Theorem 7.6 uses a single linear order of protocol actions, which is exactly what a multiparty protocol lacks; the projection and well-formedness machinery of Honda, Yoshida and Carbone [12] is not developed here, and no claim is made about three or more parties. Delegation, subtyping and session-typed value passing are also absent.

The communication topology is fixed. Channels connect components named in advance, and no construction in this Part passes a channel name as a message, so the mobility of the π-calculus [18, 16] is outside the fragment. The family of actors is fixed, so Agha's creation rule is outside the fragment and Proposition 5.3 does not cover it; a bounded pool of dormant addresses imitates activation but does not make the family dynamic. Unbounded creation would need a signature with dynamic components, and the bigraph formalism of Milner [17] is the standard place where locality and connectivity are made dynamic together. This Part does not adopt it.

The interleaving operator is defined for $T = \mathcal{P}$ only, and it requires ε-quiescent components. It is not a bifunctor for sequential composition with early start: $A \| B$ halts when both components halt, and an operator that lets a halted component hand its value on while its neighbour is still running is not constructed here. The shuffle equality of Theorem 4.10 is stated for components that share nothing; once a channel is linked in, only the inclusion survives.

The shared-object results assume one operation per tick. Objects that serve several operations per tick, or that answer out of order, are not covered, and no weaker consistency condition, such as sequential or eventual consistency, is proved for the replicated case: the replicated blackboard appears here only as a counterexample to linearizability.

The gate does not roll back a refused action. Definition 8.2 suppresses the emission while the agent's state has already advanced, so refusal is not undo. An undoing gate needs the checkpointing of Part IV, and the composition of the two is not proved here. The refinement in clause (ii) of Theorem 8.3 is also stated only for input words that are idle while the gate waits, which is the condition under which the extra tick is a stutter.

The code layer that accompanies the series is a finite model. Every row of Table 4 reports a finite-model check or an exhaustive bounded check inside a fragment with finite state spaces, finite channel capacity and bounded tick budgets. No row establishes a statement about unbounded channels or about all runs, and every theorem above rests on its proof.

11 Conclusion

Wiring is one operator. A feedback operator with a one-tick register, applied to the synchronous product of Part II, generates channels, links, interleaving, actors, blackboards, tuple spaces, session-typed protocols and the human approval gate, and the register is visible in every one of them as latency. The operator satisfies naturality, vanishing, superposing and a delayed yanking law, and fails ordinary yanking by one tick, which places agents among the delayed-trace structures rather than the traced monoidal ones.

Two coupling disciplines have different algebra. Components coupled only by channels compose by shuffle and satisfy the exchange inclusion that the synchronous product violates; components sharing an object do not, and what they satisfy instead is linearizability with tick order, which replication destroys. Keeping the two apart is what makes both statements provable, and mixing them is the most common way that informal accounts of multi-agent systems go wrong.

The approval gate is a channel like any other. That is the practical content of the last theorem: a person in the loop is a co-agent with a session type, the guarantee is that no action precedes its approval, the cost under an approving oracle is exactly the idle ticks it adds, and the failure mode is the co-agent's liveness rather than anything in the protocol.

References

[1] Agha, G. (1986). *Actors: A Model of Concurrent Computation in Distributed Systems*. MIT Press, Cambridge, MA.

[2] Agha, G., Mason, I., Smith, S., Talcott, C. (1997). A foundation for actor computation. *Journal of Functional Programming* 7(1):1-72. DOI: 10.1017/S095679689700261X.

[3] Caires, L., Pfenning, F. (2010). Session types as intuitionistic linear propositions. *CONCUR 2010*, LNCS 6269, pp. 222-236. DOI: 10.1007/978-3-642-15375-4_16.

[4] Erman, L.D., Hayes-Roth, F., Lesser, V.R., Reddy, D.R. (1980). The Hearsay-II speech-understanding system: integrating knowledge to resolve uncertainty. *ACM Computing Surveys* 12(2):213-253.

[5] Gelernter, D. (1985). Generative communication in Linda. *ACM Transactions on Programming Languages and Systems* 7(1):80-112. DOI: 10.1145/2363.2433.

[6] Hayes-Roth, B. (1985). A blackboard architecture for control. *Artificial Intelligence* 26(3):251-321. DOI: 10.1016/0004-3702(85)90063-3.

[7] Herlihy, M., Wing, J. (1990). Linearizability: a correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems* 12(3):463-492.

[8] Hewitt, C., Bishop, P., Steiger, R. (1973). A universal modular ACTOR formalism for artificial intelligence. *IJCAI 1973*, pp. 235-245.

[9] Hoare, C.A.R., Möller, B., Struth, G., Wehrman, I. (2011). Concurrent Kleene algebra and its foundations. *Journal of Logic and Algebraic Programming* 80(6):266-296. DOI: 10.1016/j.jlap.2011.04.005.

[10] Honda, K. (1993). Types for dyadic interaction. *CONCUR 1993*, LNCS 715, pp. 509-523.

[11] Honda, K., Vasconcelos, V., Kubo, M. (1998). Language primitives and type discipline for structured communication-based programming. *ESOP 1998*, LNCS 1381, pp. 122-138. DOI: 10.1007/BFb0053567.

[12] Honda, K., Yoshida, N., Carbone, M. (2016). Multiparty asynchronous session types. *Journal of the ACM* 63(1), article 9.

[13] Joyal, A., Street, R., Verity, D. (1996). Traced monoidal categories. *Mathematical Proceedings of the Cambridge Philosophical Society* 119(3):447-468. DOI: 10.1017/S0305004100074338.

[14] Kahn, G. (1974). The semantics of a simple language for parallel programming. *IFIP Congress 1974*, pp. 471-475, North-Holland.

[15] Katis, P., Sabadini, N., Walters, R.F.C. (1997). Bicategories of processes. *Journal of Pure and Applied Algebra* 115(2):141-178. DOI: 10.1016/S0022-4049(96)00012-6.

[16] Milner, R. (1999). *Communicating and Mobile Systems: the π-calculus*. Cambridge University Press.

[17] Milner, R. (2009). *The Space and Motion of Communicating Agents*. Cambridge University Press.

[18] Milner, R., Parrow, J., Walker, D. (1992). A calculus of mobile processes, I and II. *Information and Computation* 100(1):1-40 and 41-77. DOI: 10.1016/0890-5401(92)90008-4 and 10.1016/0890-5401(92)90009-5.

[19] Park, J.S., O'Brien, J., Cai, C.J., Morris, M.R., Liang, P., Bernstein, M.S. (2023). Generative agents: interactive simulacra of human behavior. *UIST 2023*. DOI: 10.1145/3586183.3606763. arXiv:2304.03442.

[20] Sprunger, D., Katsumata, S. (2019). Differentiable causal computations via delayed trace. *LICS 2019*. arXiv:1903.01093.

[21] Wadler, P. (2014). Propositions as sessions. *Journal of Functional Programming* 24(2-3):384-418. DOI: 10.1017/S095679681400001X.

[22] Wu, Q., Bansal, G., Zhang, J., Wu, Y., Li, B., Zhu, E., Jiang, L., Zhang, X., Zhang, S., Liu, J., Awadallah, A.H., White, R.W., Burger, D., Wang, C. (2023). AutoGen: enabling next-gen LLM applications via multi-agent conversation. arXiv:2308.08155.

A Code evidence

The executable model that accompanies the series is finite: state spaces are explored to a stated cap, channels have finite capacity at least as large as the tick budget, recursion depth and tick budgets are bounded, and the distribution monad uses rational weights. Each row of Table 4 is therefore a finite-model check, a property tested on generated finite instances, or an exhaustive bounded check, all cases up to a stated bound. No row establishes a statement about unbounded channels or about all runs of an unbounded system. Where a row's check covers a smaller configuration, a single effect instance, or a shorter run than the statement it accompanies, the row says which part was checked and which was not, and such a row is a bounded check rather than a demonstration of the statement. The theorems rest on their proofs; the checks rule out the errors that a proof read only by its author tends to keep.

Table 4: Code evidence for the results of this Part. Test titles are quoted verbatim from the test suite.

Claim	File and identifier	Test (file, describe, it)	Runs	What the test shows
Theorem 3.5	`src/ops.ts` `tr`; `src/leaves.ts` `regLeaf`, `ignoreReg`, `mapFst`, `superpose`; `src/rules.ts` `stepTrace`	`test/part3-interaction-coordination.test.ts`› delayed-trace-axioms › naturality, vanishing and superposing hold up to bisimilarity on random agents	100 generated register agents, seed 20260903	For $T = \mathrm{Id}$: tracing a register the agent ignores returns the agent, relabelling the non-feedback output port commutes with Tr, and one side component can be pushed through Tr. Iterated vanishing, input naturality and the $\mathcal{P}$ and $\mathcal{D}$ instances are not checked.
Counterexample 3.6	`src/ops.ts` `tr`; `src/leaves.ts` `swapAgentLeaf`, `delayWireLeaf`, `echoLeaf`	`test/part3-interaction-coordination.test.ts`› delayed-trace-axioms › Tr(swap) is bisimilar to delay_1 and is not bisimilar to skip	One witness, two-letter alphabet, $T = \mathrm{Id}$	The traced symmetry is bisimilar to the one-tick register and is not bisimilar either to skip or to the copy wire, which is the separation the counterexample states for Id.
Proposition 3.8	`src/equiv.ts` `closedMachine`; `src/semantics.ts` `closedStep`; `src/ops.ts` `tr`, `tensor`	`test/part3-interaction-coordination.test.ts`› closed-loop-is-trace › Run(A,E) is bisimilar to Tr((A tensor E) ; swap) on random pairs	100 generated pairs, $T = \mathrm{Id}$	The concrete closed system and the trace form are bisimilar on generated pairs of a non-halting agent and a modular environment. Halting components and the $\mathcal{P}$ and $\mathcal{D}$ instances are not checked.

Claim	File and identifier	Test (file, describe, it)	Runs	What the test shows
Theorem 4.10	`src/ops.ts` `asyncPar`; `src/rules.ts` `stepAsyncPar`; `src/equiv.ts` `acceptedTraceSet`	`test/part3-interaction-coordination.test.ts`› shuffle-and-cka › the bounded trace set of asyncPar equals the shuffle of the component trace sets	100 generated component pairs, to the tick budget	For components that emit one or two letters and halt, the accepted-trace set of the interleaving equals the shuffle of the two words, computed to the tick budget. The unbounded statement is the theorem and rests on its proof.
Theorem 4.12	`src/ops.ts` `asyncPar`, `seq`; `src/equiv.ts` `acceptedTraceSet`, `subset`	`test/part3-interaction-coordination.test.ts`› shuffle-and-cka › the CKA exchange inclusion holds on random small agents and is strict on a witness	100 generated quadruples, plus one witness	The exchange inclusion holds on generated one and two tick components, and on the four-action witness the right side contains *acbd* while the left side does not, which is strictness. The semiring and small-exchange clauses are not checked.
Proposition 5.3	`src/ops.ts` `actor`, `seq`, `tensor`; `src/rules.ts` `stepActor`	`test/part3-interaction-coordination.test.ts`› actor-locality › Agha receive, send and become are simulated by a step of Act(A), and creation by a hand-assembled configuration	One fixed actor, three rules, $T = \mathrm{Id}$	On a fixed actor: a message is enqueued and dequeued in order, the behaviour's send reaches the interface, and the behaviour's state moves. Creation is exhibited by a hand-assembled two-actor configuration and is not a rule of the construction.

Claim	File and identifier	Test (file, describe, it)	Runs	What the test shows
Theorem 5.4	`src/ops.ts` `actor`; `src/rules.ts` `stepActor`; `src/semantics.ts` `initT`, `stepT`	`test/part3-interaction-coordination.test.ts` › actor-locality › actor state is a function of init and the dequeued message sequence across 1000 schedules	1000 distinct arrival schedules, $T = \mathrm{Id}$	After ten ticks the actor's state equals the state the behaviour reaches when driven by the derived dequeue sequence alone, on every one of the 1000 distinct schedules. The statement for all schedules and all T rests on the proof.
Proposition 6.4	`src/linearizability.ts` `isLinearizable`; `src/ops.ts` `blackboard`; `src/rules.ts` `stepBlackboard`	`test/part3-interaction-coordination.test.ts` › blackboard-linearizable › the linearizability checker accepts every blackboard history and rejects a replicated-blackboard history	All 120 programs of length at most four	Every history the blackboard produces from a program of at most four operations over two writes and a read is accepted by a brute-force linearizability checker, and so is the history with all intervals widened to overlap.
Counterexample 6.5	`src/linearizability.ts` `isLinearizable`	`test/part3-interaction-coordination.test.ts` › blackboard-linearizable › the linearizability checker accepts every blackboard history and rejects a replicated-blackboard history	One witness history, checked exhaustively	The same checker rejects the history of two overlapping writes followed by two reads that return different values, which is the history the counterexample derives from the replicated run. The run itself is given in the paper and not in the code.

Claim	File and identifier	Test (file, describe, it)	Runs	What the test shows
Proposition 6.9	`src/ops.ts` `tuplespace`; `src/sim.ts` `allFair`	`test/part3-interaction-coordination.test.ts`› tuplespace-nondeterministic › tuple-space competition yields at least two outcomes while a FIFO link yields exactly one	All resolutions of two matching requests	Two blocking requests against a space holding two matching tuples produce at least two outcomes over the enumerated resolutions, which is the nondeterminism of associative matching the proposition uses. The two-client configuration of the proposition is not built in the code.
Proposition 6.8	`src/ops.ts` `tuplespace`; `src/rules.ts` `tupleMatches`, `stepTuplespace`	`test/part3-interaction-coordination.test.ts`› tuplespace-nondeterministic › Linda out, in and rd are simulated by tuple-space steps with associative matching	Four fixed steps, $T = \mathrm{Id}$	On a fixed space: out adds the tuple, rd returns a match and leaves it, in returns a match and removes it, and a template with no match returns the empty response, which is the blocking case.
Proposition 6.10	`src/ops.ts` `link`, `channel`; `src/sim.ts` `allFair`	`test/part3-interaction-coordination.test.ts`› tuplespace-nondeterministic › tuple-space competition yields at least two outcomes while a FIFO link yields exactly one	All enumerated runs to eight ticks, capacity eight	A deterministic sender and receiver joined by one channel produce exactly one outcome over every enumerated run. The code's link is the lockstep wiring, so the quantifier over fair interleaved schedules is not exercised and the proposition rests on its proof.

Claim	File and identifier	Test (file, describe, it)	Runs	What the test shows
Theorem 7.6	`src/sessions.ts` `dual`, `fidelity`, `progress`, `typesAs`, `findDeadlock`	`test/part3-interaction-coordination.test.ts`› binary-deadlock-freedom › random dual-typed pairs never deadlock within the tick budget and an ill-typed pair does	100 generated send-only types, budget 20 ticks	For generated types built from sends, both typing clauses hold for the sender and the dual type for the receiver, and no deadlock is reachable within the budget; two receivers, which are ill typed, deadlock; fidelity fails for a party that keeps acting after `end` and for a nondeterministic party with one wrong branch. Branching types are not checked.
Theorem 8.3	`src/ops.ts` `gate`; `src/envs.ts` `approverEnv`; `src/rules.ts` `stepGate`; `src/equiv.ts` `refinesTraces`	`test/part3-interaction-coordination.test.ts`› gate-refinement › Gate with an always-ok oracle equals A modulo stutters and a rejected action never appears	One fixed agent with two actions, $T = \mathrm{Id}$	Under the approving oracle the gated run and the bare run refine each other after marker erasure and stutter collapse, and the first action is emitted at the tick the approval arrives; under the refusing oracle neither action appears on the action port.