deepseek

A Programming Paradigm for Spatiotemporal Composability

Yifan Shi[1,2], Wei Zhang[1], Tianyi Cui[2]

[1]**Peking University** [2]**DeepSeek-AI**

Abstract

Modern software—from plugin systems to self-evolving agent harnesses—increasingly requires *dynamic composition*, yet its formal foundations remain underdeveloped. We identify two orthogonal dimensions of the problem: *temporal composability*, the ability to completely revert a component's side effects upon removal, and *spatial composability*, the ability to declare and reactively manage inter-component dependencies. We address the two dimensions by lifting classical effect and coeffect concepts to runtime mechanisms. In particular, we formalize *revertible effects*, in which every context transformation carries an inverse that the runtime tracks. We formalize *reactive coeffects*, in which each change of the context notifies a component against its coeffect specification. We unify the effect context and the coeffect context into a single *context type*, which constitutes a programming paradigm. After that, we combine these mechanisms into the notion of a *component* and give a calculus of dynamic composition, whose metatheory carries spatiotemporal composability from a single component to a whole system of interleaved components. We implement these ideas in *Cordis*, a meta-framework of spatiotemporal composability that provides a core library with effect tracking and coeffect resolution, as well as a declarative component loader with configuration reconciliation and hot module replacement.

Contents

1. Introduction

Composition—assembling complex systems from simpler parts—is a foundational principle of software engineering [1]. Traditionally, composition is *static*: function calls, module imports, and class inheritance are resolved at compile time and remain fixed throughout execution. However, modern software increasingly demands *dynamic* composition, where components are loaded, unloaded, and reconfigured at runtime. Plugin architectures [2] and self-evolving agent harnesses both require systems that can safely add and remove functionality on the fly, yet current practice defers to coarse-grained mechanisms [3] that reconfigure only by restarting, discarding runtime state. Despite the growing practical importance of dynamic composition, its theoretical foundations remain underdeveloped, compared to the rich formal frameworks available for static composition.

1.1. Dimensions of Composability

To characterize the requirements of dynamic composition, we identify two orthogonal dimensions beyond the well-studied algebraic aspects of composition:

- **Temporal composability** addresses the *time* dimension: upon removal of a component, the modifications the component made to the shared environment must be completely and safely reversed. This requires tracking every resource allocation, event registration, and state mutation the component performs, and guaranteeing their orderly reclamation upon removal.
- **Spatial composability** addresses the *space* dimension: components must be able to declare, discover, and resolve their dependencies on one another in a structured and verifiable manner. This requires managing dependency topology and coordinating component lifecycles in response to dependency changes.

In the static setting, temporal composability reduces to lexical scoping (e.g., RAII [4], bracket patterns [5]), and spatial composability reduces to module import resolution [6]. In the dynamic setting, where components arrive and depart at runtime, both dimensions become significantly harder: temporal composability must handle long-lived, stateful effects whose scope is not lexically bounded; and spatial composability must handle dependencies that appear, disappear, or change identity during execution.

1.2. Motivating Examples

1.2.1. Plugin Systems

Plugin systems are a canonical instance of dynamic composition. We use Visual Studio Code (VSCode), one of the most widely-used extensible IDEs, as a representative example.

Temporal limitation. VSCode runs all extensions in a shared process called the extension host. Although extensions can be installed dynamically, this host provides no mechanism to unload an individual extension's code at runtime. Once an extension's `activate` function has executed, disabling or uninstalling it requires restarting the entire host, affecting all loaded extensions. Purely declarative extensions such as themes, keybindings, and snippets carry no

code and can be removed freely. Among the top 100 extensions by install count, however, 87 contain executable code[1] and will therefore require such a restart upon removal. Although VSCode provides a `deactivate` hook, it serves only as a graceful shutdown callback during the host process' termination, and thus does not enable live removal. Moreover, the hook separates effect disposal from effect creation (in `activate`), violating locality of concern and making complete cleanup difficult to verify.

Spatial limitation. VSCode does provide `extensionDependencies` for declaring dependencies between extensions, but it sees little use: among the top 100 extensions by install count, only 7 declare `extensionDependencies` on non-built-in extensions.[1] This scarcity reflects the shape of the extension API, which exposes fixed, surface-level extension points such as commands, views, and language features. Extensions contribute to the host through these points rather than depending on one another, so inter-extension dependencies rarely arise. Moreover, VSCode's mechanism for inter-extension interaction provides no structural contract: it exposes an extension's functionality to others through `vscode.extensions.getExtension(...).exports`, but the returned value is untyped (`any` by default), so the dependent cannot rely on a checked interface. In short, VSCode steers extensions toward a fixed set of host-provided extension points, and offers no safe, structured way for them to depend on one another.

These two limitations are not unique to VSCode; they recur across plugin systems generally [2, 7], differing only in degree.

1.2.2. Self-Evolving Agent Harnesses

Modern AI agents rely on runtime agent harnesses [8–10]. These systems may compose diverse tool suites [11] and execution environments, govern permissions and sandboxing, maintain session state and persistence, provide context management and memory systems [12], orchestrate subagents and multi-agent workflows [13], and expose interfaces to users and automation. A future harness may generate and deploy modifications to its own components while continuously serving requests. Model-synthesized reusable tools provide a narrower precursor to component-level self-modification [14]. Each such modification is itself an instance of dynamic composition.

Because these modifications occur continuously and with limited or no human oversight, dynamic composability becomes indispensable. Without temporal composability, each self-modification forces a full restart that discards all process-local accumulated state; at such frequency the cumulative unavailability becomes substantial, and in-flight tasks are disrupted repeatedly; even worse, a faulty self-modification can disable the very process needed to recover. Without spatial composability, each module must itself detect and adapt to changes in the modules it depends on as they appear, disappear, or change identity, and can do so only by ad hoc means; even worse, a naive code-replacement strategy may silently break dependents or introduce circular dependencies that surface only at reload time.

1.2.3. The Coarse-Grained Workaround

One reason dynamic composability has received limited formal attention is that operating systems and container orchestrators already provide a coarse-grained substitute. Operating systems yield temporal composability at the granularity of a process; container orchestrators

[1] Data retrieved from the Visual Studio Code Marketplace on June 9, 2026.

[3] yield spatial composability at the granularity of a service. In practice, most software tolerates the lack of fine-grained composability by deferring to these coarse-grained mechanisms: a misbehaving module is handled by restarting the process, and a service dependency is managed by the container orchestrator.

However, this workaround imposes substantial costs. Temporally, each restart discards all process-local accumulated state (e.g., caches, connections, partial computations), and rebuilding it takes seconds to minutes [15]; maintaining availability in the interim requires redundant replicas, incurring resource overhead to compensate for the inability to recover a single component. Spatially, container-level orchestration cannot express dependencies between components sharing an address space, and introduces network overhead for interactions that could be local function calls. Both mechanisms operate at the boundary of processes and containers, yet modern systems increasingly compose at a finer level. This granularity mismatch demands a compositional abstraction that manages effects and dependencies at the same level as the components themselves.

1.3. Contributions

The two dimensions of dynamic composability concern, respectively, how computations *modify* and how they *depend on* their environment. These two directions are what *effect systems* [16, 17] and *coeffect systems* [18, 19] formalize: effects provide the formal vocabulary for reasoning about environmental modifications, and coeffects for reasoning about environmental requirements. However, existing formulations restrict reasoning to compile-time analysis over lexically fixed scopes, and do not extend to dynamic scenarios where components arrive and depart at runtime. By lifting effects to a revertible runtime model and coeffects to a reactive dependency resolution mechanism, we obtain a unified formal foundation for dynamic composability, one that is language-agnostic and applicable to any software architecture requiring dynamic composition. We make the following contributions:

1. We formalize **revertible effects** (Section 3.1): every context transformation carries an explicit inverse that the runtime tracks, and both tracking and recovery preserve composition, so the context is recovered upon component removal. This establishes local temporal composability.
2. We formalize **reactive coeffects** (Section 3.2): a component declares the coeffects it requires as a specification, and each change of the context notifies the component against that specification as activating, deactivating, or neutral. This establishes local spatial composability.
3. We unify the effect context and the coeffect context into a single **context type** (Section 3.3), in which an observational equivalence on the coeffects supplies the effects with independence, constituting a programming paradigm for spatiotemporal composability.
4. We give a **calculus of dynamic composition** (Section 4), which combines the two mechanisms into the notion of a component and equips its lifecycle with an operational semantics. Its metatheory carries spatiotemporal composability from a single component to a whole system of interleaved components.
5. We implement these ideas in **Cordis** (Section 5), a meta-framework of spatiotemporal composability that provides a core library realizing the formal model with effect tracking and coeffect resolution, as well as a declarative component loader with configuration reconciliation and hot module replacement.

2. Preliminaries

This section provides a concise overview of effect and coeffect systems—the two theoretical pillars underlying our work. We assume familiarity with basic type theory and category theory; the goal here is to fix notation and introduce the key abstractions that Section 3 will operationalize as runtime mechanisms.

2.1. Effects

In the simply typed lambda calculus (STLC) [20, 21], a typing judgment $\Gamma \vdash t : T$ states that term t has type T under context Γ. An *effect system* refines the type to describe what side effects a computation may produce, yielding judgments of the form

$$\Gamma \vdash t : T_{\text{effect}} \tag{1}$$

Here, the result type is annotated with an element of an effect algebra that describes which side effects the computation may produce, enabling compositional reasoning about stateful computations. This approach originates with Lucassen and Gifford [22], who introduced a kinded type system distinguishing types, effects, and regions to discover scheduling constraints in parallel programs.

Monadic effects. Moggi [16] first modeled computational effects categorically via monads; Wadler [23] popularized the approach in Haskell. A monad (T, η, μ) on a category $\mathcal{C}$ encapsulates an effectful computation as a value of type $T(A)$, with $\eta : A \to T(A)$ lifting pure values and $\mu : T(T(A)) \to T(A)$ sequencing nested computations. Classic instances include the Maybe monad (for partiality), State monad (for mutable state), and IO monad (for external interaction).

Algebraic effects. Plotkin and Power [17, 24] showed that algebraic operations determine monads, establishing a framework in which effect interfaces are decoupled from their implementations. An effect signature Σ declares a set of operations (e.g., $\text{get} : () \to S$, $\text{put} : S \to ()$ for state); programs invoke operations freely without committing to a particular interpretation. Plotkin and Pretnar [25] subsequently introduced *effect handlers*, which interpret operations by providing continuation semantics:

$$\text{handle } e \text{ with } \{\ \text{op}(v, \kappa) \mapsto \ldots\ \} \tag{2}$$

The handler receives the operation argument v and the delimited continuation κ, which it may invoke zero, one, or multiple times, enabling exceptions, coroutines, and non-determinism within a uniform framework [26]. Languages such as Koka [27, 28], Eff [29], and OCaml 5 [30] have adopted algebraic effects with varying design trade-offs.

2.2. Coeffects

Dually to effects, a *coeffect system* [18, 31] enriches the context rather than the type, yielding judgments of the form

$$\Gamma_{\text{coeffect}} \vdash t : T \tag{3}$$

Here, the context is annotated with an element of a coeffect algebra describing what the computation requires from its environment, such as resources to access, permissions to hold,

or services to depend on. While effects model a program's impact on the world, coeffects model the world's constraints on the program.

Comonadic coeffects. The idea of using comonads to structure context-dependent computation was first developed by Uustalu and Vene [32], who proposed symmetric (semi)monoidal comonads as the dual of Moggi's monadic framework for effects, capturing notions such as dataflow and attribute evaluation. Petricek et al. [18] built on this foundation to propose coeffects as a unified static analysis of context-dependence. A comonad (D, ε, δ) captures context-dependent computation: $\varepsilon : D(A) \to A$ extracts the current value from a context, and $\delta : D(A) \to D(D(A))$ duplicates context for nested access. The Environment comonad $D(X) = E \times X$ models dependence on a fixed environment E; the Stream comonad $D(X) = \mathbb{N} \to X$ models dependence on temporal data.

Graded coeffects. For finer-grained tracking, *graded* coeffect systems use a pre-ordered semiring $\mathcal{S} = (S, \leq, +, \times, 0, 1)$ as the coeffect algebra [33], a discipline later unified with graded effects by Gaboardi et al. [19]. Elements of S annotate each variable binding to quantify its usage: 0 for unused, 1 for linear use, n for bounded use, ∞ for unrestricted use. The semiring operations compose coeffects sequentially $(\times)$ and in parallel $(+)$, enabling precise resource tracking, sensitivity analysis [34], and information-flow control [35, 36] within a unified algebraic framework [37].

2.3. Relationship to Dynamic Composability

Effect and coeffect systems organize reasoning about computation along two complementary directions: effects describe how a computation *modifies* its environment, whereas coeffects describe how it *depends on* its environment. These two directions correspond to the two dimensions of dynamic composability identified in Section 1:

- **Temporal composability** demands that a component's modifications to the shared environment be revertible upon unloading. The relevant effects are the stateful ones, which durably transform that environment; undoing such a transformation requires it to admit an inverse.
- **Spatial composability** demands that inter-component dependencies be declared and managed reactively. Such dependencies are the very thing coeffects capture, and managing them amounts to resolving each against what the environment supplies.

However, classical effect and coeffect systems are static instruments: effects are tracked within lexically fixed scopes and discharged by compile-time handlers; coeffect annotations are verified against contexts determined before execution. Dynamic composition, by contrast, requires these guarantees to hold for components that arrive and depart at runtime, against contexts that evolve continuously. No fixed lexical scope can delimit a plugin loaded after deployment; no compile-time context can anticipate dependencies that emerge from runtime configuration.

This motivates a shift in perspective: rather than extending static type systems with more annotations, we reify the conceptual structures of effects and coeffects so that a runtime can operate on them directly, establishing dynamically the guarantees these systems provide statically.

3. Revertible Effects and Reactive Coeffects

This section lifts the concepts of effects and coeffects introduced in Section 2 to runtime mechanisms, constructing a theory of dynamic composition. The central idea is to turn the *typing contexts* carrying effects and coeffects into *context types*, i.e., runtime-operable types that reify the context as a first-class entity. For the effect type, we model it as a context transformation paired with an inverse, achieving local temporal composability. For the coeffect context, we model it as a type carrying dependency information, achieving local spatial composability. An observational equivalence on the coeffects then supplies the effects with independence. The unified context that carries both effects and coeffects constitutes a programming paradigm in its own right.

3.1. Revertible Effects

Temporal composability is the ability to load and unload components at runtime such that, upon unloading, the shared environment is recovered to its pre-composition state. This requires that every modification a component makes to the environment be both trackable and recoverable. We therefore model an effect as a function of type $\Gamma \to \Gamma \times (\Gamma \to \Gamma)$: applied to the current context, it yields the modified context together with an explicit inverse. Supplying that inverse is what lets the effect be reverted, and returning it to the runtime is what makes the effect trackable. We call such effects *revertible*: by tracking and composing these inverses during execution, complete environment recovery becomes a structural guarantee.

3.1.1. Effect Context

Given any impure function $f_{\text{impure}} : X \to Y$, we transform it into a pure form $f : \Gamma \times X \to \Gamma \times Y$, where Γ is the context and all possible side effects can be represented as transformations on Γ. For any fixed input $x : X$, the induced map $\gamma \mapsto \text{pr}_1(f(\gamma, x))$ captures the side effect of f independently of the return value. Effects on Γ therefore live in the monoid of transformations $\Gamma \to \Gamma$ under composition $\circ$, where each monoid axiom has a direct reading as a property of effects:

- *Closure*: the sequential composition of two effects is again an effect;
- *Associativity*: a composite effect is independent of how it is bracketed;
- *Identity*: id_Γ, the identity function on Γ, acts as the unit of composition.

To model effects that can be undone, we pair each transformation f with another transformation g that undoes f, and call g a *left inverse* of f, abbreviated to *inverse* throughout the paper. Undoing is one-sided: what an inverse is held to is $g \circ f$ and never $f \circ g$. Pairs of transformations carry a multiplication of their own:

Definition 1. Define the *twisted composition* of pairs of context transformations by

$$(f_1, g_1) \circ (f_2, g_2) \coloneqq (f_1 \circ f_2,\, g_2 \circ g_1) \tag{4}$$

As for $\circ$ itself, the left operand acts after the right, and the inverses accumulate in the opposite order. It makes $(\Gamma \to \Gamma) \times (\Gamma \to \Gamma)$ a monoid with unit $(\text{id}_\Gamma, \text{id}_\Gamma)$, the product of the monoid of transformations with its opposite, which we call the twisted composition monoid $\mathfrak{T}_\Gamma$ over Γ.

To track effects within the context itself, we introduce the following definition:

Definition 2. Given a context Γ, define its *effect context* as:

$$\partial\Gamma \coloneqq \Gamma \times (\Gamma \to \Gamma) \tag{5}$$

It can be understood as a pair (γ, φ), where:

- $\gamma : \Gamma$ is the current context state;
- $\varphi : \Gamma \to \Gamma$ is the *accumulator*, the composite of the inverses of the effects performed so far, and the function that recovers the context to its initial state.

In particular, the initial effect context can be represented as $(\gamma_0, \mathrm{id}_\Gamma)$.

We also write $\partial^2\Gamma = \partial\Gamma \times (\partial\Gamma \to \partial\Gamma)$, and so on up the tower.

Given the presence of the accumulator φ, all effects performed on $\partial\Gamma$ can be tracked and recovered. We now give the concrete constructions for tracking and recovery.

Definition 3. Define the transformation track_Γ on pairs of context functions:

$$\begin{array}{lcccccc} \mathrm{track}_\Gamma & : & (\Gamma \to \Gamma) \times (\Gamma \to \Gamma) & \to & \partial\Gamma & \to & \partial\Gamma \\ \mathrm{track}_\Gamma & = & (f, g) & \mapsto & (\gamma, \varphi) & \mapsto & (f(\gamma), \varphi \circ g) \end{array} \tag{6}$$

This transformation converts a forward function f together with a candidate inverse g into a transformation of the effect context $\partial\Gamma$. Applying $\mathrm{track}_\Gamma(f, g)$ to a state (γ, φ) transforms γ by f and composes the inverse g onto φ, thereby tracking the effect of f in the context.

Theorem 4. For every $(f, g) \in (\Gamma \to \Gamma) \times (\Gamma \to \Gamma)$ the following diagram commutes, that is,

$$\mathrm{pr}_1 \circ \mathrm{track}_\Gamma(f, g) = f \circ \mathrm{pr}_1 \tag{7}$$

$$\begin{array}{ccc} \Gamma & \xrightarrow{\quad f \quad} & \Gamma \\ {\scriptstyle \mathrm{pr}_1}\uparrow & \Downarrow {\scriptstyle \mathrm{track}} & \uparrow{\scriptstyle \mathrm{pr}_1} \\ \partial\Gamma & \xrightarrow[\quad f' \quad]{} & \partial\Gamma \end{array}$$

Proof. For all $(\gamma, \varphi) \in \partial\Gamma$:

$$\begin{aligned} (\mathrm{pr}_1 \circ \mathrm{track}_\Gamma(f, g))(\gamma, \varphi) &= \mathrm{pr}_1(f(\gamma), \varphi \circ g) \\ &= f(\gamma) \\ &= (f \circ \mathrm{pr}_1)(\gamma, \varphi) \end{aligned}$$ □

Theorem 5. track_Γ is a monoid homomorphism from $\mathfrak{T}_\Gamma$ into $\partial\Gamma \to \partial\Gamma$. That is,

1. $\mathrm{track}_\Gamma(\mathrm{id}_\Gamma, \mathrm{id}_\Gamma) = \mathrm{id}_{\partial\Gamma}$;
2. for all $(f_1, g_1), (f_2, g_2) \in \mathfrak{T}_\Gamma$,
$$\mathrm{track}_\Gamma((f_1, g_1) \circ (f_2, g_2)) = \mathrm{track}_\Gamma(f_1, g_1) \circ \mathrm{track}_\Gamma(f_2, g_2) \tag{8}$$

Proof.

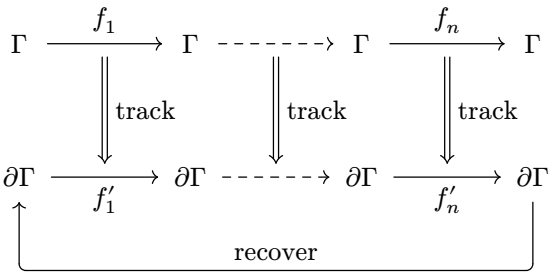
1. The unit is carried to the unit, since $\mathrm{track}_\Gamma(\mathrm{id}_\Gamma, \mathrm{id}_\Gamma)(\gamma, \varphi) = (\gamma, \varphi \circ \mathrm{id}_\Gamma) = (\gamma, \varphi)$.
2. For the multiplication, take any $(\gamma, \varphi) \in \partial\Gamma$:

$$\begin{aligned}(\mathrm{track}_\Gamma(f_1,g_1)\circ\mathrm{track}_\Gamma(f_2,g_2))(\gamma,\varphi) &= \mathrm{track}_\Gamma(f_1,g_1)(f_2(\gamma),\varphi\circ g_2)\\ &= (f_1(f_2(\gamma)),\varphi\circ g_2\circ g_1)\\ &= \mathrm{track}_\Gamma(f_1\circ f_2,\, g_2\circ g_1)(\gamma,\varphi)\end{aligned}$$ □

Definition 6. Define the transformation $\mathrm{recover}_\Gamma$ on $\partial\Gamma$:

$$\begin{array}{rcccc}\mathrm{recover}_\Gamma & : & \partial\Gamma & \to & \partial\Gamma\\ \mathrm{recover}_\Gamma & = & (\gamma,\varphi) & \mapsto & (\varphi(\gamma),\mathrm{id}_\Gamma)\end{array} \tag{9}$$

This transformation applies the recovery function φ to the current state γ and resets φ to the identity. The following diagram illustrates how recover recovers the context to its initial state after a sequence of effects $\mathrm{track}(f_1,g_1),\cdots,\mathrm{track}(f_n,g_n)$ has been applied to $\partial\Gamma$:



The diagram shows that the tracked effects followed by recover carry the initial effect context back to itself. What each tracking step preserves is the result of recovery itself, from whatever state it is taken:

Theorem 7. For every $(\gamma,\varphi)\in\partial\Gamma$ and every pair (f,g) with $g(f(\gamma))=\gamma$,

$$\mathrm{recover}_\Gamma(\mathrm{track}_\Gamma(f,g)(\gamma,\varphi)) = \mathrm{recover}_\Gamma(\gamma,\varphi) \tag{10}$$

Proof.

$$\begin{aligned}\mathrm{recover}_\Gamma(\mathrm{track}_\Gamma(f,g)(\gamma,\varphi)) &= \mathrm{recover}_\Gamma(f(\gamma),\varphi\circ g)\\ &= (\varphi(g(f(\gamma))),\mathrm{id}_\Gamma)\\ &= (\varphi(\gamma),\mathrm{id}_\Gamma) = \mathrm{recover}_\Gamma(\gamma,\varphi)\end{aligned}$$ □

A sequence of pairs needs no separate argument. Let $(f_1,g_1),\cdots,(f_n,g_n)$ be applied in order from (γ,φ), and write $\delta_0=\gamma$ and $\delta_i=f_i(\delta_{i-1})$. By Theorem 5 the composite $\mathrm{track}_\Gamma(f_n,g_n)\circ\cdots\circ\mathrm{track}_\Gamma(f_1,g_1)$ is track_Γ of the twisted composite $(f_n\circ\cdots\circ f_1,\, g_1\circ\cdots\circ g_n)$, and if $g_i(\delta_i)=\delta_{i-1}$ for every i then $(g_1\circ\cdots\circ g_n)(\delta_n)=\delta_0=\gamma$. That pair therefore meets the hypothesis of Theorem 7 at γ, and one application of the theorem gives

$$\mathrm{recover}_\Gamma((\mathrm{track}_\Gamma(f_n,g_n)\circ\cdots\circ\mathrm{track}_\Gamma(f_1,g_1))(\gamma,\varphi)) = \mathrm{recover}_\Gamma(\gamma,\varphi) \tag{11}$$

Taking $(\gamma,\varphi)=(\gamma_0,\mathrm{id}_\Gamma)$, recovery carries every state reached this way back to $(\gamma_0,\mathrm{id}_\Gamma)$. A pair with $g\circ f=\mathrm{id}_\Gamma$ meets the hypothesis at every state.

Recovery reads a state through the quantity $\varphi(\gamma)$, and we refer to $\varphi(\gamma)=\gamma_0$ as the *soundness invariant* of a state in $\partial\Gamma$.

3.1.2. Revertible Effect Functions

The track/recover model of the previous section takes inverses as given a priori: $\mathrm{track}_\Gamma(f, g)$ fixes g before any context state is seen, so one g has to serve every state the effect is applied at. In practice, however, the inverse of each effect is not known a priori: it must be supplied by the caller at the point of effect application. Moreover, recover is all-or-nothing: it cannot selectively undo one effect while retaining others. To address both issues, we enhance the model at both the input and output sides:

1. On the input side, we not only transform Γ but also return an inverse function alongside it, so that the inverse is supplied where the effect is applied: $\Gamma \to \Gamma \times (\Gamma \to \Gamma)$, i.e., $\Gamma \to \partial\Gamma$;
2. On the output side, we not only transform $\partial\Gamma$ but also return an inverse function alongside it, so that one effect can be undone while the others are retained: $\partial\Gamma \to \partial\Gamma \times (\partial\Gamma \to \partial\Gamma)$, i.e., $\partial\Gamma \to \partial^2\Gamma$.

This enhancement preserves structural consistency between input and output, so we can still define corresponding theory that maintains the mathematical properties of track. The resulting types are the effect functions $\mathfrak{E}_\Gamma$ and their witnessed refinement $\mathfrak{E}^*_\Gamma$:

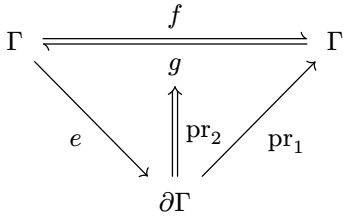
Definition 8. Define the effect function $\mathfrak{E}_\Gamma$ and *witnessed* effect function $\mathfrak{E}^*_\Gamma$ as:

$$\begin{aligned} \mathfrak{E}_\Gamma &:= \Gamma \to \Gamma \times (\Gamma \to \Gamma) \\ \mathfrak{E}^*_\Gamma &:= (e : \Gamma \to \Gamma \times (\Gamma \to \Gamma)) \\ &\quad \times ((\gamma : \Gamma) \to ((\delta : \Gamma) \times (g : \Gamma \to \Gamma) \times ((\delta, g) = e(\gamma) \to g(\delta) = \gamma))) \end{aligned} \tag{12}$$

where $e(\gamma)$ yields a pair (δ, g) representing:

- $\delta : \Gamma$ is the new context;
- $g : \Gamma \to \Gamma$ is the inverse function of the current effect.

An element of $\mathfrak{E}^*_\Gamma$ chooses its inverse per state, and the constraint $g(\delta) = \gamma$ holds that choice to reverting the effect where it was applied, leaving g unconstrained everywhere else. A single g with $g \circ f = \mathrm{id}_\Gamma$ meets the constraint at every state at once, and induces an element of $\mathfrak{E}^*_\Gamma$ by $(f, g) \mapsto \gamma \mapsto (f(\gamma), g)$, which Theorem 11 shows to be a homomorphism. The constraint can be visualized as the following commutative diagram, ensuring that the inverse e returns indeed reverses the transformation at the state where e was applied:



Since effect functions $\mathfrak{E}_\Gamma$ are no longer endomorphisms on the context, they cannot be directly composed. We therefore define a new operation for effect composition:

Definition 9. Given functions $f, g \in \mathfrak{E}_\Gamma$, define their *effect composition* $f \diamond g$ as:

$$
\begin{array}{rcl}
f \diamond g & : & \Gamma \to \qquad\qquad \partial\Gamma \\
f \diamond g & = & \gamma \mapsto \begin{array}{l} \textbf{let } (\delta, s) = g(\gamma) \textbf{ in} \\ \textbf{let } (\varepsilon, t) = f(\delta) \textbf{ in} \\ (\varepsilon, s \circ t) \end{array}
\end{array}
\tag{13}
$$

Theorem 10. Effect composition carries the monoid structure of $\mathfrak{T}_\Gamma$ over to $\mathfrak{E}_\Gamma$. That is,

1. $(\mathfrak{E}_\Gamma, \diamond)$ is a monoid with unit $\eta_\Gamma := \gamma \mapsto (\gamma, \mathrm{id}_\Gamma)$;
2. the assignment $(f, g) \mapsto \gamma \mapsto (f(\gamma), g)$ is a monoid homomorphism from $\mathfrak{T}_\Gamma$ into $\mathfrak{E}_\Gamma$.

Proof.

1. Associativity and the unit laws follow componentwise from those of $\circ$.
2. Write $e_i = \gamma \mapsto (f_i(\gamma), g_i)$; then $(e_1 \diamond e_2)(\gamma) = (f_1(f_2(\gamma)),\, g_2 \circ g_1)$, which is the image of $(f_1, g_1) \circ (f_2, g_2)$, and $(\mathrm{id}_\Gamma, \mathrm{id}_\Gamma)$ maps to η_Γ. □

Theorem 11. Witnessing survives effect composition, and a uniform inverse witnesses at every state. That is,

1. $\mathfrak{E}^*_\Gamma$ is a submonoid of $\mathfrak{E}_\Gamma$;
2. the homomorphism of Theorem 10 carries every pair with $g \circ f = \mathrm{id}_\Gamma$ into $\mathfrak{E}^*_\Gamma$.

Proof.

1. The unit lies in $\mathfrak{E}^*_\Gamma$ since $\mathrm{id}_\Gamma(\gamma) = \gamma$. For closure, take $f, g \in \mathfrak{E}^*_\Gamma$ and any $\gamma \in \Gamma$, and let $(\delta, s) = g(\gamma), (\varepsilon, t) = f(\delta)$, so that $(f \diamond g)(\gamma) = (\varepsilon, s \circ t)$. Then $s(\delta) = \gamma$ and $t(\varepsilon) = \delta$, therefore $(s \circ t)(\varepsilon) = s(\delta) = \gamma$.
2. $g \circ f = \mathrm{id}_\Gamma$ gives $g(f(\gamma)) = \gamma$ at every γ, so the image of such a pair is witnessed at every state. □

Just as track lifts a pair of transformations on Γ to $\partial\Gamma$, we define effect to lift $\mathfrak{E}_\Gamma$ to $\mathfrak{E}_{\partial\Gamma}$:

Definition 12. Define the effect function transformation effect_Γ as:

$$
\begin{array}{rcl}
\mathrm{effect}_\Gamma & : & \mathfrak{E}_\Gamma \to \partial\Gamma \to \qquad\qquad \partial^2\Gamma \\
\mathrm{effect}_\Gamma & = & e \mapsto (\gamma, \varphi) \mapsto \begin{array}{l} \textbf{let } (\delta, g) = e(\gamma) \textbf{ in} \\ ((\delta, \varphi \circ g),\, \mathrm{track}_\Gamma(g, \mathrm{pr}_1 \circ e)) \end{array}
\end{array}
\tag{14}
$$

Since $\mathrm{effect}_\Gamma(e)$ is itself $\mathfrak{E}_{\partial\Gamma}$, what it returns is an inverse in the sense of Definition 8 read one level up. That inverse is itself a track of the pair obtained by swapping the two directions of the effect. The ordinary tracking rule applies once more: undoing the effect is an effect in its own right, transforming the state by g, and the way to undo *that* is to perform the effect again, which is what $\mathrm{pr}_1 \circ e$ does. The inverse therefore composes onto the accumulator it is handed, exactly as track prescribes.

We can now prove properties for effect analogous to those of track.

Theorem 13. effect preserves the $\diamond$ operation. That is, $\forall f, g \in \mathfrak{E}_\Gamma$:

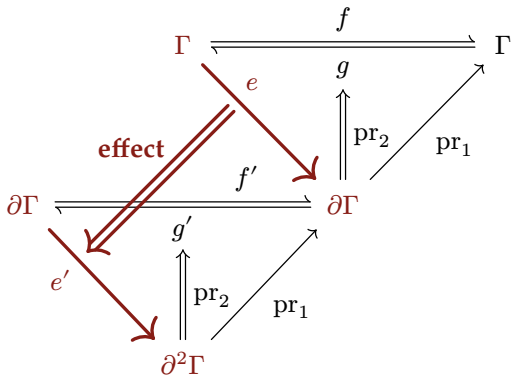
$$
\mathrm{effect}_\Gamma(f) \diamond \mathrm{effect}_\Gamma(g) = \mathrm{effect}_\Gamma(f \diamond g) \tag{15}
$$

Proof. Take any $(\gamma, \varphi) \in \partial\Gamma$, and let $(\delta, s) = g(\gamma)$ and $(\varepsilon, t) = f(\delta)$, so that $(f \diamond g)(\gamma) = (\varepsilon, s \circ t)$ and $\mathrm{pr}_1 \circ (f \diamond g) = (\mathrm{pr}_1 \circ f) \circ (\mathrm{pr}_1 \circ g)$. Then

$$\begin{aligned}(\mathrm{effect}_\Gamma(f) \diamond \mathrm{effect}_\Gamma(g))(\gamma,\varphi) &= ((\varepsilon, \varphi \circ s \circ t),\, \mathrm{track}_\Gamma(s, \mathrm{pr}_1 \circ g) \circ \mathrm{track}_\Gamma(t, \mathrm{pr}_1 \circ f)) \\ &= ((\varepsilon, \varphi \circ s \circ t),\, \mathrm{track}_\Gamma(s \circ t,\, (\mathrm{pr}_1 \circ f) \circ (\mathrm{pr}_1 \circ g))) \\ &= \mathrm{effect}_\Gamma(f \diamond g)(\gamma,\varphi)\end{aligned}$$

where the first step unfolds Definition 12 at (γ,φ) and at $(\delta, \varphi \circ s)$, the second is Theorem 5, and the third folds Definition 12. □

How the two levels relate is what the following diagram shows. Its upper triangle is the witness condition of e, according to Definition 8, and its lower triangle is the question of whether e' is witnessed the way e is.



Between the levels, the projection pr_1 relates each lifted map to the map it lifts, as it does for track_Γ in Theorem 4.

Theorem 14. Let $e \in \mathfrak{E}_\Gamma$, write $f := \mathrm{pr}_1 \circ e$, and let $e' := \mathrm{effect}_\Gamma(e)$ with forward map $f' := \mathrm{pr}_1 \circ e'$. Then

1. $\mathrm{pr}_1 \circ f' = f \circ \mathrm{pr}_1$;
2. for each $(\gamma,\varphi) \in \partial\Gamma$, the lifted inverse $g' := \mathrm{pr}_2(e'(\gamma,\varphi))$ and the inverse $g := \mathrm{pr}_2(e(\gamma))$ witnessed there satisfy $\mathrm{pr}_1 \circ g' = g \circ \mathrm{pr}_1$.

Proof.

1. By Definition 12, $f'(\gamma,\varphi) = (f(\gamma), \varphi \circ g)$, whose state is $f(\gamma) = (f \circ \mathrm{pr}_1)(\gamma,\varphi)$.
2. This is Theorem 4 applied to $g' = \mathrm{track}_\Gamma(g, f)$. □

Whether the lower triangle closes is settled by computing what the lifted inverse returns:

Theorem 15. Let $e \in \mathfrak{E}^*_\Gamma$ and write $f := \mathrm{pr}_1 \circ e$. Fix $(\gamma,\varphi) \in \partial\Gamma$, let $(\delta, g) = e(\gamma)$, and write (Δ, g') for the value of $\mathrm{effect}_\Gamma(e)$ at (γ,φ). Then

$$g'(\Delta) = (\gamma,\, \varphi \circ g \circ f) \tag{16}$$

The state is recovered exactly. The accumulator is restored as well, equivalently $\mathrm{effect}_\Gamma(e) \in \mathfrak{E}^*_{\partial\Gamma}$, if and only if $g \circ f = \mathrm{id}_\Gamma$; and in every case $(\varphi \circ g \circ f)(\gamma) = \varphi(\gamma)$, so the soundness invariant is preserved.

Proof. By Definition 12, $\Delta = (\delta, \varphi \circ g)$ and $g' = \mathrm{track}_\Gamma(g, f)$, so

$$g'(\Delta) = (g(\delta),\, \varphi \circ g \circ f) = (\gamma,\, \varphi \circ g \circ f)$$

using $g(\delta) = \gamma$. Membership in $\mathfrak{E}^*_{\partial\Gamma}$ requires this to equal (γ, φ) at every input; taking $\varphi = \mathrm{id}_\Gamma$ turns the equality of accumulators into $g \circ f = \mathrm{id}_\Gamma$, and that condition conversely gives the equality of accumulators for every φ. Finally $(\varphi \circ g \circ f)(\gamma) = \varphi(g(\delta)) = \varphi(\gamma)$. □

The lower triangle therefore closes only when the inverse witnessed at γ reverts f at every state, so effect_Γ does not carry $\mathfrak{E}^*_\Gamma$ into $\mathfrak{E}^*_{\partial\Gamma}$. What holds in every case is agreement at γ: $\mathrm{recover}_\Gamma(g'(\Delta)) = \mathrm{recover}_\Gamma(\gamma, \varphi)$, which is the whole of what Theorem 7 assumes of an accumulator, so reverting leaves the recovery target untouched.

Reverting effects in the reverse of the order in which they were applied requires nothing further, because each inverse then meets the state its own application produced:

Theorem 16. Let $e_1, \cdots, e_n \in \mathfrak{E}^*_\Gamma$ be applied in order from $(\gamma_0, \mathrm{id}_\Gamma)$ and reverted in the reverse order. Then

1. each revert recovers the context state its application ran against;
2. every intermediate state satisfies the soundness invariant.

Proof. Each step is an application or a revert. An application carries (γ, φ) to $(\delta, \varphi \circ g)$ with $g(\delta) = \gamma$, so it preserves $\varphi(\gamma)$ by Theorem 7, whose hypothesis is exactly the witness of $\mathfrak{E}^*_\Gamma$. Reverting in the reverse order hands each inverse the state its own application produced, so by Theorem 15 that revert recovers the preceding state exactly and preserves $\varphi(\gamma)$ as well; neither conclusion depends on the accumulator the inverse receives. □

3.1.3. Independence of Effects

Reverting an effect at the state its own application produced is what Theorem 16 covers; reverting one at any other state is what this subsection covers. Two situations call for the latter. An inverse may be run while later effects are still in place, which is what withdrawing one component from a running system amounts to; and one sequence may interleave the effects of several components, each keeping the inverses of its own, so that the inverses of one component are separated by the applications of another. In both an inverse meets a state that foreign effects have moved, and whether it still reverts what it was built to revert is a question of commutation: what has to commute is every transformation one effect can perform with every transformation the other can perform, forward map and yielded inverse alike. A single accumulator settles neither situation, φ being a composite that runs every inverse it holds in one order and all at once.

Definition 17. For an effect function $e \in \mathfrak{E}_\Gamma$, the *transformation monoid* $\mathfrak{M}(e)$ is the submonoid of $\Gamma \to \Gamma$ generated by the forward map of e together with every inverse e yields, and the *generators* of $\mathfrak{M}(e)$ are the elements of that generating set:

$$\mathfrak{M}(e) \coloneqq \langle \{\mathrm{pr}_1 \circ e\} \cup \{\mathrm{pr}_2(e(\gamma)) \mid \gamma \in \Gamma\} \rangle \tag{17}$$

An effect induced by a pair $(f, g) \in \mathfrak{T}_\Gamma$ has f and g for its generators, the inverse it yields being g at every state.

Lemma 18. Commutation is settled on the generators, and $\diamond$ enlarges no transformation monoid. That is,

1. if every generator of $\mathfrak{M}(e_1)$ commutes with every generator of $\mathfrak{M}(e_2)$, then every element of $\mathfrak{M}(e_1)$ commutes with every element of $\mathfrak{M}(e_2)$;
2. $\mathfrak{M}(e_1 \diamond e_2) \subseteq \langle \mathfrak{M}(e_1) \cup \mathfrak{M}(e_2) \rangle$.

Proof.

1. The maps commuting with every generator of $\mathfrak{M}(e_2)$ form a submonoid of $\Gamma \to \Gamma$, since id_Γ lies in it and $f \circ f'$ does where f and f' do. That submonoid contains the generators of $\mathfrak{M}(e_1)$ by hypothesis and hence contains $\mathfrak{M}(e_1)$. Fixing $f \in \mathfrak{M}(e_1)$, the maps commuting with f likewise form a submonoid containing the generators of $\mathfrak{M}(e_2)$ and hence $\mathfrak{M}(e_2)$.
2. By Definition 9 the forward map of $e_1 \diamond e_2$ is $(\mathrm{pr}_1 \circ e_1) \circ (\mathrm{pr}_1 \circ e_2)$ and the inverse it yields at any state is $s \circ t$ for an s yielded by e_2 and a t yielded by e_1. Every generator of $\mathfrak{M}(e_1 \diamond e_2)$ is therefore a composite of generators of the two. □

Definition 19. Effect functions $e_1, e_2 \in \mathfrak{E}_\Gamma$ are *independent* when

1. every transformation of one commutes with every transformation of the other,
$$\forall f \in \mathfrak{M}(e_1),\, g \in \mathfrak{M}(e_2). \quad f \circ g = g \circ f \tag{18}$$
2. neither one's transformations disturb the inverse the other yields,
$$\forall g \in \mathfrak{M}(e_2),\, \gamma \in \Gamma. \quad \mathrm{pr}_2(e_1(g(\gamma))) = \mathrm{pr}_2(e_1(\gamma)) \tag{19}$$
and the same with e_1 and e_2 exchanged.

A family $(e_l)_{l \in L}$ is *pairwise independent* when e_l and $e_{l'}$ are independent for every $l \neq l'$. A family may repeat an effect function, and holding one independent of itself is holding $\mathfrak{M}(e)$ commutative.

For effects induced by pairs (f_1, g_1) and (f_2, g_2), clause (1) is by Lemma 18(1) the commutation of the four pairs f_1, f_2; g_1, g_2; f_1, g_2; and g_1, f_2, and clause (2) holds outright, an induced effect yielding one inverse at every state. Commutation under $\diamond$ is a different property. What $e_1 \diamond e_2 = e_2 \diamond e_1$ equates is the composite forward map of the two orders with each other and the composite inverse of the two orders with each other, each inverse entering the composite at the state its own application produced; independence instead relates each transformation of one effect to each transformation of the other, a forward map paired with a foreign inverse included.

Under independence an inverse may be run at a state later effects have moved, and what it withdraws there is its own contribution and nothing else:

Theorem 20. Let $e_1, \cdots, e_n \in \mathfrak{E}_\Gamma^*$ be pairwise independent and applied in order from γ_0. Write $f_i \coloneqq \mathrm{pr}_1 \circ e_i$, let $\delta_i \coloneqq f_i(\delta_{i-1})$ with $\delta_0 \coloneqq \gamma_0$, and let $g_i \coloneqq \mathrm{pr}_2(e_i(\delta_{i-1}))$ be the inverse e_i yields where it is applied. Fix j and write $\delta'_i \coloneqq \big(f_i \circ \cdots \circ f_{j+1}\big)\big(\delta_{j-1}\big)$ for the states of the sequence with e_j omitted, so that $\delta'_j = \delta_{j-1}$. Then for every u with $j \leq u \leq n$,

1. $\delta_u = f_j(\delta'_u)$ and $g_j(\delta_u) = \delta'_u$;
2. each e_i with $i > j$ yields at δ'_{i-1} the same inverse g_i it yields at δ_{i-1}.

Proof.

1. The first equation is an induction on u. At $u = j$ it reads $\delta_j = f_j\big(\delta_{j-1}\big)$, which is the definition of δ_j. For the inductive step, $\delta_{u+1} = f_{u+1}(\delta_u) = f_{u+1}\big(f_j(\delta'_u)\big) = f_j(f_{u+1}(\delta'_u)) = f_j(\delta'_{u+1})$, the middle equality being clause (1) of Definition 19 for e_{u+1} and e_j, which are distinct effects of the family since $u + 1 > j$. For the second equation, clause (1) carries g_j out through the forward maps applied after e_j, leaving the witness of e_j to be used at the one state it holds at:
$$g_j(\delta_u) = \big(g_j \circ f_u \circ \cdots \circ f_{j+1}\big)(\delta_j) = \big(f_u \circ \cdots \circ f_{j+1}\big)\big(g_j\big(f_j(\delta_{j-1})\big)\big) = \delta'_u$$
the last equality resting on $g_j\big(f_j(\delta_{j-1})\big) = \delta_{j-1}$, which is the witness Definition 8 requires of e_j at δ_{j-1}.

2. By (1) the state δ_{i-1} is $f_j(\delta'_{i-1})$, and $f_j \in \mathfrak{M}(e_j)$, so clause (2) of Definition 19 for e_i and e_j gives $\mathrm{pr}_2\big(e_i\big(f_j(\delta'_{i-1})\big)\big) = \mathrm{pr}_2(e_i(\delta'_{i-1}))$. □

Clause (1) locates the state an inverse reaches: it is the state the same sequence would have reached had the effect never been applied, whatever effects were applied after it. Clause (2) locates the inverses the others hold there, and together the two let the theorem be applied again to the shorter sequence:

Corollary 21. Let $e_1, \cdots, e_n \in \mathfrak{E}^*_\Gamma$ be pairwise independent and applied in order from γ_0, and let $g_1, \cdots, g_n$ be as above. Applying the n inverses at δ_n in the order of any permutation of $\{1, \cdots, n\}$ reaches γ_0.

Proof. By downward induction on n. Let the permutation begin with j. By Theorem 20(1) applying g_j at δ_n reaches δ'_n, the state the sequence with e_j omitted reaches, and by Theorem 20(2) the inverses the remaining effects yielded there are the g_i in hand. That sequence is pairwise independent, being a subfamily, so the induction hypothesis applies to it and to the rest of the permutation; the empty sequence reaches γ_0. □

LIFO order is one such permutation, and Theorem 16 reverts in it with no hypothesis at all. What independence buys is every other order, and with it the sequence that interleaves several components, which Section 4.4.2 carries to a trace of a whole system.

Together, these constructions constitute *revertible effects*: each effect function in $\mathfrak{E}^*_\Gamma$ explicitly provides its own inverse, effect tracks these inverses on the effect context $\partial\Gamma$, and the $\diamond$ operation composes them while preserving revertibility. What they deliver is *local temporal composability*, local in that the guarantee is read of one component's effects taken by themselves. We take that to be the following criterion: for every sequence of effect functions a component applies, the accumulator recovers the context it began at (Theorem 7), and reverting the sequence hands each inverse the state its own application ran against (Theorem 16). Loading a component is applying such a sequence and accumulating its inverses in φ; unloading it is applying φ.

Two things the criterion leaves out, and both arrive once several components are in play: reverting out of the order the accumulator imposes, and a sequence that interleaves the effects of others. Independence delivers them (Corollary 21), and it is a condition on the effects rather than a property of the construction, Section 3.3.2 being where the discipline that meets it is identified and Section 4.4.2 where the guarantee is read of a whole system's trace. Where independence fails, the order has to be carried elsewhere: within one component by the accumulator, which reverts in LIFO order whatever the effects (Section 4.3.2), and across components by a declared coeffect, which orders one activation against another (Section 4.3.1).

3.2. Reactive Coeffects

Spatial composability is the ability for components to declare dependencies on one another and for the system to resolve, provide, and withdraw those dependencies at runtime. This requires that dependency satisfaction be re-evaluated whenever the shared context changes, so that a component activates when its dependencies become available and deactivates when they are withdrawn. We therefore model dependencies of a component as a specification and classify each change to the context, against that specification, as activating, deactivating, or neutral. Classifying against the specification is what detects a change in satisfaction; responding to that classification is what drives activation and deactivation. We call such coeffects *reactive*: by

classifying context changes and driving activation and deactivation from them, correct coeffect ordering becomes a structural guarantee.

3.2.1. Coeffect Context

Traditional inversion-of-control (IoC) containers [38] typically model dependencies as simple key-value mappings. This section formalizes IoC as a coeffect context that synergizes with revertible effects to provide a mathematical foundation for dynamic composition.

Definition 22. Given a type family $\mathcal{V} : K \to \mathrm{Type}$, define the *coeffect context* as the dependent partial function type:

$$\Sigma := (k : K) \rightharpoonup \mathcal{V}_k \tag{20}$$

where $\sigma : \Sigma$ is a finite partial function assigning to each $k \in \mathrm{dom}(\sigma) \subseteq K$ a value of type $\mathcal{V}_k$. We write:

- $\sigma(k)$ for application (defined when $k \in \mathrm{dom}(\sigma)$);
- $\sigma[k \mapsto v]$ for the table binding v at k and agreeing with σ elsewhere;
- $\sigma \setminus k$ for restriction (defined when $k \in \mathrm{dom}(\sigma)$);
- $k \in \mathrm{dom}(\sigma)$ for membership.

The use of a type family $\mathcal{V}$ ensures that each dependency key k is associated with a specific value type $\mathcal{V}_k$, providing static type safety for dependency access. Extension and restriction carry preconditions, imposed by the operations below: a dependency cannot be provided twice ($k \notin \mathrm{dom}(\sigma)$ for extension) nor revoked if absent ($k \in \mathrm{dom}(\sigma)$ for restriction). A violated precondition is signalled as an error and produces no transition, so the effect algebra, which describes the transitions that do occur, applies to these operations unchanged. A reader preferring to internalize the failure may read every $\Sigma \rightharpoonup \Sigma$ below as $\Sigma \to \mathsf{Maybe}(\Sigma)$ and compose in the Maybe monad (Section 2.1), at the cost of replacing each identity by the partial identity on the operation's domain. Based on this context structure, we define two core operations:

Definition 23. The get and set operations on Σ are defined as:

$$\begin{array}{lcccccc} \mathrm{get} & : & (k : K) & \to & \Sigma & \rightharpoonup & \mathcal{V}_k \\ \mathrm{get} & = & k & \mapsto & \sigma & \mapsto & \sigma(k) \\ \mathrm{set} & : & (k : K) \times \mathcal{V}_k & \to & \Sigma & \rightharpoonup & \Sigma \times (\Sigma \rightharpoonup \Sigma) \\ \mathrm{set} & = & (k, v) & \mapsto & \sigma & \mapsto & (\sigma[k \mapsto v],\ \lambda\sigma'.\sigma' \setminus k) \end{array} \tag{21}$$

where $\mathrm{get}(k)$ requires $k \in \mathrm{dom}(\sigma)$ and $\mathrm{set}(k, v)$ requires $k \notin \mathrm{dom}(\sigma)$ as preconditions.

Notably, $\mathrm{set}(k, v)$ has type $\mathfrak{E}^*_\Sigma$, precisely an effect function on the coeffect context. We can therefore directly apply the effect machinery from Section 3.1: effect_Σ provides automatic tracking and recovery of dependency registrations. This is the synergy between reactive coeffects and revertible effects: coeffect operations are effects, and effects are revertible.

What get hands a component is a value, and what the component can do with that value is whatever the coeffect at that key provides. A key therefore carries more than a value type:

Definition 24. A *coeffect* at a key k is a triple $\left(\mathcal{V}_k, \tilde{k}, \mathcal{A}_k\right)$, where $\mathcal{V}_k$ is the value type of Definition 22, $\tilde{k}$ is an equivalence relation on $\mathcal{V}_k$ up to which values at k are compared (Section 3.3.2), and $\mathcal{A}_k$ is a set of *coeffect operations*, the operations the value bound at k provides to a component

holding it. An operation $a \in \mathcal{A}_k$ carries an argument type X_a and an outcome type B_a, and acts on the value alone:

$$a : X_a \to \mathcal{V}_k \rightharpoonup \mathcal{V}_k \times (\mathcal{V}_k \rightharpoonup \mathcal{V}_k) \times B_a \tag{22}$$

its first two constituents forming an effect function on $\mathcal{V}_k$ witnessed as Definition 8 requires, and its third an *outcome*. Each operation is required to *respect* $\underset{k}{\simeq}$: at $\underset{k}{\simeq}$-related values it is defined at both or at neither, and where defined it yields $\underset{k}{\simeq}$-related successors, inverses that again carry $\underset{k}{\simeq}$-related values to $\underset{k}{\simeq}$-related values, and equal outcomes. An operation acts on the coeffect context through its *lift*

$$a^{\Sigma}(x)(\sigma) \coloneqq \mathbf{let}\ (v, g, b) = a(x)(\sigma(k))\ \mathbf{in}\ (\sigma[k \mapsto v],\ \lambda\sigma'.\sigma'[k \mapsto g(\sigma'(k))],\ b) \tag{23}$$

defined when $k \in \mathrm{dom}(\sigma)$, whose first two constituents are an effect function on Σ.

Typing an operation of k on $\mathcal{V}_k$ is what confines it to the binding at k: the lift reads and writes that binding and leaves every other key as it stands, so no side condition is needed to say so. Where isolation is in force the binding it reaches is the one the realm resolves to (Definition 28), two keys sharing a realm sharing one binding. An operation whose behaviour turns on another key reads that key's value into its argument X_a, and the reactive discipline of the next subsection is what holds the value fixed for as long as the component that read it runs (Theorem 63).

3.2.2. Specification and Notification

The preceding definitions describe how individual dependencies are registered and accessed. Accessing an absent dependency, however, is a runtime failure. A component should therefore activate only once all the dependencies it declares are present, rather than accessing them optimistically and failing when one is missing. This raises two questions: whether a component's declared dependencies are jointly satisfied, and how the system should respond when that status changes. The coeffect context Σ carries a natural observational structure that makes both questions tractable: for any coeffect specification $d \subseteq K$, define the *satisfaction predicate*:

$$\sigma \vDash d \coloneqq \forall k \in d.\ k \in \mathrm{dom}(\sigma) \tag{24}$$

This predicate is decidable (since $\mathrm{dom}(\sigma)$ is finite). Since all mutations to σ pass through effect functions (whose inverses recover the previous domain), changes to satisfaction are detectable at each effect boundary. This is the algebraic basis of reactivity: the effect system guarantees that every coeffect change is observed.

Definition 25. A *coeffect specification* is:

$$\mathfrak{D}_{\Sigma} \coloneqq \mathsf{Set}(K) \tag{25}$$

representing the set of dependencies a component declares from the environment.

What makes this specification reactive is how it classifies state transitions. Any effect that transforms σ to σ' can be classified by a specification $d \in \mathfrak{D}_{\Sigma}$ according to whether d's satisfaction status is altered:

Definition 26. Given a coeffect specification $d \subseteq K$ and states $\sigma, \sigma' \in \Sigma$, define:

$$\mathrm{notify}_d(\sigma, \sigma') := \begin{cases} \text{activating} & \text{if } \sigma \not\models d \wedge \sigma' \models d \\ \text{deactivating} & \text{if } \sigma \models d \wedge \sigma' \not\models d \\ \text{neutral} & \text{otherwise} \end{cases} \tag{26}$$

This is well-defined because $\sigma \models d$ is decidable and all state transitions are mediated by effect functions. The reactive invariant is: an activating transition triggers execution of the component's effects (with full effect tracking), whereas a deactivating transition triggers recovery by applying the accumulator. The precise operational semantics of these transitions depend on their interaction with control flows, and are developed in Section 4.

What set and notify deliver together is *local spatial composability*, local in the same sense as before, the guarantee being read of one component's coeffects taken by themselves. We take that to be the following criterion: a component activates only at a state satisfying its specification, so it never reads a binding that is absent, and every change to the context is classified against that specification, so a loss of satisfaction is detected where it happens and drives a deactivation. Both halves are immediate from the definitions above, satisfaction being a precondition checked where the component would activate and notify_d being defined at every transition.

The criterion covers one direction of the coeffect ordering and not the other. If component A provides a key k and component B declares $k \in d_B$, then B can activate only after A has activated and provided k, since $\sigma \models d_B$ requires $k \in \mathrm{dom}(\sigma)$. The converse fails: unloading A removes k from $\mathrm{dom}(\sigma)$ and so breaks B's satisfaction, but a notification cannot by itself keep k readable for as long as B's own teardown needs it, nor hold A's recovery back until B has finished. Ordering a withdrawal after the deactivations it causes is a condition on other components rather than on the one acting, so it belongs to the global form of the guarantee, and Section 4.3.1 supplies the machinery it takes.

3.2.3. Isolation and Interception

The basic coeffect context Σ models a flat dependency table. In practice, however, the system may need to bind distinct values to the same logical dependency for different components. This section extends the coeffect context with two mechanisms: *coeffect isolation* (the same key resolves differently in different contexts) and *coeffect interception* (cross-cutting behavior on dependency access).

Realization. The two mechanisms differ from get and set in what they act on. A provision writes the shared table every component reads, so it is an effect on that table and carries an inverse to withdraw it. Isolation and interception instead adjust how a key is *resolved* for the components under one context, leaving the table itself as it stands. Typing an operation as an effect fixes its *denotation*, a successor state paired with an inverse, but not its *realization*, which determines how that inverse is carried out.

Definition 27. An effect function on a context admits two *realizations*:

- *In-place* realization mutates the context and returns a nontrivial inverse; the successor aliases the input, and recovery runs the inverse to undo the mutation.
- *Derived* realization leaves the input intact and returns a fresh context deriving from it, with the identity as its inverse; recovery discards the derived context. A context derived from another is what the recursive structure of Definition 32 carries.

In a purely functional setting the two coincide, and an imperative host may choose either per operation; Section 5.1.2 implements both. Isolation and interception are given derived realiza-

tion outright: each produces a fresh context whose own table differs from the inherited one, so each is typed below as a map from context to context rather than as an effect function. Nothing in the shared table changes, so there is no inverse to track and nothing for Definition 12 to lift, and recovery discards the derived context along with the adjustment it carried. Assignment on a derived table overrides whatever the inherited table held at the key, which is why neither operation carries a precondition.

Coeffect Isolation. By introducing *isolation realms*, coeffect isolation allows the same dependency to bind to different values in different contexts. This has broad applications in multi-tenant systems, testing environments, and component sandboxes.

Definition 28. Define the coeffect context with isolation as:

$$\Sigma^{\text{iso}} := (K \rightharpoonup R) \times ((r : R) \rightharpoonup \mathcal{V}_r) \tag{27}$$

It can be represented as a pair (ρ, σ), where:

- $\rho : K \rightharpoonup R$ is the isolation realm table, assigning a realm identifier to each isolated key; a key outside $\mathrm{dom}(\rho)$ resolves to its own realm, so we write $\rho(k) = k$ there ($R \supseteq K$);
- $\sigma : (r : R) \rightharpoonup \mathcal{V}_r$ is the dependency table, a partial dependent function from realm identifiers to typed values.

The two-layer mapping structure decouples the logical layer from the storage layer, making dependency access context-aware. When accessing a key k, the system first resolves $\rho(k)$ to obtain a realm identifier r, then accesses $\sigma(r)$ for the actual value.

Definition 29. The get, set, and $\mathrm{isolate}$ operations on Σ^{iso} are:

$$\begin{array}{rcccccc}
\mathrm{get} & : & (k : K) & \to & \Sigma^{\text{iso}} & \rightharpoonup & \mathcal{V}_{\rho(k)} \\
\mathrm{get} & = & k & \mapsto & (\rho, \sigma) & \mapsto & \sigma(\rho(k)) \\
\mathrm{set} & : & (k : K) \times \mathcal{V}_{\rho(k)} & \to & \Sigma^{\text{iso}} & \rightharpoonup & \Sigma^{\text{iso}} \times \left(\Sigma^{\text{iso}} \rightharpoonup \Sigma^{\text{iso}}\right) \\
\mathrm{set} & = & (k, v) & \mapsto & (\rho, \sigma) & \mapsto & ((\rho, \sigma[\rho(k) \mapsto v]),\ \lambda(\rho', \sigma').(\rho', \sigma' \setminus \rho'(k))) \\
\mathrm{isolate} & : & K \times R & \to & \Sigma^{\text{iso}} & \to & \Sigma^{\text{iso}} \\
\mathrm{isolate} & = & (k, r) & \mapsto & (\rho, \sigma) & \mapsto & (\rho[k \mapsto r], \sigma)
\end{array} \tag{28}$$

where get and set carry the preconditions of Definition 23 transported along ρ, namely $\rho(k) \in \mathrm{dom}(\sigma)$ and $\rho(k) \notin \mathrm{dom}(\sigma)$. The context that $\mathrm{isolate}(k, r)$ derives assigns the realm r to k and inherits the dependency table unchanged, so a key already isolated is reassigned rather than refused.

The coeffect isolation mechanism essentially implements a runtime ad-hoc polymorphism system. Through isolation realm identifiers, the same dependency key can resolve to entirely different values in different contexts, and this polymorphism can be dynamically adjusted at runtime. Compared to traditional dependency injection, coeffect isolation provides finer-grained control, enabling customized isolation for specific components; set remains an effect function ($\mathfrak{E}^*_{\Sigma^{\text{iso}}}$) and thus inherits revertibility, whereas $\mathrm{isolate}$ needs none, deriving a context instead of writing the shared table.

Coeffect Interception. The second mechanism, *coeffect interception*, attaches cross-cutting metadata to dependency access, adding behavior without modifying the dependency value. This metadata can be either context-carried or component-declared, so we extend both the coeffect context and the coeffect specification:

Definition 30. Define the coeffect context and specification with interception as:

$$\begin{aligned}\Sigma^{\text{inter}} &:= ((k : K) \to \mathcal{M}_k) \times ((k : K) \rightharpoonup (\mathcal{M}_k \to \mathcal{V}_k)) \\ \mathfrak{D}^{\text{inter}} &:= (k : K) \rightharpoonup \mathcal{M}_k\end{aligned} \tag{29}$$

The context Σ^{inter} is a pair (ι, σ): ι is the *context-carried* metadata installed on the context itself, empty (ϵ_k) by default; and σ maps each key k to a *provider function* from metadata $\mathcal{M}_k$ to value $\mathcal{V}_k$. A specification $d \in \mathfrak{D}^{\text{inter}}$ carries the *component-declared* metadata, assigning each key its metadata $d(k)$, with $\mathrm{dom}(d)$ serving as the dependency set. Each key equips its metadata with a monoid $(\mathcal{M}_k, \oplus_k, \epsilon_k)$: the merge $\oplus_k$ is associative with identity ϵ_k (the empty metadata).

Definition 31. The get, set, and intercept operations on Σ^{inter} are:

$$\begin{array}{rcccccc}
\mathrm{get} & : & (k : K) \times \mathcal{M}_k & \to & \Sigma^{\text{inter}} & \rightharpoonup & \mathcal{V}_k \\
\mathrm{get} & = & (k, \mu) & \mapsto & (\iota, \sigma) & \mapsto & \sigma(k)(\mu \oplus_k \iota(k)) \\
\mathrm{set} & : & (k : K) \times (\mathcal{M}_k \to \mathcal{V}_k) & \to & \Sigma^{\text{inter}} & \rightharpoonup & \Sigma^{\text{inter}} \times (\Sigma^{\text{inter}} \rightharpoonup \Sigma^{\text{inter}}) \\
\mathrm{set} & = & (k, \psi) & \mapsto & (\iota, \sigma) & \mapsto & ((\iota, \sigma[k \mapsto \psi]),\ \lambda(\iota', \sigma').(\iota', \sigma' \setminus k)) \\
\mathrm{intercept} & : & (k : K) \times \mathcal{M}_k & \to & \Sigma^{\text{inter}} & \to & \Sigma^{\text{inter}} \\
\mathrm{intercept} & = & (k, \nu) & \mapsto & (\iota, \sigma) & \mapsto & (\iota[k \mapsto \iota(k) \oplus_k \nu], \sigma)
\end{array} \tag{30}$$

where get and set carry the preconditions of Definition 23 on the provider table, namely $k \in \mathrm{dom}(\sigma)$ and $k \notin \mathrm{dom}(\sigma)$. The context that $\mathrm{intercept}(k, \nu)$ derives merges ν onto the metadata inherited at k and inherits the provider table unchanged.

When a component with specification d accesses key k, the system evaluates $\sigma(k)(d(k) \oplus_k \iota(k))$: the component-declared metadata is merged with the context-carried metadata ι, and the provider function is applied to the result. This merge follows each key's own semantics (e.g. scalar fields are overwritten, set-valued fields unioned) and is right-biased, so $\iota(k)$ takes priority and can override the component's declaration, letting an enclosing context constrain how a component uses a coeffect without modifying that component (e.g. Section 6.3).

3.3. The Context Paradigm

Section 3.1 and Section 3.2 each act on a context, the first as the carrier of effects and the second as the carrier of coeffects, leaving open what a single context carrying both looks like. This section gives that unification a concrete construction, assembles from the coeffects an observational equivalence that supplies the effect independence Section 3.1.3 leaves open, and argues that the resulting context type constitutes a programming paradigm in its own right.

3.3.1. Unified Context

For a context Γ, the effect context $\partial\Gamma$ (Section 3.1) provides a higher-level abstraction, carrying the previous-level context and that level's accumulator (Definition 2). Making this structure recursive and combining it with the coeffect context Σ yields the following type:

Definition 32. The context type Γ_∞ is defined as:

$$\Gamma_\infty := \mu\Gamma.\ \Gamma \times (\Gamma \to \Gamma) \times \Sigma \tag{31}$$

where the three projections are:

- Γ: the current context state (recursive);
- $\Gamma \to \Gamma$: the accumulator, which recovers this level's effects;
- Σ: the coeffect context carrying dependency information.

Under this definition, effect maps $\mathfrak{E}_{\Gamma_\infty}$ to itself, unifying the ∂-tower into a single self-similar type. The coeffect context Σ is structurally integrated: dependency operations (set, get) act on Σ, and the accumulator tracks their reversal. Since the type family $\mathcal{V}$ underlying Σ is unconstrained, any state the system needs to share across components can be encoded as a dependency with an appropriate value type—Σ subsumes all shared mutable states, not just inter-component dependencies. Every interaction between a component and its environment passes through this single entity.

Hierarchical composition. The recursive structure of Γ_∞ supports hierarchical control: a parent context aggregates multiple child-level effects, forming a tree-shaped control structure that maintains modularity while enabling unified cross-level management. The effect transformation realizes a literal "plug-in" metaphor:

- Loading a component corresponds to executing its effects (plugging in);
- Unloading a component corresponds to recovering its effects (unplugging, without affecting other running components);
- Components at different levels of the hierarchy are independently loadable and unloadable; a parent context aggregates and manages the effects of all its children, enabling arbitrarily nested composition.

3.3.2. Observational Equivalence

The recovery guarantee of Section 3.1 asserts an equality of states (Theorem 7), which is an idealization, because the physical state cannot be recovered as it stood. For example, `free` releases a block to the allocator without restoring the layout the heap had before `malloc`; and a generative name is not restored by the inverse that discards it, since the next creation draws a fresh one [39]. The equalities of Section 3 are therefore to be read up to an equivalence $\simeq$, and we take $\simeq$ to be an *observational equivalence*: two states are related when no observer can distinguish them. Comparing behaviour rather than representation is the established route to program equivalence [40], and the relation such a comparison yields depends on what the observer is given to work with [41]. What an observer of a context is given is the coeffects it carries, each of which arrives with an equivalence of its own (Definition 24), so the relation on a context is assembled from theirs. Assembling it is the business of this subsection, and quotienting by it is what buys the independence Section 3.1.3 asks for.

Definition 33. Two coeffect contexts are related when they bind the same keys to related values, and two states of a context when their coeffect projections are:

$$\begin{aligned} \sigma \simeq \sigma' &\coloneqq \mathrm{dom}(\sigma) = \mathrm{dom}(\sigma') \wedge \forall k \in \mathrm{dom}(\sigma).\ \sigma(k) \underset{k}{\simeq} \sigma'(k) \\ \gamma \simeq \gamma' &\coloneqq \sigma_\gamma \simeq \sigma_{\gamma'} \end{aligned} \tag{32}$$

writing σ_γ for the coeffect projection of γ (Definition 32).

The part of a state that no key binds is thereby forgotten, and forgetting it is what lets Theorem 7 be read up to $\simeq$ at all: the heap layout and the generative name of the examples above lie outside the relation unless some key binds them. What Section 3.2.2 needs of $\simeq$ follows rather than being assumed. Related states have the same domain, so they agree on the

satisfaction predicate $\sigma \models d$ and on the classification notify_d of Definition 26, and reactivity is a property of $\Sigma/\simeq$.

Calling the relation observational is a claim about each $\underset{k}{\simeq}$, namely that it separates no more than the operations of k can tell apart. An observer of a value runs those operations and reads their outcomes.

Definition 34. Let V carry a set $\mathcal{A}$ of operations in the sense of Definition 24, and write $\mathfrak{M}(a)$ for the transformation monoid (Definition 17) of the effect functions $a(x)$ over every argument $x : X_a$. A *test* over $\mathcal{A}$ is a finite word over the generators of the monoids $\mathfrak{M}(a)$, $a \in \mathcal{A}$, each letter applied to the value the letters before it left; its *outcomes* are those the letters that are forward maps of operations yield along the way, and it is undefined where a precondition fails. Values $v, v' : V$ are *indistinguishable*, written $v \underset{\mathcal{A}}{\approx} v'$, when every test over $\mathcal{A}$ is defined at both or at neither and yields the same outcomes at both.

Lemma 35. Indistinguishability is the coarsest relation the operations respect. That is,

1. every operation of $\mathcal{A}$ respects $\underset{\mathcal{A}}{\approx}$ in the sense of Definition 24;
2. every equivalence that every operation of $\mathcal{A}$ respects is contained in $\underset{\mathcal{A}}{\approx}$.

Every admissible choice of $\underset{k}{\simeq}$ is therefore contained in $\underset{\mathcal{A}_k}{\approx}$, and $\underset{\mathcal{A}_k}{\approx}$ is itself admissible.

Proof.

1. Let $v \underset{\mathcal{A}}{\approx} v'$ and let $a \in \mathcal{A}$ be applied to an argument. Prefixing a test by one letter is again a test, so the values the forward map reaches are indistinguishable, as are the values any one yielded inverse reaches from indistinguishable arguments; the one-letter test gives definedness at both or neither and equality of the outcome.
2. Let R be such an equivalence and vRv'. Each letter of a test is a forward map or a yielded inverse of an operation, and respect carries R along either, keeping the values reached related and the outcomes equal at every letter. Hence every test agrees at v and v'. □

Substituting $\simeq$ for $=$ throughout is not by itself enough, because an effect function returns an inverse as well as a state, and two states that $\simeq$ identifies have to yield inverses $\simeq$ identifies as well.

Definition 36. A map $f : \Gamma \to \Gamma$ *respects* $\simeq$ when

$$\forall \gamma, \gamma' \in \Gamma. \quad \gamma \simeq \gamma' \Rightarrow f(\gamma) \simeq f(\gamma') \tag{33}$$

Two maps are related when they agree at every state, and two pairs in $\partial\Gamma$ when both components are:

$$\begin{aligned} f \simeq g &\coloneqq \forall \gamma \in \Gamma.\ f(\gamma) \simeq g(\gamma) \\ (\delta, g) \simeq (\delta', g') &\coloneqq \delta \simeq \delta' \wedge g \simeq g' \end{aligned} \tag{34}$$

A map respecting $\simeq$ is one that descends to $\Gamma/\simeq$, and two maps related by $\simeq$ are two that descend to the same map there. An effect function needs both: the first so that the state it computes is determined on the quotient, the second so that the inverse it returns is.

Definition 37. Read Definition 8 up to $\simeq$: an $e \in \mathfrak{E}_\Gamma$ lies in $\mathfrak{E}^*_\Gamma$ when e respects $\simeq$ as a map $\Gamma \to \partial\Gamma$ and, writing $(\delta, g) = e(\gamma)$, for every $\gamma \in \Gamma$

1. $g(\delta) \simeq \gamma$;

2. g respects $\simeq$.

Taking $\simeq$ to be equality on Γ recovers Definition 8.

Lemma 38. With $\mathfrak{E}^*_\Gamma$ read as in Definition 37, every equality of states asserted in Section 3.1 holds with $=$ replaced by $\simeq$, and the accumulator of every state reachable from $(\gamma_0, \mathrm{id}_\Gamma)$ respects $\simeq$.

Proof. An accumulator is a composition of inverses, each respecting $\simeq$ by Definition 37(2), and a composition of maps respecting $\simeq$ respects $\simeq$, the base case being id_Γ. The proofs of Section 3.1 then go through unchanged, respect being what carries a relation through an inverse: from $g_2(\delta_2) \simeq \delta_1$ and $g_1(\delta_1) \simeq \gamma$ respect gives $(g_1 \circ g_2)(\delta_2) \simeq \gamma$, which is the step each composition of inverses takes, and the soundness invariant of Theorem 7 reads $\varphi(\gamma) \simeq \gamma_0$ by that step. □

The commutation Definition 19 asks for is read up to $\simeq$ by the same lemma, and reading it that way is what makes it attainable at all: two operations may leave values that $\underset{k}{\simeq}$ identifies and still count as commuting. Of two operations it asks one thing more than of the effect functions their lifts induce, an operation yielding an outcome as well.

Definition 39. Operations a and a' are *independent* when their lifts are independent as effect functions (Definition 19) at every pair of arguments, and neither one's transformations disturb the outcome the other yields:

$$\forall x : X_a,\, g \in \mathfrak{M}(a'^{\Sigma}),\, \sigma \in \Sigma. \quad \mathrm{pr}_3(a^{\Sigma}(x)(g(\sigma))) = \mathrm{pr}_3(a^{\Sigma}(x)(\sigma)) \tag{35}$$

and the same with a and a' exchanged, writing $\mathfrak{M}(a^{\Sigma})$ for the transformation monoid of the lifts $a^{\Sigma}(x)$ over every argument as Definition 34 writes $\mathfrak{M}(a)$ for that of the operation itself. A key k is *commutative* when any two operations of $\mathcal{A}_k$ are independent, an operation being held independent of itself as well.

Across distinct keys the condition holds outright.

Theorem 40. Operations at distinct keys are independent.

Proof. Let a lie in $\mathcal{A}_k$ and a' in $\mathcal{A}_{k'}$ with $k \neq k'$. By Definition 24 every generator of $\mathfrak{M}(a^{\Sigma})$ is of the form $\sigma \mapsto \sigma[k \mapsto u(\sigma(k))]$ for a map u on $\mathcal{V}_k$, being either the lift of a forward map or the lift of a yielded inverse, and likewise for a' at k'. Two such maps commute, each reading and writing one key alone and the two keys differing, and Lemma 18(1) extends the commutation from the generators to the two monoids. For the second condition, what a^{Σ} yields at σ, inverse and outcome alike, is determined by $\sigma(k)$, which every generator of $\mathfrak{M}(a'^{\Sigma})$ leaves as it stands. □

A key whose value is a table of entries added and removed independently is commutative, registration of a route or of an event listener being the representative case: two registrations in either order leave a table that answers every test alike, and either registration can be withdrawn while the other stands. A key whose value is an ordered chain is not, since a middleware inserted before another sees a different request, and neither order can be withdrawn without disturbing the other. The allocator of the opening example divides by what its interface publishes. Where the handles it hands out are compared by no operation of the key, $\underset{k}{\simeq}$ may relate two heaps up to a renaming of handles, which is how CompCert relates the memory states of a program and of its translation [42], and allocation is commutative; where the addresses are outcomes compared by equality, no admissible $\underset{k}{\simeq}$ makes the two orders of allocation agree, and the key is not commutative.

What a component performs is a sequence of operations in which each may depend on what the ones before it yielded, and effect functions of that shape are what the theorem below speaks of.

Definition 41. The *coeffect-mediated* effect functions form the least set $\mathfrak{E}^{\mathcal{A}}_{\Sigma} \subseteq \mathfrak{E}_{\Sigma}$ that contains the unit η_{Σ} and is closed under the following: for a key k, an operation $a \in \mathcal{A}_k$, an argument $x : X_a$, and a family $(e_b)_{b \in B_a}$ of members,

$$\sigma \mapsto \mathbf{let}\ (\delta, s, b) = a^{\Sigma}(x)(\sigma)\ \mathbf{in}\ \mathbf{let}\ (\varepsilon, t) = e_b(\delta)\ \mathbf{in}\ (\varepsilon,\ s \circ t) \tag{36}$$

is again a member. Each stage performs one operation and chooses what follows it by the outcome, so an argument may depend on the outcomes already obtained. The operations *occurring in* a member are the ones its stages perform, over every choice of outcome.

Theorem 42. Let $e_1, e_2 \in \mathfrak{E}^{\mathcal{A}}_{\Sigma}$ and let every key at which operations of both occur be commutative (Definition 39). Then e_1 and e_2 are independent (Definition 19).

Proof. By induction on the construction of Definition 41, $\mathfrak{M}(e_i)$ lies in the submonoid generated by the generators of the operations occurring in e_i: the unit generates the trivial monoid, and a stage is a $\diamond$-composite of $a^{\Sigma}(x)$ with a member, to which Lemma 18(2) applies.

For clause (1) of Definition 19 it is therefore enough, by Lemma 18(1), that a generator of an operation occurring in e_1 commute with a generator of one occurring in e_2. Where the two operations lie at distinct keys this is Theorem 40, and where they lie at one key that key carries operations of both and is commutative by hypothesis.

For clause (2), take $g \in \mathfrak{M}(e_2)$, a composite of generators of the operations occurring in e_2, and induct on the construction of e_1. The unit yields id_{Σ} at every state. At a stage, let $(\delta, s, b) = a^{\Sigma}(x)(\sigma)$ and $(\varepsilon, t) = e_b(\delta)$, so that the stage yields $s \circ t$ at σ. Independence of the operations, applied to one generator of g at a time, yields s and b again at $g(\sigma)$, so the same continuation e_b is chosen, and clause (1) puts the state it runs from at $g(\delta)$, where the induction hypothesis yields t again. The stage therefore yields $s \circ t$ at $g(\sigma)$. □

Every interaction between a component and its environment passes through the context, and the type family $\mathcal{V}$ is unconstrained, so a system may bind every location it shares across components at a key of its own (Section 3.3.1). A component's effect function is then the lift of a coeffect-mediated one along the coeffect projection, and independence transfers to that lift, whose transformations move the projection alone. The assumption Section 3.1.3 leaves open is met that way, and with it the temporal composability of a whole system of components.

What the decomposition divides is a computation's commuting part from its order-sensitive part. The commuting part is carried by the effects: a component performs them in whatever order its task calls for, and Corollary 21 reverts them in whatever order the system finds convenient, no two components constraining each other. The order-sensitive part is carried by the coeffects, since a key whose operations do not commute is one whose order has to be imposed from outside the effects, and two places are available for imposing it. Within one component the accumulator imposes it, reverting in LIFO order whatever the effects (Theorem 16). Across components a declared coeffect imposes it, one component providing what another declares and the provision preceding the declaration's satisfaction (Section 3.2.2). Composability is thereby had at the grain of components rather than of single effects, which is the scale Section 4 works at.

Two limits of the theorem are worth naming. Binding every shared location at a key is the paradigm's discipline and not a property of the construction, so a location the system cannot reify as a coeffect lies outside the boundary of Section 6.1 and outside the theorem with it. And commutativity of a key is a property of the interface that key publishes, so meeting it is an obligation on the component providing the key rather than on the components consuming it.

3.3.3. Situating the Context Paradigm

Programming paradigms differ fundamentally in how they handle side effects. Two established poles define the spectrum:

Explicit state threading (functional). To preserve referential transparency, purely functional languages model side effects as explicit transformations on state. The `State` monad $S \to (A, S)$ [23] threads an environment through every computation. This approach yields strong compositional guarantees: effects are visible in types and amenable to equational reasoning. However, it imposes significant ergonomic costs: every function in the call chain must accept and return the state parameter, even when it merely passes the state through unchanged. As the number of effect dimensions grows (logging, configuration, I/O), monadic stacking or effect-handler boilerplate proliferates.

Implicit mutation (imperative/OOP). Mainstream imperative languages permit components to modify shared state and access dependencies without explicit declaration at the call site. On the effect side, a representative example is React's `useEffect` hook: it registers a persistent side effect on the component's internal fiber, yet neither the effect target nor the registration mechanism appears as an explicit parameter—identification relies on call-order position within hidden runtime state. On the coeffect side, Java's service locator pattern (e.g., Spring's `ApplicationContext.getBean(...)`) retrieves dependencies from a process-wide registry at runtime, requiring null checks and type casts at each call site; dependency relationships are implicit and scattered across the codebase. More generally, understanding how `f()` modifies or depends on the system requires reading its implementation transitively. Refactoring becomes fragile because moving or removing a call may silently break distant invariants.

The context paradigm combines the traceability of the functional approach with the ergonomics of the imperative approach. Effects and coeffects are both mediated through an explicit context parameter. Each operation is therefore attributable to the specific context on which it was invoked, and hence to the component that context belongs to.

Beyond combining the strengths of both poles, the context paradigm lets the developer handle each effect and dependency individually and composes them into the system's behavior automatically. For revertible effects, the developer supplies the inverse of each atomic operation, and the inverse of any composite follows by composition (Section 3.1), so a component's teardown is derived from its loading rather than written alongside it. For reactive coeffects, a component declares only the dependencies it needs, and the runtime resolves and re-wires them automatically (Section 3.2), keeping them consistently wired as providers are added, removed, or replaced. In both directions, correctness that would otherwise rest on developer discipline becomes a structural property of the paradigm.

4. A Calculus of Dynamic Composition

Section 3 establishes spatial and temporal composability in their local form alone. Carrying them to a whole system takes a decomposition of the system into *components*, each pairing a coeffect specification with a witnessed effect function, so that every interaction with the shared environment is attributable to one of them. The sections below give that decomposition an operational semantics, and establishes spatial and temporal composability in their global form.

Section 4.1 and Section 4.2 present the smallest calculus in which the lifecycle can be given rules, one that takes each transition to be atomic, immediate, and infallible; Section 4.3 drops the three assumptions, atomicity once for each direction a transition may run in, admitting the forms of control flow a runtime interposes between the start of a transition and its end, and arrives at the calculus a real runtime implements; and Section 4.4 establishes the metatheory of that calculus, namely preservation, global temporal and spatial composability, progress, and confluence.

4.1. Components and Fibers

This section fixes the objects the rules act on: the *component*; the *fiber*, an instantiation of a component carrying a lifecycle state of its own; and the *registry*, which holds the fibers a state carries and from which the coeffect context is read off.

Components. A component is given as a triple, its coeffect side split into what it reads from the environment and what it provides to it.

Definition 43. A *component* over a context Γ carrying both effects and coeffects (Definition 32) is defined as:

$$\mathfrak{C}_\Gamma := \mathfrak{D}_\Gamma \times \mathfrak{P}_\Gamma \times \mathfrak{E}^*_\Gamma \tag{37}$$

representing a triple (d, p, e), where:

- $d : \mathfrak{D}_\Gamma$ is the coeffect specification of Definition 25, declaring the dependencies required from the environment;
- $p : \mathfrak{P}_\Gamma := \mathsf{Set}(K)$ is the *provision*, declaring the coeffect keys the component may provide, and no key outside p is one its effect function writes;
- $e : \mathfrak{E}^*_\Gamma$ is the witnessed effect function of Definition 8, defining the effects contributed when the component is active together with the inverse that withdraws them.

The two declarations are the two directions of one interface, d what the component reads from the environment and p what the component writes to the environment, and Section 4.2 admits no two fibers of one registry whose provisions meet. Subscripts are taken on Γ throughout, the coeffect context being one of its projections (Definition 32), so the $\mathfrak{D}_\Sigma$ of Definition 25 is written $\mathfrak{D}_\Gamma$ here.

Disjointness of provisions is where this chapter parts company with Section 3.2.3. The isolation of Definition 28 lets one key resolve through a realm table, so that two fibers may provide the same key in different realms; a calculus carrying realms would relax disjointness to disjointness within a realm and would resolve a declared key against the realm of the fiber declaring it. We do not introduce realms here, and read every key at one shared realm instead, which is what makes the disjointness above the right condition and each key's provider unique (Definition 45). What it restricts is how often a component may be instantiated: one with a non-empty provision has one fiber at a time, so the many instantiations below are of components

providing nothing, which is the common case of a component that only consumes, or that registers others.

A component instantiated in a running system is activated and deactivated over time, so it carries a *lifecycle state*, and a *transition* is what moves it from one lifecycle state to another: an *activation* executes e, accumulating side effects on the context, and a *deactivation* applies the accumulator to recover the context. In its simplest form the lifecycle is the two-state model of Figure 1, which Section 4.2 gives rules for; Section 4.3 refines it as each control-flow feature is admitted.

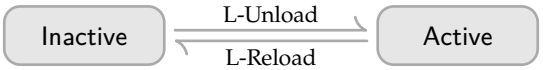


Figure 1 | Base component lifecycle

Fibers. One component may be instantiated many times over, each instantiation carrying a lifecycle state of its own. We name such an instantiation a *fiber*. A fiber records the component that produced it, the fiber it was instantiated under, the coeffects it provides, and where in its lifecycle it stands.

Definition 44. Fix a set $\mathfrak{N}$ of fiber names. A *fiber* instantiating the component $(d, p, e) \in \mathfrak{C}_\Gamma$ is a tuple $\langle d, p, e, \pi, \sigma, \tau, \theta \rangle$, where

- $d : \mathfrak{D}_\Gamma$, $p : \mathfrak{P}_\Gamma$, and $e : \mathfrak{E}^*_\Gamma$ are the coeffect specification, provision, and effect function of Definition 43;
- $\pi : \mathfrak{N} \cup \{\mathsf{root}\}$ is the *parent*, the fiber this one was instantiated under, or the root marker root;
- $\sigma : \Sigma$ is the fiber's own coeffect table (Definition 22), empty until it activates and written by its effects as they run;
- $\tau : \{\bot, \top\}$ is the *retirement* flag, $\bot$ in a fresh fiber and $\top$ once the orchestrator has retired the fiber;
- $\theta : \Theta_\Gamma$ is the *lifecycle state*, which in the two-state model of Section 4.2 is
$$\Theta_\Gamma := \mathsf{Inactive} \mid \mathsf{Active}(g, \omega) \tag{38}$$
where $g : \Gamma \to \Gamma$ is the accumulator and $\omega : d \to \mathfrak{N}$ the *committed view*.

The committed view ω sends each key the fiber declares to the name of the fiber that provided it when the transition committed. Section 4.3 replaces Θ_Γ by the extension that transitions in progress require; the rest of Definition 44 is given once for both, save that e is read at the richer effect type each layer of Section 4.3 introduces.

Registry. A state holds its fibers under their names, and both the identity of a fiber and the coeffect context of Section 3.2 are read off that arrangement.

Definition 45. Write $\mathfrak{F}_\Gamma$ for the set of fibers over Γ. A state $\gamma \in \Gamma$ carries a *registry*
$$F_\gamma : \mathfrak{N} \rightharpoonup \mathfrak{F}_\Gamma \tag{39}$$
a finite partial function whose parent pointers form a tree rooted at root, together with whatever else in Γ no fiber's σ names. We write $\gamma(n)$ for $F_\gamma(n)$, and abbreviate a field of $\gamma(n)$ by subscripting it with n where the state is clear, so that $d_n, p_n, e_n, \pi_n, \sigma_n, \tau_n, \theta_n$ are the fields of Definition 44 and g_n, ω_n the accumulator and committed view that θ_n carries; $\gamma[\theta_n \mapsto \theta']$, $\gamma[n \mapsto \langle \cdots \rangle]$, and $\gamma \setminus n$ are the states differing from γ in one field, one fiber, and the presence of one fiber respectively.

A fiber's name is what gives it an identity that survives its own mutation: every rule below rewrites the lifecycle state of one fiber and leaves the others alone, so the rule has to say which one, and two fields refer to fibers rather than describe them, the parent π and the committed view ω. Names are atoms: no rule computes one, inspects its structure, or relates two of them by anything but equality, and introducing a fiber simply draws one not already in use. This is the discipline of dynamically created local names [39], used here for fiber identity.

Each fiber owning a table means the coeffect context is derived rather than stored: it is what the active fibers jointly provide.

$$\sigma_\gamma := \bigcup \{ \sigma_m \mid m \in \mathrm{dom}(F_\gamma),\, \theta_m = \mathsf{Active}(-,-) \} \tag{40}$$

The union is well defined because a fiber writes only the keys it declares, $\mathrm{dom}(\sigma_n) \subseteq p_n$, and the provisions of distinct fibers are disjoint (Definition 43), so each $k \in \mathrm{dom}(\sigma_\gamma)$ lies in the table of exactly one Active fiber, whose name we write $\mathrm{provider}_k(\gamma) \in \mathfrak{N}$ and call the *provider* of k. Each key therefore has one possible provider, fixed by the provisions and not by the state. No rule writes σ_n directly: a fiber's provisions are the set operations its own effect function performs, which land in σ_n and so are already part of the state e_n returns, and they leave again with the accumulator. Only the coeffect part of an effect is recorded this way, because only the coeffect part is what other fibers declare against; effects that mutate state elsewhere in γ are tracked by g like any other, but no fiber can name them in a specification, so they contribute no ordering constraint.

The satisfaction relation of Section 3.2.2 then applies unchanged, with $\gamma \vDash d$ abbreviating $\sigma_\gamma \vDash d$. A key lies in $\mathrm{dom}(\sigma_\gamma)$ exactly when some Active fiber has installed it, its provision being the keys it may install rather than the ones it has, so $\gamma \vDash d$ already requires that every declared key have an Active provider. Taking the union over Active fibers alone is what lets a fiber cease to provide before it has withdrawn anything, which Section 4.3.1 turns into the ordering discipline.

4.2. The Base Calculus

This section gives the calculus of the two-state lifecycle of Figure 1 and nothing more: the target each fiber is compared against, and the five rules that move it.

Target views. The rules compare each fiber against a *target*, namely whether it ought to be running and against which resolution of its dependencies. The target is not a property of the fiber alone, since the keys a fiber declares are resolved against the whole state, so it is a predicate on that state.

Definition 46. The *target view* of n at γ maps each declared key to its provider, so it is a total map $d_n \to \mathfrak{N}$, and is $\bot$ when n ought not to be running at all:

$$\mathrm{target}_n(\gamma) := \begin{cases} \bot & \text{if } \tau_n \vee \neg(\gamma \vDash d_n) \\ (k \in d_n) \mapsto \mathrm{provider}_k(\gamma) & \text{otherwise} \end{cases} \tag{41}$$

A state is *quiescent* when every fiber has reached its target view:

$$\mathrm{quiet}(\gamma) := \forall n \in \mathrm{dom}(F_\gamma).\, \begin{cases} \mathrm{target}_n(\gamma) = \bot & \text{if } \theta_n = \mathsf{Inactive} \\ \mathrm{target}_n(\gamma) = \omega_n & \text{if } \theta_n = \mathsf{Active}(-, \omega_n) \end{cases} \tag{42}$$

The target answers to two things and to nothing else: retirement, through τ_n, and coeffect resolution, through $\gamma \vDash d_n$ and $\mathrm{provider}_k$, each declared key being read off σ_γ at the one shared realm of Definition 43.

The committed view of Definition 44 has the same type as the target view, and the lifecycle is driven by comparing them: ω_n is the resolution n activated against, $\mathrm{target}_n(\gamma)$ the one it should be running against, and every rule below fires on their agreeing or differing. Recording a provider rather than a value is what makes the comparison usable, since a different fiber providing an equal value would otherwise compare equal. The value a component reads is reached through the view, since the provider's table holds that value, and the implementation holds the map in `fiber.committed` and a hash of it in `fiber.target` (Section 5.1.3).

Rules. The base calculus takes each transition to be atomic, immediate, and infallible: an activation applies its effect function in one step, a deactivation applies the accumulator in one step, and both succeed in doing so. Section 4.3 drops all three.

Five rules generate two relations. An *orchestration* rule, prefixed O- and written $\gamma \Rightarrow \delta$, is an action the orchestrator may perform; its premises say when the action is legal, not when it occurs. A *lifecycle* rule, prefixed L- and written $\gamma \longrightarrow \delta$, is a step the system takes unprompted whenever its premises hold. A sequence of steps interleaves the two, and $\xrightarrow{*}$ below means lifecycle steps alone.

$$\frac{n \notin \mathrm{dom}(F_\gamma) \quad \pi \in \mathrm{dom}(F_\gamma) \cup \{\mathsf{root}\} \quad (d,p,e) \in \mathfrak{C}_\Gamma \quad \forall m \in \mathrm{dom}(F_\gamma).\ p \cap p_m = \varnothing}{\gamma \Rightarrow \gamma[n \mapsto \langle d,p,e,\pi,\varnothing,\bot,\mathsf{Inactive}\rangle]} \text{ O-Insert}$$

$$\frac{n \in \mathrm{dom}(F_\gamma)}{\gamma \Rightarrow \gamma[\tau_n \mapsto \top]} \text{ O-Retire}$$

$$\frac{\tau_n = \top \quad \theta_n = \mathsf{Inactive} \quad \forall m.\ \pi_m \neq n}{\gamma \Rightarrow \gamma \setminus n} \text{ O-Remove}$$

Insertion and retirement are the only external inputs: the orchestrator asks for a fiber to exist or to stop existing, and never sets its lifecycle state directly. O-Retire is unconditional on the fiber's state because retiring is a request, and the lifecycle rules are what carry it out. Retirement is separated from removal for the same reason: a retired fiber that is still Active must first be deactivated, and removing it earlier would discard the accumulator and leak. The premise $\forall m.\ \pi_m \neq n$ keeps the tree well-formed by removing children before their parent. The last premise of O-Insert is where the single-source discipline is imposed: a key has one possible provider because the orchestrator may not admit a second component declaring it.

$$\frac{\theta_n = \mathsf{Inactive} \quad \omega = \mathrm{target}_n(\gamma) \neq \bot \quad e_n(\gamma) = (\delta, g)}{\gamma \longrightarrow \delta[\theta_n \mapsto \mathsf{Active}(g,\omega)]} \text{ L-Reload}$$

$$\frac{\theta_n = \mathsf{Active}(g,\omega) \quad \mathrm{target}_n(\gamma) \neq \omega \quad g(\gamma) = \delta}{\gamma \longrightarrow \delta[\theta_n \mapsto \mathsf{Inactive}]} \text{ L-Unload}$$

L-Reload installs the committed view alongside the inverse; L-Unload applies the inverse and discards the committed view. Both are driven by the same comparison: L-Reload fires when a fiber holds no committed view and its target view is not $\bot$, L-Unload when the committed view it holds is not its target view. This is the reactive discipline of Section 3.2, read off a target that answers to retirement as well as to the coeffects: a transition is initiated whenever the target view changes, regardless of which of the two moved it.

Instantiation. A component may instantiate another while installing its effects, which is what a plugin host does when a plugin loads plugins of its own. The rules so far leave the

registry to the orchestration rules alone, so such an instantiation has nowhere to happen. One primitive gives it somewhere.

Definition 47. An application of e_n, or one of its iterations where Section 4.3.2 applies, may *register* a component $(d, p, e) \in \mathfrak{C}_\Gamma$. In place of a state map it takes the O-Insert of that component with $\pi = n$, and it yields as its inverse the O-Retire of the fiber so registered. The rule draws the name, subject to the freshness premise of O-Insert, and hands it to the effect function.

The inverse retires rather than removes, and the reason is that an inverse has to apply wherever it is reached. O-Remove carries premises, so an inverse built from it can fail to: a parent whose child is still Active could not run its accumulator, and no rule would move the child, since Definition 46 does not read the fiber tree. O-Retire has $n \in \mathrm{dom}(F_\gamma)$ as its only premise. The entry it leaves behind at the state the registration was taken is retired, Inactive($\perp$), and holds an empty table, which is the vestigial entry of Lemma 57: it differs from the absence of the fiber in control fields alone, and no rule tells the two apart.

Retiring a child sets τ and so takes its target view to $\perp$, after which the ordinary rules carry it back to Inactive. The parent is not made to wait, O-Retire being unconditional, so L-Unload applies to the parent whether or not the child has left. A grandchild is reached one level at a time, the child's own accumulator retiring what the child registered. Theorem 66 covers this cascade and the one Section 4.3.1 imposes along coeffects together.

Confinement. With the one exception in hand, the discipline an effect function is held to can be given. It bounds what an application writes, so that the rule applying it accounts for every other change, and what an application reads, so that a fiber sees the coeffects it declared and no more of the registry. Bounding the writes is what lets Section 4.4 read Table 1 as a complete inventory of them.

Definition 48. A map $f : \Gamma \to \Gamma$ is *confined to* n when for every $\gamma \in \Gamma$ with $n \in \mathrm{dom}(F_\gamma)$, writing $\delta = f(\gamma)$,

1. (*Writes.*) $\mathrm{dom}(F_\delta) = \mathrm{dom}(F_\gamma)$, $\delta(m) = \gamma(m)$ for every $m \in \mathrm{dom}(F_\gamma)$ with $m \neq n$, and $\delta(n)$ and $\gamma(n)$ differ in σ alone;
2. (*Reads.*) two states agreeing on σ_n, on the restrictions $\sigma_m|_{d_n}$ for every $m \in \mathrm{dom}(F_\gamma)$, and on the part of the state that no fiber's table names are carried by f to states agreeing on the same three.

An effect function e is *confined to* n when every application of it, and of each of its iterations where Section 4.3.2 applies, either registers a component (Definition 47) or has both its state map $\mathrm{pr}_1 \circ e$ and the inverse it yields confined to n. Every fiber's effect function is required to be confined to that fiber.

A registration writes the entry O-Insert writes, at the one name it draws, and nothing else; the O-Retire it yields as its inverse writes the τ of that name and nothing else. An application of either kind therefore writes no control field of a fiber already present, save that one τ, and reads none at all.

Clause (2) is why a component may read the values it declared: those lie in the tables of its providers, so an effect function that reads no table but σ_n would be unable to use its own coeffects. What it may not read is a table outside d_n, or any control field, which is what keeps a component from branching on the lifecycle state of a fiber it did not declare.

The rules are nondeterministic: several fibers may hold a committed view differing from their target view, and the relation commits to no order among them. They are also *reactive*

only, in that no rule mentions a scheduler; the steps are any sequence of rule applications, so a theorem proved over all such sequences holds for every scheduling policy a runtime might adopt.

4.3. Transitions in Progress

This section extends the base calculus in four settings. The first supplies something Section 3.2 requires and Section 4.2 cannot express, a deactivation spread over an interval its dependents may occupy; the other three drop the idealization that a transition is atomic, immediate, and infallible, none of which a transition in a real runtime is. What is dropped is that a whole transition is one step, not that a step is one application of one rule, and the four share one structural consequence, taken here once: a transition that is not a step needs a state to occupy while it is under way, one for each direction it may run in.

Definition 49. The lifecycle states of this section replace Θ_Γ by

$$\Theta_\Gamma \coloneqq \mathsf{Inactive}(\zeta) \mid \mathsf{Reloading}(i, g, \omega) \mid \mathsf{Active}(g, \omega) \mid \mathsf{Unloading}(g, \omega, \zeta) \tag{43}$$

where $i : \mathfrak{C}_\Gamma^{\mathrm{iter}*}$ is the remaining effect iterator (Definition 51 below), $g : \Gamma \to \Gamma$ the accumulator built so far, $\omega : d \to \mathfrak{N}$ the committed view, and $\zeta : \{\bot\} \cup \Xi$ the *outcome*, carried by Unloading as the one its deactivation is headed for and by Inactive as the one it reached, either $\bot$ or an error drawn from the set Ξ of errors that Section 4.3.4 supplies.

A fiber is *installed* when it is in one of the three states carrying an accumulator and a committed view, and *failed* when it carries an error outcome:

$$\mathrm{installed}_n(\gamma) \coloneqq \theta_n \neq \mathsf{Inactive}(-), \qquad \mathrm{failed}_n(\gamma) \coloneqq \exists \xi \in \Xi.\ \theta_n = \mathsf{Inactive}(\xi) \tag{44}$$

An installed fiber n *resolves* k *to* m when $\omega_n(k) = m$. The quiescence of Definition 46 is read on the wider state space as

$$\mathrm{quiet}(\gamma) \coloneqq \forall n \in \mathrm{dom}\big(F_\gamma\big).\ \begin{cases} \zeta \neq \bot \vee \mathrm{target}_n(\gamma) = \bot & \text{if } \theta_n = \mathsf{Inactive}(\zeta) \\ \mathrm{target}_n(\gamma) = \omega_n & \text{if } \theta_n = \mathsf{Active}(-, \omega_n) \\ \bot & \text{otherwise} \end{cases} \tag{45}$$

The definitions of Section 4.1 carry over to this state space, with two readings to fix. First, the Inactive of Section 4.2 is read as $\mathsf{Inactive}(\bot)$ in the conclusion of O-Insert and as $\mathsf{Inactive}(-)$ in the premise of O-Remove. Second, σ_γ still unions the tables of Active fibers alone, so a fiber whose transition is under way in either direction reads its coeffects through the ω it holds and provides none of its own; a key that its transition has already written is therefore not yet one a dependent may activate against. In the two-state calculus the distinction is empty, every installed fiber being Active there.

Figure 2 draws the lifecycle these states form, and the four subsections below supply the rules on its edges.

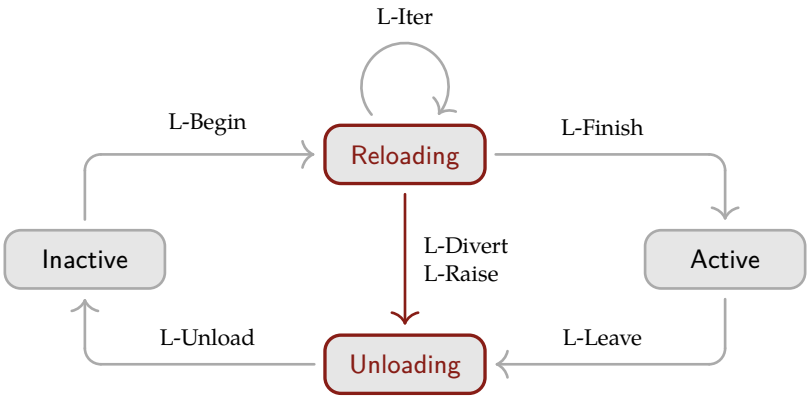


Figure 2 | Lifecycle with transitions in progress; the two transition states are outlined

4.3.1. Withdrawal

Section 3.2 requires that dependents activate after their dependencies and that dependencies withdraw their provisions only after their dependents have deactivated. The first half holds in the base calculus already: an activation requires $\gamma \vDash d_n$, so a fiber declaring k cannot activate before some fiber is actively providing k. The second half is the substantive one, and it must deliver more than an ordering of state changes. A component being torn down because its provider is going away is running its own teardown code, which may need the very coeffect that is being withdrawn; closing a connection pool typically means handing the connections back to whatever provided them. What the second half must deliver is that a consumer can still read k throughout its own deactivation, and that the provider's withdrawal of k takes effect only afterwards. The base calculus cannot deliver it at all: its L-Unload removes the provisions and runs the inverse together, leaving no interval between them for a consumer's teardown to occupy.

This layer splits that step in two, and guards the second half by the following condition.

Definition 50. The fiber n is *relied upon* at γ when some other installed fiber resolves a key to it:

$$\mathrm{relied}_n(\gamma) := \exists m \in \mathrm{dom}\big(F_\gamma\big),\, k \in d_m.\; m \neq n \wedge \mathrm{installed}_m(\gamma) \wedge \omega_m(k) = n \tag{46}$$

$$\frac{\theta_n = \mathsf{Active}(g,\omega) \quad \mathrm{target}_n(\gamma) \neq \omega}{\gamma \longrightarrow \gamma[\theta_n \mapsto \mathsf{Unloading}(g,\omega,\bot)]}\ \text{L-Leave}$$

$$\frac{\theta_n = \mathsf{Unloading}(g,\omega,\zeta) \quad \neg\,\mathrm{relied}_n(\gamma) \quad g(\gamma) = \delta}{\gamma \longrightarrow \delta[\theta_n \mapsto \mathsf{Inactive}(\zeta)]}\ \text{L-Unload}$$

L-Leave records the decision to deactivate without acting on it, which stops the fiber providing its coeffects while leaving its own committed view and everyone else's intact. L-Unload applies the accumulator, discards the committed view, and leaves the fiber Inactive on the outcome it carries; the outcome is $\bot$ until Section 4.3.4 supplies the other case. It is the only rule in the calculus that applies an accumulator.

The two halves of the ordering are then carried by different parts of the form: the visibility half by the committed view, which L-Unload discards as its last act, and the ordering half by the premise $\neg\,\mathrm{relied}_n(\gamma)$, which we call the *guard* and which holds the withdrawal of k back until every consumer that resolves it to n has gone. Theorem 63 establishes both.

The guard is imposed per binding rather than per fiber: $\mathrm{relied}_n(\gamma)$ tests whether some committed view names n, so a fiber that declares none of n's keys is no obstacle, and neither is one that resolved a key of n's in another realm (Section 3.2.3). Under the single-source discipline of Section 4.2 the per-binding reading coincides with the coarser test $\exists m \neq n,\, k \in d_m.\ \mathrm{installed}_m(\gamma) \wedge k \in p_n$, a key having one possible provider there.

A guard of this kind ordinarily deadlocks. What keeps it from doing so is Unloading together with σ_γ being the union over Active fibers alone: once L-Leave has marked n, its table leaves σ_γ, so no target view can name n any longer, and every consumer that committed to n is itself on its way out. Theorem 66 turns that into the claim that the guard always releases.

The guard orders deactivations along coeffects and not along the fiber tree: a parent may run its inverse while a child of it is still Unloading, since relied speaks only of committed views. Parent and child are accordingly ordered more weakly than Theorem 63 orders a provider and its consumer, and a parent and a child whose effects meet in the ambient state are governed by the independence hypothesis of Definition 60 instead.

4.3.2. Iteration

An activation may execute multiple effects in sequence, and the deactivation must recover them. We model such an activation with an *effect iterator*, each of whose iterations yields the modified context, an inverse, and a continuation:

Definition 51. Define the effect iterator $\mathfrak{E}^{\mathrm{iter}}_\Gamma$ and witnessed effect iterator $\mathfrak{E}^{\mathrm{iter}*}_\Gamma$ as the following recursive types:

$$\begin{aligned} \mathfrak{E}^{\mathrm{iter}}_\Gamma &:= \mu\mathfrak{I}.\ \Gamma \to \Gamma \times (\Gamma \to \Gamma) \times \mathsf{Maybe}(\mathfrak{I}) \\ \mathfrak{E}^{\mathrm{iter}*}_\Gamma &:= \mu\mathfrak{I}.\ (e : \Gamma \to \Gamma \times (\Gamma \to \Gamma) \times \mathsf{Maybe}(\mathfrak{I})) \\ &\qquad \times ((\gamma : \Gamma) \to (\mathbf{let}\ (\delta, g, o) = e(\gamma)\ \mathbf{in}\ g(\delta) \simeq \gamma)) \end{aligned} \tag{47}$$

where $e(\gamma)$ yields a triple (δ, g, o) representing:

- δ is the new context;
- g is the inverse function of the current effect;
- o indicates the continuation:
  - Nothing signals iteration termination;
  - $\mathsf{Just}(i)$ provides the next iteration.

The witness is read at the $\simeq$ of Definition 33, as Definition 37 reads that of $\mathfrak{E}^*_\Gamma$: an $i \in \mathfrak{E}^{\mathrm{iter}}_\Gamma$ lies in $\mathfrak{E}^{\mathrm{iter}*}_\Gamma$ when i respects $\simeq$ and each g it yields respects $\simeq$ and satisfies the clause above. A triple is compared componentwise, Nothing with Nothing alone and $\mathsf{Just}(i)$ with $\mathsf{Just}(i')$ when $i \simeq i'$, and $\simeq$ on iterators is the greatest relation meeting those clauses. Taking $\simeq$ to be equality on Γ recovers the reading on the nose.

The effect iterator transformation $\mathrm{effect}^{\mathrm{iter}}_\Gamma$ extends effect_Γ to the iterator structure through recursive invocation:

Definition 52. Define the effect iterator transformation $\mathrm{effect}^{\mathrm{iter}}_\Gamma$ as:

$$
\begin{aligned}
&\mathrm{effect}^{\mathrm{iter}}_{\Gamma} \;:\; \mathfrak{E}^{\mathrm{iter}}_{\Gamma} \;\to\; \partial\Gamma \;\to\; \partial^2\Gamma \\
&\mathrm{effect}^{\mathrm{iter}}_{\Gamma} \;=\; i \;\mapsto\; (\gamma, \varphi) \;\mapsto\; \begin{array}{l}
\mathbf{let}\ (\delta, g, o) = i(\gamma)\ \mathbf{in} \\
\mathbf{let}\ t = \mathrm{track}_{\Gamma}(g, \mathrm{pr}_1 \circ i)\ \mathbf{in} \\
\mathbf{match}\ o \\
\quad \mid \mathsf{Nothing} \Rightarrow ((\delta, \varphi \circ g), t) \\
\quad \mid \mathsf{Just}(i') \Rightarrow \mathbf{let}\ (s, r) = \mathrm{effect}^{\mathrm{iter}}_{\Gamma}(i')(\delta, \varphi \circ g)\ \mathbf{in} \\
\qquad\qquad (s, t \circ r)
\end{array}
\end{aligned}
\tag{48}
$$

At each iteration, the inverse g is composed onto φ in application order, so the accumulator $\varphi \circ g_1 \circ \cdots \circ g_k$ naturally recovers effects in LIFO order when applied. Because $\mathrm{effect}^{\mathrm{iter}}_{\Gamma}$ lands in the same $\partial\Gamma \to \partial^2\Gamma$ as effect_{Γ} does, an iterator is an effect in its own right and can be used wherever an effect can. A component's whole activation is one such use, which is what the rest of this section formalizes, and the implementation admits an iterator at every mutation site (Section 5.1.1). The $\mathsf{Maybe}(\mathfrak{E}^{\mathrm{iter}})$ continuation makes a boundary available between any two consecutive iterations, at which the context is whatever the iterations so far have made it and the accumulator recovers those and nothing more. In this sense the effect iterator is a reified delimited continuation, the structure that mainstream languages expose through the `yield` operator [43], so the model maps directly onto the generators they already provide.

In the calculus, the e_n of Definition 44 is read at $\mathfrak{E}^{\mathrm{iter}*}_{\Gamma}$ from here on, and replacing the atomic effect function by an iterator splits the base L-Reload into a begun state that the trace passes through, and gives the fiber a second way out of that state.

$$
\frac{\theta_n = \mathsf{Inactive}(\bot) \quad \omega = \mathrm{target}_n(\gamma) \neq \bot}{\gamma \longrightarrow \gamma[\theta_n \mapsto \mathsf{Reloading}(e_n, \mathrm{id}_{\Gamma}, \omega)]}\ \text{L-Begin}
$$

$$
\frac{\theta_n = \mathsf{Reloading}(i, g, \omega) \quad \mathrm{target}_n(\gamma) \neq \omega \quad (\delta, h) = (\gamma, \mathrm{id}_{\Gamma}) \vee i(\gamma) = (\delta, h, -)}{\gamma \longrightarrow \delta[\theta_n \mapsto \mathsf{Unloading}(g \circ h, \omega, \bot)]}\ \text{L-Divert}
$$

$$
\frac{\theta_n = \mathsf{Reloading}(i, g, \omega) \quad \mathrm{target}_n(\gamma) = \omega \quad i(\gamma) = (\delta, h, \mathsf{Just}(i'))}{\gamma \longrightarrow \delta[\theta_n \mapsto \mathsf{Reloading}(i', g \circ h, \omega)]}\ \text{L-Iter}
$$

$$
\frac{\theta_n = \mathsf{Reloading}(i, g, \omega) \quad \mathrm{target}_n(\gamma) = \omega \quad i(\gamma) = (\delta, h, \mathsf{Nothing})}{\gamma \longrightarrow \delta[\theta_n \mapsto \mathsf{Active}(g \circ h, \omega)]}\ \text{L-Finish}
$$

Each iteration composes the newly yielded inverse onto the accumulator as $g \circ h$, following Definition 52, so that the accumulator applies the inverses in last-in-first-out order. Between any two consecutive iterations the system may *divert* the transition if its target view has changed, applying the inverse accumulated so far to recover the context. L-Divert routes through Unloading like every other deactivation rather than applying the accumulator where it stands, and the guard it meets there is vacuous, a fiber that has never been Active providing nothing and appearing in no committed view. The first of its two alternatives *aborts* the iteration the fiber is holding, which only an iteration boundary makes possible, so the granularity at which a divert may fall is that of the iterator; the second lets that iteration land, and Section 4.3.3 is where it is needed.

A plain effect function ($\mathfrak{E}_{\Gamma}$) is the degenerate case where the first iteration already yields Nothing. Such a transition still passes through Reloading and L-Divert still applies there, but the accumulator is id_{Γ} and no iteration has run, so nothing is restored and the transition installs either all of its effects or none of them.

4.3.3. Asynchrony

The layers so far let the environment move between one iteration and the next, and assume that each iteration itself completes instantaneously, its launch and its landing being one step. We model non-immediacy abstractly: an iteration yields a value of type $\mathsf{Future}(A)$, where Future is an opaque type constructor whose defining property is that between submission and resolution, external state may change.

Under this model an iteration is launched at one state and lands at another, and the fiber is Reloading while it is in flight. What the layer adds is *inertia*: once launched, an iteration lands, and its landing cannot be declined. A target view that turns during the flight therefore cannot be answered by aborting the iteration, and only the alternative of L-Divert that lands one remains available: the iteration lands, and the fiber deactivates afterwards. This layer therefore adds no rule and no type that a rule matches on; at the granularity of Γ inertia is its whole content, and it takes the form of a restriction on which alternative of L-Divert a host may take.

That alternative is what the base calculus could not express. There, a transition whose target view had turned was undone in the same step that discovered it; here the iteration in flight must land first, so the fiber needs somewhere to be while its inverse runs, and the only sound place is Unloading holding the inverse the iteration produced. Routing through Active instead would let the fiber provide its coeffects for the length of one step and oblige its dependents to activate against a component that is already leaving. This is the mutual chaining of `reload` and `unload` in the implementation.

A deactivation may also chain straight back into an activation, by a composite rather than a rule. L-Unload carries no premise on the target view, so whatever the target view has become while the fiber was deactivating, the accumulator runs and the fiber becomes Inactive, from which L-Begin may immediately start a new transition.

4.3.4. Failure

Every rule so far assumes the effect it runs succeeds, and a runtime cannot. The effects a component installs reach outside the context that tracks them, and what they reach may refuse: a port already bound, a file that is not there, a peer that does not answer. A failing transition must still leave the fiber's effects recovered rather than stranded.

Let Ξ be a set of errors and refine the effect iterator of Definition 51 so that an iteration may raise in place of yielding a triple:

$$\begin{aligned} \mathfrak{E}_{\Gamma}^{\mathrm{fail}} &:= \mu\mathfrak{I}.\ \Gamma \to \mathsf{Either}(\Xi,\ \Gamma \times (\Gamma \to \Gamma) \times \mathsf{Maybe}(\mathfrak{I})) \\ \mathfrak{E}_{\Gamma}^{\mathrm{fail}*} &:= \mu\mathfrak{I}.\ (e : \Gamma \to \mathsf{Either}(\Xi,\ \Gamma \times (\Gamma \to \Gamma) \times \mathsf{Maybe}(\mathfrak{I}))) \\ &\qquad \times ((\gamma : \Gamma) \to (\mathbf{let}\ \mathsf{Right}(\delta, g, o) = e(\gamma)\ \mathbf{in}\ g(\delta) \simeq \gamma)) \end{aligned} \tag{49}$$

The witness constrains the Right case alone, being vacuous where the pattern does not match, a raise having nothing to undo, and the i that Reloading carries is read at $\mathfrak{E}_{\Gamma}^{\mathrm{fail}*}$ from here on. The lift of Definition 52 carries over with a raise propagated in place of a triple, so a raising iterator is usable wherever an effect is, as an ordinary one is. The layer adds one rule and puts the second outcome of Definition 49 to use, O-Remove needing no widening to admit it. The premises of L-Iter, L-Finish, and L-Divert are read with Right around the triple they match. A raise is something an iteration does, so the rule is an exit from Reloading.

$$\frac{\theta_n = \mathsf{Reloading}(i, g, \omega) \quad i(\gamma) = \mathsf{Left}(\xi)}{\gamma \longrightarrow \gamma[\theta_n \mapsto \mathsf{Unloading}(g, \omega, \xi)]}\ \text{L-Raise}$$

L-Raise recovers before it records. The fiber routes into Unloading carrying the error as its outcome, the accumulator built up to the failing iteration is applied there, and the fiber arrives at $\mathsf{Inactive}(\xi)$ having installed nothing, at a state differing from the one an aborting L-Divert would have produced only in the outcome the fiber carries. Routing a failure like every other deactivation is what makes every outcome reachable only through L-Unload, which is the single fact Theorem 59 turns on. L-Begin has $\mathsf{Inactive}(\bot)$ as a premise, so the lifecycle is not re-entered from an error outcome; this is the substance of the outcome, which withholds a fiber whose effect function has shown itself to be unsound in the state it ran against rather than retrying it against an unchanged environment. A failed fiber also obstructs nothing: it is Inactive, so it carries no committed view and cannot make relied hold.

A failure is recorded on the fiber rather than propagated to its parent, so a component whose transition fails leaves its siblings running, which is the behavior a plugin host wants and the reason the outcome is per-fiber rather than a property of the whole state.

4.4. Metatheory

Section 4.3 supplies ten rules: the three orchestration rules of Section 4.2; L-Begin, L-Iter, and L-Finish for an activation; L-Divert and L-Raise for the two ways an activation may end early; and L-Leave and L-Unload for a deactivation. This section reads the two dimensions of composability off those rules in their global form, one fiber's guarantee holding whatever the other fibers do in between, and adds what only a whole system can be asked for: that it always reaches the configuration its targets call for, and that the configuration is the one a static assembly would have produced. Every property below is a property of a sequence of steps, so we index the steps and read the fields of a state off that index.

Two conventions carry Section 3.3.2 into this section. Every equality between states below is read up to the observational equivalence $\simeq$ of Definition 33, as Lemma 38 reads those of Section 3.1, and the witness condition an effect function is held to is the one Definition 37 gives, read of an iterator as Definition 51 gives it and of a registering iteration at the $\approx$ below.

Definition 53. Index the steps by t, so that γ^t is the state the first t of them reach, and write

$$\mathrm{step}^t := r(n) \tag{50}$$

for the step taken at γ^t: the rule r it applies, one of the ten, and the name $n \in \mathfrak{N}$ it applies that rule at. The sequence starts at a γ^0 with $\mathrm{dom}(F^0) = \varnothing$, so every fiber comes into existence by an O-Insert, whether the orchestrator's or one an iteration takes (Definition 47). A field of γ^t carries the index as a superscript, so that θ_n^t, ω_n^t, σ_n^t, g_n^t, and i_n^t are the lifecycle state, committed view, table, accumulator, and remaining iterator of n at γ^t, and F^t and σ^t the registry and coeffect context of γ^t itself, the F_γ and σ_γ of Definition 45 read there. Predicates take the state as their argument and everything else as a subscript, so $\mathrm{installed}_n^t$, target_n^t, relied_n^t, and quiet^t are the predicates of Definition 46, Definition 49, and Definition 50 at γ^t. An *episode* of n is a maximal interval $[b, u]$ of indices throughout which $\mathrm{installed}_n^t$ holds. It *opens* at b, where $b > 0$ and $\neg\,\mathrm{installed}_n^{b-1}$, the empty F^0 leaving no fiber installed at the outset; it *closes* at u when $\mathrm{installed}_n^u$ and not $\mathrm{installed}_n^{u+1}$, which a final episode need not do.

Every rule of Section 4.3 concludes in the shape $\gamma \longrightarrow \delta[\cdots]$, where the premises compute δ from γ and leave it as γ where they compute nothing, and the bracket edits named fields of the

registry. The two halves are named separately, and both are maps on all of Γ. The *state map* of a step taken at γ^t by a rule acting on n is

$$\Psi^t := \begin{cases} \mathrm{pr}_1 \circ i & \text{at L-Iter, L-Finish, and a landing L-Divert} \\ g & \text{at L-Unload} \\ \mathrm{id}_\Gamma & \text{at every other rule} \end{cases} \tag{51}$$

where i and g are the iterator and the accumulator that θ_n^t carries, and the *edit* $\mathrm{edit}^t : \Gamma \to \Gamma$ is the bracket read as a function, assigning to the fields it names the values the premises computed at γ^t. Both are therefore fixed by step^t together with γ^t and defined at every state, which is what lets Theorem 61 and Lemma 71 evaluate them away from γ^t. Each step factors as

$$\gamma^{t+1} = \mathrm{edit}^t(\Psi^t(\gamma^t)) \tag{52}$$

At L-Unload, for instance, edit^t is $[\theta_n \mapsto \mathsf{Inactive}(\zeta)]$, and at O-Remove it is the removal $\setminus n$, which is why the second half is an edit rather than an assignment. The fields divide along the same seam: the *tables* σ_m, which no edit^t writes once the O-Insert creating m has set it empty, and the *control fields* θ_m, τ_m, π_m, d_m, p_m, e_m together with $\mathrm{dom}(F_\gamma)$, which no Ψ^t writes save through the primitive of Definition 47. Write $\gamma \approx \delta$ when two states agree on everything but the control fields.

The relation $\approx$ is not the $\simeq$ of Definition 33, and neither refines the other, because each forgets what the other has to keep. Recovery exactness is a claim about effects, so $\approx$ compares the tables and the ambient state exactly and forgets only the registry's record of which fiber installed them. A rule reads the control fields to decide whether it applies, so $\simeq$ has to keep them, and this section reads it as the conjunction of Definition 33 with agreement on the registry's domain and on every control field of every fiber:

$$\gamma \simeq \delta \quad := \quad \sigma_\gamma \simeq \sigma_\delta \wedge \mathrm{dom}(F_\gamma) = \mathrm{dom}(F_\delta) \wedge \forall n,\, c \in \{\theta, \tau, \pi, d, p, e\}.\; c(\gamma(n)) \simeq c(\delta(n)) \tag{53}$$

A field of function type, as e_n and the g inside θ_n are, is compared as Definition 36 compares maps, an iterator as Definition 51 compares two, and a field of any other type by equality. The results below hold up to both relations, one for each half of the state, Lemma 55 establishing the $\simeq$ half once for all ten rules.

Table 1 is the ten rules of Section 4.3 read as such writes. The accumulator, the committed view, and the remaining iterator are constituents of θ_n, so the third column records the writes to them as well, and h there names the inverse the iteration of the fourth column yields, id_Γ where L-Divert aborts that iteration. Where a Ψ^t built from an iterator registers a fiber (Definition 47), that registration carries the writes of the O-Insert row at the name it draws, and an L-Unload whose accumulator retires one carries those of the O-Retire row. Every case analysis below is a lookup in the table, and five lookups recur often enough to name.

rule	θ_n^t	θ_n^{t+1}	Ψ^t	control fields edited
O-Insert	undefined	Inactive$(\bot)$	id_Γ	$\mathrm{dom}(F_\gamma)$
O-Retire	unconstrained	unchanged	id_Γ	τ_n
O-Remove	Inactive$(-)$	undefined	id_Γ	$\mathrm{dom}(F_\gamma)$
L-Begin	Inactive$(\bot)$	Reloading$(e_n, \mathrm{id}_\Gamma, \omega)$	id_Γ	θ_n
L-Iter	Reloading(i, g, ω)	Reloading$(i', g \circ h, \omega)$	$\mathrm{pr}_1 \circ i$	θ_n
L-Finish	Reloading(i, g, ω)	Active$(g \circ h, \omega)$	$\mathrm{pr}_1 \circ i$	θ_n
L-Divert	Reloading(i, g, ω)	Unloading$(g \circ h, \omega, \bot)$	id_Γ or $\mathrm{pr}_1 \circ i$	θ_n
L-Raise	Reloading(i, g, ω)	Unloading(g, ω, ξ)	id_Γ	θ_n
L-Leave	Active(g, ω)	Unloading$(g, \omega, \bot)$	id_Γ	θ_n
L-Unload	Unloading(g, ω, ζ)	Inactive(ζ)	g	θ_n

Table 1 | The rules as writes on the fiber n they act on, where step^t is that rule applied at n.

Lemma 54. Reading Table 1 together with Definition 48, for every step t and all fibers m, n present at γ^t:

1. $\sigma_m^{t+1} \neq \sigma_m^t$ only where step t acts on m, the write lying inside Ψ^t;
2. ω_n comes into existence only where $\mathrm{step}^t = \text{L-Begin}(n)$ and ceases only where $\mathrm{step}^t = \text{L-Unload}(n)$, so ω_n^t is constant for t in an episode of n;
3. $\Psi^t = g_n^t$ only where $\mathrm{step}^t = \text{L-Unload}(n)$, and no other step applies g_n to the state;
4. $\neg\,\mathrm{installed}_n^t \wedge \mathrm{installed}_n^{t+1} \Rightarrow \mathrm{step}^t = \text{L-Begin}(n)$, and $\mathrm{installed}_n^t \wedge \neg\,\mathrm{installed}_n^{t+1} \Rightarrow \mathrm{step}^t = \text{L-Unload}(n)$;
5. π_n, d_n, p_n, and e_n come into existence with the entry of n and are never written again, and τ_n is monotone, written only at $\top$ and only by an O-Retire.

Proof. Let step t apply r at n. By Definition 53 it factors as $\mathrm{edit}^t \circ \Psi^t$, where edit^t writes the fields the fifth column of Table 1 names and nothing else, and Ψ^t is id_Γ, an application of one of n's iterations, or the accumulator g_n^t, which is a composite of the inverses those iterations yielded. Each of the three is confined to n by Definition 48, so Ψ^t writes no field of a fiber present at γ^t but σ_n, together with the entry a registration adds and the τ its inverse writes. The two halves therefore partition the writes, and each clause is that partition read at one field. One reading of the second and third columns is used twice: Inactive is the one lifecycle state carrying no committed view, L-Begin the one rule leading out of it, and L-Unload the one rule leading into it, while every other row carries the ω of its premise into its conclusion unchanged.

(1) An edit^t writes no table, the fifth column naming none, and a Ψ^t writes no σ_m for a present $m \neq n$. So σ_m can move only at $m = n$, and only inside Ψ^t.

(2) ω_n is a constituent of θ_n, which only an edit^t writes and only at the fiber the step acts on, so by the reading above ω_n comes into existence at an L-Begin of n and ceases at an L-Unload of n. An episode of n is an interval on which $\mathrm{installed}_n$ holds, hence one throughout which ω_n is defined, so neither rule falls in its interior.

(3) The fourth column, where an accumulator appears at L-Unload alone: the other rules take a forward map $\mathrm{pr}_1 \circ i$ or id_Γ, and no edit^t applies a map to the state at all.

(4) $\mathrm{installed}_n$ is $\theta_n \neq \text{Inactive}(-)$, and by the reading above L-Begin and L-Unload are the only rules whose premise and conclusion differ in whether θ_n is Inactive. A step acting on some $m \neq n$ writes no θ_n, and the entry a registration adds is at a name not present at γ^t.

(5) No row of the fifth column names a π, d, p, or e; those come into existence with the entry O-Insert adds, which its conclusion writes, as does the O-Insert a registration takes. Only O-Retire writes a τ, at $\top$, whether taken by the orchestrator or as the inverse of a registration (Definition 47); O-Insert sets $\tau = \bot$ at a name not already present, so no step returns a τ to $\bot$.□

Three further lookups say what the rules cannot see. The first is that they read the state only through the observations above, so that the whole calculus descends to $\Gamma/\simeq$.

Lemma 55. ($\simeq$-invariance.) Let $\gamma \simeq \gamma'$ as read above. Then a rule of Section 4.3 applies at γ acting on n if and only if it applies at γ' acting on n, and the states the two applications reach are again related by $\simeq$.

Proof. Every premise of Section 4.3 is of one of four kinds, and each reads a constituent the relation keeps. A premise matching θ_n or τ_n against a pattern, and the premise $\forall m.\ \pi_m \neq n$ of O-Remove, read control fields. The premises $(d, p, e) \in \mathfrak{C}_\Gamma$ and $\forall m.\ p \cap p_m = \varnothing$ of O-Insert read d, p, and e. A premise mentioning target_n or relied_n reads τ_n, the committed views inside the θ_m, and $\mathrm{dom}(\sigma_\gamma)$, which Definition 45 computes from the θ_m and the $\mathrm{dom}(\sigma_m)$, and Definition 33 relates two coeffect contexts only where their domains agree. The remaining premises read $\mathrm{dom}(F_\gamma)$. None reads a value $\sigma_\gamma(k)$ otherwise than up to $\underset{k}{\simeq}$, so no premise separates two $\simeq$-related states.

For the conclusion, $\gamma^{t+1} = \mathrm{edit}^t(\Psi^t(\gamma^t))$ by Definition 53. The values an edit^t assigns are the constituents of the premises it matched, related at the two states by the paragraph above and by Definition 51, which relates the triples an iterator yields at $\simeq$-related states. And Ψ^t respects $\simeq$: it is id_Γ, or an iteration of e_n, which Definition 51 requires to respect $\simeq$, or the accumulator inside θ_n, a composite of inverses each respecting $\simeq$ by the same definition. □

The names a state carries are read by two of those observations, $\mathrm{dom}(F_\gamma)$ and the indexing of the control fields, and the rule that draws a name draws any name not already in use (Definition 47). Reading the results below up to $\simeq$ therefore also calls for reading them up to a renaming, which is the discipline of Section 4.1 cashed out.

Lemma 56. (Equivariance.) Let $\chi : \mathfrak{N} \to \mathfrak{N}$ be a bijection and let $\chi \cdot \gamma$ be the state carrying the registry $F_\gamma \circ \chi^{-1}$, with every name occurring in a π_m or an ω_m replaced by its image. Then $\chi \cdot \gamma$ is a state, well formed where γ is, and $\mathrm{step}^t = r(n)$ carries γ^t to γ^{t+1} if and only if $r(\chi(n))$ carries $\chi \cdot \gamma^t$ to $\chi \cdot \gamma^{t+1}$.

Proof. A premise reads a name only by comparing it with another, whether directly, as in the freshness $n \notin \mathrm{dom}(F_\gamma)$ of O-Insert and the $\forall m.\ \pi_m \neq n$ of O-Remove, or through a table of names, as target_n and relied_n read the π_m and the ω_m. A bijection preserves each such comparison. The only names a rule writes are the π that O-Insert sets and the ω that L-Begin sets, both taken from what its premises read, so the writes commute with χ; an effect function writes no name at all, drawing one only through the primitive of Definition 47, which Definition 48 confines to the entry that primitive adds. Well-formedness (Definition 58) is four conditions comparing names with names. □

A sequence and its renaming therefore take the same rules in the same order and reach states differing by χ alone. Two sequences agreeing save in the names their registrations draw are accordingly identified, and the results below are read up to the renaming that identifies them.

The second lookup is that an entry stripped of everything but its name is invisible to the rules, which is what lets Definition 47 retire a fiber where the state it recovers has none, and Lemma 72 remove the registrations a deleted episode made.

Lemma 57. (Vestigial entries.) Call n *vestigial* at γ when $\tau_n = \top$, $\theta_n = \mathsf{Inactive}(\bot)$, $\sigma_n = \varnothing$, and no m has $\pi_m = n$; a vestigial entry satisfies $\gamma \approx \gamma \setminus n$. If n is vestigial at γ then for every rule and every $m \neq n$:

1. a rule applying at γ acting on m applies at $\gamma \setminus n$ acting on m, and the states the two reach differ in the entry at n alone, which stays vestigial;
2. conversely a rule applying at $\gamma \setminus n$ acting on m applies at γ, unless it is an O-Insert drawing the name n or claiming a key of p_n.

Proof. A vestigial n contributes to no observation a premise of a rule acting on $m \neq n$ reads. It is not Active, so σ_n enters no σ_γ and n is the provider of no key, leaving $\gamma \vDash d_m$ and target_m unmoved; $\mathrm{installed}_n$ fails, so n contributes no disjunct to relied_m; no $\pi_{m'}$ names n, so the premise $\forall m'.\ \pi_{m'} \neq m$ of an O-Remove of m is unmoved; and θ_n, τ_n, and π_n are read by rules acting on n alone. The two premises clause (2) excepts are the two the removal relaxes, an absent name being fresh and an absent provision meeting every other. By Lemma 54 no rule acting on $m \neq n$ writes a field of n, so the entry survives, and the state map of the step is confined to m by Definition 48, so it leaves σ_n empty. □

Simplifying the lifecycle states, together with the rules that match on them, yields a sub-calculus, and not every result survives the simplification. Dropping Section 4.3.1 is the case that matters, which is the division Section 4.3 opens with, read from the metatheory's side: its guard is what establishes clauses (3) and (4) of Definition 58, and Theorem 63 rests on the interval the guard creates, so those three fail without it. What the other three subsections add can be simplified away without disturbing the results below, each of them only adding rules to the one state space Definition 49 fixes.

4.4.1. Preservation

Definition 45 fixes the shape of a registry, and the rules have to be checked against it before the results below can add to it. This subsection identifies the invariant the rules preserve, of which the first clause is that shape and the rest what those results assume.

Definition 58. A registry F_γ is *well formed* when, for all $m, n \in \mathrm{dom}(F_\gamma)$ and all $k \in K$,

1. $\pi_n \in \mathrm{dom}(F_\gamma) \cup \{\mathsf{root}\}$;
2. $m \neq n \Rightarrow p_m \cap p_n = \varnothing$;
3. $\mathrm{installed}_n(\gamma) \Rightarrow \omega_n$ is total on d_n and valued in $\mathrm{dom}(F_\gamma)$;
4. $\mathrm{installed}_n(\gamma) \wedge k \in d_n \wedge \omega_n(k) = m \Rightarrow \mathrm{installed}_m(\gamma)$.

Clause (1) is the tree of Definition 45 read one edge at a time, keeping a parent pointer landing in the registry. The acyclicity that definition also requires needs no clause, since the fiber a pointer names is registered before the fiber naming it.

Theorem 59. (Preservation.) If F^t is well formed then so is F^{t+1}, whichever rule step t applies. Each clause is established at γ^{t+1} from all four at γ^t.

Proof. Let step t act on n.

(1) By Table 1 only O-Insert and O-Remove write a π or $\mathrm{dom}(F_\gamma)$. O-Insert has $\pi_n \in \mathrm{dom}(F^t) \cup \{\mathsf{root}\}$ as a premise, which is the clause for the fiber it adds, and it leaves every other π alone

while enlarging $\mathrm{dom}(F_\gamma)$. O-Remove has $\forall m.\ \pi_m \neq n$, so no surviving π_m names the fiber it takes away.

(2) The last premise of O-Insert is $\forall m.\ p_n \cap p_m = \varnothing$, which is the clause for the fiber it adds, and by Table 1 no other rule writes a p or enlarges $\mathrm{dom}(F_\gamma)$. Two consequences are used below: $\mathrm{dom}(\sigma_m) \subseteq p_m$ by Definition 43, so distinct tables are disjoint and σ_γ is a function; and $k \in p_m \cap p_{m'}$ forces $m = m'$, so k has at most one possible provider.

(3) By Lemma 54(2) the only rule that writes an ω_n is L-Begin, whose premise $\omega = \mathrm{target}^t_n \neq \bot$ makes it total on d_n and valued in $\mathrm{dom}(F^t)$, target naming providers. By Table 1 the only rule that shrinks $\mathrm{dom}(F_\gamma)$ is O-Remove, whose premise $\theta^t_n = \mathsf{Inactive}(-)$ gives $\neg\,\mathrm{installed}^t_n$, whence by clause (4) at γ^t no m has $\omega^t_m(k) = n$ for a $k \in d_m$ while $\mathrm{installed}^t_m$; and n itself carries no ω.

(4) By Lemma 54(2) and (4) the clause can fail at γ^{t+1} only where some installed has fallen, some ω has been written, or a fiber some ω names has left $\mathrm{dom}(F_\gamma)$. The last is an O-Remove, whose removed fiber is not installed and hence, by clause (4) at γ^t, is named by no ω^t_m of an installed m. The first is an L-Unload of n, whose premise $\neg\,\mathrm{relied}^t_n$ reads

$$\forall m \neq n,\, k \in d_m.\ \mathrm{installed}^t_m \Rightarrow \omega^t_m(k) \neq n$$

and which writes no ω_m for $m \neq n$ and leaves $\neg\,\mathrm{installed}^{t+1}_n$, so the clause holds of n as well. The second is an L-Begin of n, writing target^t_n, whose values are the providers of the keys of d_n and hence Active at γ^t; the step alters no other fiber's θ, so they are installed at γ^{t+1} too. □

The guard on L-Unload is what carries clauses (3) and (4). The premise $\forall m.\ \pi_m \neq n$ of O-Remove speaks only of parent pointers; what keeps a committed view from naming a removed fiber is the guard, imposed several steps earlier and for a different reason. Because a failure is routed through Unloading as well, the argument does not have to be repeated for an error outcome. Two things follow that the base calculus does not enjoy. A name freed by O-Remove may be reissued by O-Insert, since no stale committed view can name it; and a fiber may be removed as soon as it is Inactive, without a separate check that nobody depends on it.

4.4.2. Temporal Composability

Local temporal composability recovers one sequence of effects with one accumulator (Section 3.1.3). The registry holds one accumulator per fiber and the fibers interleave: between the moment n composes an inverse onto g_n and the moment g_n runs, other fibers have moved the state. Whether g_n still undoes what it was built to undo there is what the global form of the guarantee asserts, and the condition it turns on is that the intervening steps commute with g_n.

Definition 60. For $i \in \mathfrak{E}^{\mathrm{iter}*}_\Gamma$ let $\mathrm{reach}(i)$ be the least set of iterators containing i and closed under continuation, and read the transformation monoid $\mathfrak{M}$ of Definition 17 at an iterator by taking for its generators the forward maps and the yielded inverses of every iterator in $\mathrm{reach}(i)$:

$$\begin{aligned} \mathrm{reach}(i) &:= \bigcap \{S \mid i \in S \wedge \forall i' \in S, \gamma \in \Gamma.\ i'(\gamma) = (-,-,\mathsf{Just}(i'')) \Rightarrow i'' \in S\} \\ \mathfrak{M}(i) &:= \langle \{\mathrm{pr}_1 \circ i' \mid i' \in \mathrm{reach}(i)\} \cup \{\mathrm{pr}_2(i'(\gamma)) \mid i' \in \mathrm{reach}(i), \gamma \in \Gamma\} \rangle \end{aligned} \tag{54}$$

reading Right around the triple where Section 4.3.4 applies, and write $\mathrm{len}(i)$ for the supremum of $|C|$ over the chains $C \subseteq \mathrm{reach}(i)$ that continuation orders. Two iterators i, j are *independent* when they are so in the sense of Definition 19, read with these transformation monoids and with the yield of an iteration being its inverse together with its continuation:

$$\begin{aligned} \forall f \in \mathfrak{M}(i),\, g \in \mathfrak{M}(j). \quad & f \circ g \simeq g \circ f \\ \forall i' \in \mathrm{reach}(i),\, g \in \mathfrak{M}(j),\, \gamma \in \Gamma. \quad & \mathrm{pr}_{2,3}(i'(g(\gamma))) \simeq \mathrm{pr}_{2,3}(i'(\gamma)) \end{aligned} \tag{55}$$

and symmetrically in j, reading $\simeq$ on maps as Definition 36 does, on a continuation as Definition 51 does, and on a registering iteration (Definition 47) as agreement of the component it names. A family $(i_l)_{l \in L}$ of iterators is *pairwise independent* when i_l and $i_{l'}$ are independent for every $l \neq l'$, and a sequence of steps is pairwise independent when $(e_n)_{n \in N}$ is, where N is the set of names the sequence ever holds, one for each fiber the orchestrator inserts and each fiber an iteration registers.

Independence in this sense is what trace theory takes as primitive: commuting actions generate an equivalence on sequences under which reordering two adjacent independent actions preserves the endpoint [44], and Lemma 71 is that reordering for these rules. A family rather than a set is what keeps two names of one component in scope: the condition then requires that component's effect function to be independent of itself, which is to require that $\mathfrak{M}(i)$ be commutative. The first condition is what Theorem 61 uses and the second what Theorem 73 needs in addition: reordering the steps of two fibers evaluates an iterator at a state the other fiber moved, and commuting the maps does not by itself say that the iterator yields the same inverse and the same continuation there. Checking the first condition calls for no more than the iterations themselves, since Lemma 18(1) carries commutation from the generators to the monoids they generate.

Under these conditions the single-accumulator invariant of Theorem 7 survives the interleaving, in the form that gives temporal composability its content: running an inverse withdraws the fiber's contribution and nothing else.

Theorem 61. (**Recovery exactness.**) Let the sequence of steps be pairwise independent, let an episode of n open at b, let $u \geq b$ lie in it, and let $t_1 < \cdots < t_l$ be the indices in $[b, u)$ at which the acting fiber is not n. Then

$$g_n^u(\gamma^u) \approx \big(\Psi^{t_l} \circ \cdots \circ \Psi^{t_1}\big)\big(\gamma^b\big) \tag{56}$$

That is, applying n's accumulator at γ^u yields, up to the control fields, the state those same steps would have produced from γ^b. Reading the right side as the state reached had n never begun assumes in addition that no fiber n registers take a step in $[b, u)$, since a fiber n registers is one that would not be there to take it.

Proof. By induction on u, over the indices u with $u + 1$ in the episode. At $u = b$ the step at $b - 1$ is an L-Begin, the episode opening by Definition 53, so $g_n^b = \mathrm{id}_\Gamma$ by Table 1, the index set is empty, and the claim is $\gamma^b \approx \gamma^b$. Two facts are used at each step. Since edit^t writes control fields only,

$$\gamma^{t+1} \approx \Psi^t(\gamma^t)$$

and since every map in $\mathfrak{M}(e_n)$ writes no control field but those a registration adds, by Definition 48 together with Definition 47, each such map carries $\approx$-equal states to $\approx$-equal states.

Let step u act on n. Since the episode is open at u and $u + 1$, Lemma 54(4) excludes an L-Begin and an L-Unload of n, and O-Insert and O-Remove read a θ_n that $\mathrm{installed}_n^u$ denies, leaving two cases. Where the rule is L-Iter, L-Finish, or a landing L-Divert, Table 1 gives $\Psi^u = \mathrm{pr}_1 \circ i_n^u$ and $g_n^{u+1} = g_n^u \circ h$ for the inverse h that iteration yields. The witness condition of Definition 51 reads $h(\Psi^u(\gamma^u)) = \gamma^u$, up to $\approx$ where the iteration registers a fiber (Lemma 57), and g_n^u carries $\approx$ by the equation above, so

$$g_n^{u+1}(\gamma^{u+1}) \approx (g_n^u \circ h)(\Psi^u(\gamma^u)) = g_n^u(\gamma^u)$$

Where the rule is L-Leave, L-Raise, an aborting L-Divert, or an O-Retire of n, Table 1 gives $\Psi^u = \mathrm{id}_\Gamma$ and $g_n^{u+1} = g_n^u$, so the same equation holds with $h = \mathrm{id}_\Gamma$. Either way the induction hypothesis carries over with the index set unchanged, which is the computation of Theorem 7 one step at a time.

Let step u act on $m \neq n$. Then $g_n^{u+1} = g_n^u$ by Table 1, and $\Psi^u \in \mathfrak{M}(e_m)$, or $\Psi^u = \mathrm{id}_\Gamma$ where the rule is an orchestration rule, so independence gives

$$g_n^u(\gamma^{u+1}) \approx g_n^u(\Psi^u(\gamma^u)) = \Psi^u(g_n^u(\gamma^u))$$

which is the induction hypothesis with Ψ^u appended. □

Corollary 62. (**Terminal recovery.**) Let the sequence of steps be pairwise independent and let an episode of n open at b and close at u, whatever outcome n arrives at. Then, with $t_1 < \cdots < t_l$ as in Theorem 61,

$$\gamma^{u+1} \approx \big(\Psi^{t_l} \circ \cdots \circ \Psi^{t_1}\big)\big(\gamma^b\big) \tag{57}$$

A fiber removed by O-Remove leaves nothing behind either, its premise admitting only $\theta_n = \mathsf{Inactive}(-)$.

Proof. By Lemma 54(4) step u is an L-Unload of n, whose Ψ^u is g_n^u by Lemma 54(3), so $\gamma^{u+1} \approx g_n^u(\gamma^u)$ and Theorem 61 applies. Neither the statement nor $\approx$ mentions ζ, which by Table 1 is the one field in which the states L-Divert and L-Raise lead to differ. □

Pairwise independence is assumed of the components by the results above, and Section 3.3.2 is what discharges it: where every effect a component performs is an operation of a key and every key is commutative, any two effect functions built from those operations are independent (Theorem 42). Carrying that result from effect functions to iterators calls for nothing new, a coeffect-mediated effect function (Definition 41) already choosing what follows each stage by the outcome that stage yields, which is what an iterator carries in its continuation. The coeffect operations of Section 3.2 are the case that needs no hypothesis at all: the maps a component contributes there are composites of set operations and of the corresponding restrictions, two such commute whenever they touch disjoint keys, and clause (2) of Definition 58 makes the provisions of distinct fibers disjoint.

4.4.3. Spatial Composability

Local spatial composability holds a component to its own specification, activating it only where its dependencies are provided and classifying every context change against them (Section 3.2.2). The global form adds what quantifies over other fibers: a provider withdraws a binding only after every dependent that resolved it has deactivated, and the resolution a transition installs its effects against does not shift under it. Two properties of the coeffect side deliver the two, and they are proved together, being two halves of one invariant, namely the fixity of ω_n over an episode that Lemma 54(2) establishes. The ordering theorem is what that fixity buys over the part of the episode in which n is Active and then Unloading, and the coherence theorem what it buys over the part in which n is installing its effects.

Theorem 63. (**Ordering.**) A fiber begins a transition only where its dependencies are provided:

$$\mathrm{step}^t = \text{L-Begin}(m) \Rightarrow \gamma^t \vDash d_m \tag{58}$$

Let further $[b', u']$ be an episode of m with $\omega_m^{b'}(k) = n$ for some $m \neq n$ and $k \in d_m$, let $[b, u]$ be the episode of n containing b', and let t range over $[b', u']$. Then

1. $\omega_m^t(k) = n$;
2. $b < b'$, and $u' < u$ if $[b, u]$ closes;
3. $k \in \mathrm{dom}(\sigma_n^t)$ and $\sigma_n^t(k) = \sigma_n^{b'}(k)$.

Proof. The first claim is the premise $\mathrm{target}_m^t \neq \bot$ of L-Begin, which by Definition 46 gives $\gamma^t \vDash d_m$.

(1) is Lemma 54(2).

For (2), the L-Begin at $b' - 1$ writes $\omega_m^{b'} = \mathrm{target}_m^{b'-1}$, whose values are providers, so $\theta_n^{b'} = \mathsf{Active}(-,-)$; the L-Begin at $b - 1$ leaves $\theta_n^b = \mathsf{Reloading}(-,-,-)$, so $b \neq b'$ and hence $b < b'$, both episodes opening by Definition 53. Let $[b, u]$ close and suppose $u \leq u'$. Then $u \in [b', u']$, so $\mathrm{installed}_m^u$ and, by (1), $\omega_m^u(k) = n$; that is relied_n^u, which the L-Unload at u denies. Hence $u' < u$.

For (3), n is the provider of k at $\gamma^{b'}$, so $k \in \mathrm{dom}\big(\sigma_n^{b'}\big)$. No L-Unload of n falls in $[b', u']$: where $[b, u]$ closes it falls at $u > u'$ by (2), and where it does not, Lemma 54(4) leaves n with no L-Unload at all. Since $\theta_n^{b'} = \mathsf{Active}(-,-)$, Table 1 therefore leaves L-Leave as the only rule n can be acted on by within $[b', u']$, and its Ψ^t is id_Γ; by Lemma 54(1) σ_n is constant there. □

A transition spread over steps could otherwise install effects computed against a resolution that has changed under it, and two premises prevent that. L-Iter and L-Finish carry $\mathrm{target}_n(\gamma) = \omega$, so a transition proceeds only while its committed view is still its target view, and L-Divert carries the negation, so any change to the target view takes the fiber out of the transition. L-Raise is not conditioned on the target view at all, a raise being something the iteration does rather than something the environment asks for, and it exits the transition in any case. The two directions of change are not distinguished: a component whose dependency has gone and one whose dependency has been replaced leave by the same route, because a target view that has become $\bot$ and one that has become some other fiber are equally unequal to ω.

Inertia is what stops this from being a guarantee about every step. An iteration already in flight when the target view turns lands regardless, by L-Divert, and that landing installs an effect computed against a resolution that no longer holds. What the rules deliver is therefore a disjunction, and the second branch is what makes the first safe.

Theorem 64. (Resolution coherence.) Let an episode $[b, u]$ of n open at b with $\omega_n^b = \omega$. Then θ_n is $\mathsf{Reloading}(-,-,-)$ on an initial interval $[b, r]$ of the episode, and every iteration of the transition runs against the one resolution ω:

$$\forall t \in [b, r].\ \mathrm{step}^t \in \{\text{L-Iter}(n), \text{L-Finish}(n)\} \Rightarrow \mathrm{target}_n^t = \omega \tag{59}$$

Where the fiber leaves that interval, so that $r < u$, exactly one of the following holds:

1. $\mathrm{step}^r = \text{L-Finish}(n)$ and $\theta_n^{r+1} = \mathsf{Active}(-, \omega)$;
2. $\mathrm{step}^r \in \{\text{L-Divert}(n), \text{L-Raise}(n)\}$, and the episode closes at some $u > r$ with $\gamma^{u+1} \approx \big(\Psi^{t_l} \circ \cdots \circ \Psi^{t_1}\big)\big(\gamma^b\big)$ as in Corollary 62.

Proof. The L-Begin at $b - 1$ writes $\mathsf{Reloading}$, and by Table 1 it is the one rule leading into that lifecycle state; its premise $\theta_n = \mathsf{Inactive}(\bot)$ and Lemma 54(4) put any second application of it outside the episode. So $\mathsf{Reloading}$ occupies an initial interval $[b, r]$ of $[b, u]$ and is not re-entered.

The first claim is then the premise $\mathrm{target}_n(\gamma) = \omega'$ that Table 1 gives L-Iter and L-Finish, together with $\omega' = \omega$ by Lemma 54(2).

For the dichotomy, step^r is a rule whose premise has $\theta_n = \mathsf{Reloading}(-,-,-)$ and whose conclusion does not, of which Table 1 offers L-Finish, L-Divert, and L-Raise; the first lands in $\mathsf{Active}(-,\omega)$ and the other two in $\mathsf{Unloading}(-,\omega,-)$, from which Lemma 54(4) makes an L-Unload the only exit and Corollary 62 supplies the equation. The iteration a landing L-Divert contributes is one of n's own, hence among the maps that accumulator withdraws. Where instead $r = u$, the sequence ends with the transition still in flight and the first claim is all that is asserted. □

4.4.4. Progress

A guard that defers a provider's withdrawal until its dependents are gone delivers Theorem 63 only if it eventually releases. One relation on the fibers of a registry carries the argument.

Definition 65. The *precedence relation* on the names of a registry is

$$n \prec m \coloneqq p_n \cap d_m \neq \varnothing \tag{60}$$

so that n may provide a key m declares. It reads d and p alone, which by Lemma 54(5) come into existence with a fiber's entry and are never written again.

Theorem 66 and Theorem 73 are established on the hypothesis that $\prec$ is acyclic, which is an assumption and not something the definition delivers, $n \prec n$ holding of a component that declares a key it provides itself. What $\prec$ orders is the two fibers' activations and not their lifetimes: $n \prec m$ says that n has to become Active before m can, whereas that a provider outlives its consumer is Theorem 63(2), a theorem about the guarded calculus.

A fiber's target view answers to the fiber that created it as well as to its providers. What a creator writes is τ_n, through the primitive of Definition 47, and τ is monotone by Lemma 54(5). A creator can therefore turn its child's target view at most once over that child's whole existence.

Progress is a claim that some rule applies, so it is formulated over the rules a host must offer: L-Begin, L-Leave, L-Unload, the landing rules L-Iter, L-Finish, and L-Raise, and L-Divert. It appeals to the aborting alternative of L-Divert nowhere, so a host bound by the inertia of Section 4.3.3 is covered as well.

Theorem 66. (Progress.) Assume $\prec$ acyclic, $\mathrm{len}(e_n) \le K$ for every n, and the set N of names of Definition 60 finite; and let every step apply a lifecycle rule. Write $S(n)$ for the number of steps acting on n and

$$V(n) \coloneqq \left|\left\{t : \mathrm{target}_n^t \neq \mathrm{target}_n^{t+1}\right\}\right| \tag{61}$$

for the number of times its target view turns. Then

1. (*No deadlock.*) $\neg\,\mathrm{quiet}^t$ implies that some lifecycle rule applies at γ^t;
2. (*Termination.*) $S(n) \le (K+4)(V(n)+1)$, and both $V(n)$ and $\sum_n S(n)$ are finite.

Consequently every maximal sequence of lifecycle steps ends in a quiescent state.

Proof. No deadlock. Let $\neg\,\mathrm{quiet}^t$, so some fiber n satisfies neither clause of the quiet of Definition 49. Reading Table 1 against the four kinds it can then be:

- $\theta_n^t = \mathsf{Inactive}(\bot)$ with $\mathrm{target}_n^t \neq \bot$: L-Begin applies;

- $\theta_n^t = \mathsf{Reloading}(-,-,\omega_n)$ with $\mathrm{target}_n^t = \omega_n$: whichever of L-Iter, L-Finish, and L-Raise the value of $i_n^t(\gamma^t)$ selects applies;
- $\theta_n^t = \mathsf{Reloading}(-,-,\omega_n)$ with $\mathrm{target}_n^t \neq \omega_n$: L-Raise applies if $i_n^t(\gamma^t)$ raises, and otherwise L-Divert does, landing that iteration rather than aborting it;
- $\theta_n^t = \mathsf{Active}(-,\omega_n)$ with $\mathrm{target}_n^t \neq \omega_n$: L-Leave applies.

Let no fiber be of any of these kinds, leaving some m_0 with $\theta_{m_0}^t = \mathsf{Unloading}(-,-,-)$. Construct $m_0, m_1, \ldots$ as follows: given m_j in $\mathsf{Unloading}$, either $\neg\,\mathrm{relied}_{m_j}^t$, in which case L-Unload applies to m_j and the construction stops, or there are $m_{j+1} \neq m_j$ and k_j with $\mathrm{installed}_{m_{j+1}}^t$ and $\omega_{m_{j+1}}^t(k_j) = m_j$. In the latter case

$$k_j \in d_{m_{j+1}} \cap \mathrm{dom}\big(\sigma_{m_j}^t\big) \subseteq d_{m_{j+1}} \cap p_{m_j}$$

the second membership being Theorem 63(3) at the episode of m_{j+1} that t lies in, so that $m_j \prec m_{j+1}$. Moreover $\mathrm{target}_{m_{j+1}}^t \neq \omega_{m_{j+1}}^t$: an $\mathsf{Unloading}$ fiber is outside the union defining σ_γ, so k_j at γ^t is unprovided or provided by a fiber other than m_j. Were m_{j+1} in Active or $\mathsf{Reloading}$ it would then be of one of the four kinds excluded, so it is in $\mathsf{Unloading}$ and the construction continues. The m_j are $\prec$-increasing, hence distinct by acyclicity, and $\mathrm{dom}(F^t)$ is finite, so the construction stops.

Termination. Two claims bound $S(n)$.

(A) Over a maximal interval on which target_n^t is constant at ω^*, at most $K+4$ steps act on n. Reading the θ_n columns of Table 1, from $\mathsf{Active}(-,\omega)$ with $\omega \neq \omega^*$ the fiber takes an L-Leave and an L-Unload and then, if $\omega^* \neq \bot$, an L-Begin and at most $\mathrm{len}(e_n) \leq K$ landings, plus a second L-Unload where the last landing is an L-Raise; from $\mathsf{Reloading}$ against an $\omega \neq \omega^*$ it takes an L-Divert in place of the L-Leave, and from any other state a suffix of that sequence. No further L-Divert or L-Leave falls in the interval, the ω that the L-Begin writes being $\mathrm{target}_n^t = \omega^*$ itself, and at $\mathsf{Active}(-,\omega^*)$, at $\mathsf{Inactive}(\bot)$ with $\omega^* = \bot$, and at $\mathsf{Inactive}(\xi)$ no rule applies at all.

(B) If $\mathrm{target}_n^t \neq \mathrm{target}_n^{t+1}$ and step t acts on m, then either $m \prec n$ or step t writes τ_n. By Definition 46 the value of target_n is a function of τ_n and of the tables of the providers of the keys of d_n; a provider satisfies $k \in \mathrm{dom}(\sigma_m) \cap d_n$ and hence $m \prec n$, and a table changes only at a step acting on its own fiber by Lemma 54(1). Acyclicity gives $m \neq n$ in the first case, and the monotonicity of Lemma 54(5) admits the second at one t per fiber.

By (A) the interval count bounds $S(n)$ as $S(n) \leq (K+4)(V(n)+1)$, and by (B) each turn of target_n either consumes a step of a fiber strictly $\prec$-below n or is the one turn τ_n affords, so $V(n) \leq 1 + \sum_{m \prec n} S(m)$. Since $\prec$ is acyclic and N is finite, the recursion

$$B(n) := (K+4)\left(2 + \sum_{m \prec n} B(m)\right)$$

is well founded and defines B with $S(n) \leq B(n)$; hence $V(n)$ is finite and $\sum_n S(n) \leq \sum_n B(n)$. By (1) a sequence that cannot be extended is quiescent. □

Finiteness of N is assumed rather than derived, and one condition on the components delivers it. The components a host holds are finitely many programs given before anything runs, so if no component can register, however indirectly, a fiber of a component that registers one of its own, the registrations form a tree of bounded depth, and $\mathrm{len}(e_n) \leq K$ bounds its branching. What the assumption rules out is a component that registers instances of itself without bound.

The target records the providing fiber rather than a boolean, and under the single-source discipline of Section 4.2 the two drive the same transitions, a key having one possible provider there. What the view buys is the vocabulary of the results above, Theorem 63 and Theorem 64 both speaking of the resolution a fiber activated against, and it is what makes those results survive the scoped resolution of Section 3.2.3, under which one key resolves to different providers in different realms and the provisions no longer force the view. The implementation carries that scoping and holds the view in `fiber.committed` (Section 5.1.3).

4.4.5. Confluence

The results so far are about individual fibers. The property that characterizes the system as a whole is that its dynamic history leaves no trace: whatever sequence of activations and deactivations a running system has been through, the state it quiesces at is the one the same insertions and retirements would have produced had each component that ends up active been loaded once, in dependency order, and none ever unloaded. The lifecycle relation is confluent, and the normal form it converges on is the statically assembled one. This is the analogue, for dynamic composition, of the consistency with a from-scratch evaluation that change propagation establishes for incremental computation [45].

The claim is about $\longrightarrow$ alone. Orchestration steps are inputs, and two sequences given different inputs land in different places for no interesting reason; what is at issue is whether the lifecycle rules, which are nondeterministic in which fiber steps next and in which exit a Reloading fiber takes, can be made to disagree.

Three lemmas are needed first. The first fixes the set of fibers that end up Active without reference to any sequence of steps, which is what makes it a function of the input rather than of the schedule.

Definition 67. A fiber is *supported* at γ when it is not retired, the fiber registering it is supported, and every key it declares is provided by a supported fiber. The *support relation* on $\mathrm{dom}(F_\gamma)$ is the union of the two relations those clauses read,

$$m \lhd n := m \prec n \vee \pi_n = m \tag{62}$$

and where it is well founded (Lemma 68) we write A for the *support set*, the fibers supported at γ:

$$n \in A := \neg\tau_n \wedge (\pi_n = \mathsf{root} \vee \pi_n \in A) \wedge \forall k \in d_n.\ \exists m \in A.\ k \in p_m \tag{63}$$

where $\pi_n = \mathsf{root}$ marks a fiber the orchestrator inserted and π_n otherwise the fiber whose activation registers n. The clauses read no field but τ, π, d, p. Both halves relate a fiber to one immediately below it, a parent rather than an ancestor and a direct provider rather than a transitive one, since that is what the clauses read; where the results below want an order they take the transitive closure, whose minimal elements, maximal elements, and linearizations are those of $\lhd$.

The clauses refer to A itself, so the definition is a recursion along $\lhd$, and it is the following that makes it one with a solution.

Lemma 68. (Support is well founded.) Let $\prec$ be acyclic and let γ be reached by a sequence of steps. Then $\lhd$ is well founded, and A is the one solution of Definition 67, a function of τ, π, d, and p alone.

Proof. Order the names of $\mathrm{dom}(F_\gamma)$ by the index of the step that registered each, which Definition 53 supplies by starting the sequence at an empty registry. The parent half of $\lhd$ descends in that index: an O-Insert has $\pi \in \mathrm{dom}(F_\gamma)$ as a premise, so a parent pointer names a fiber registered earlier, and iterating it reaches the whole ancestry of a name in finitely many steps. A cycle therefore has to use $\prec$, and since $\prec$ is acyclic it has to mix the two, which needs some m to declare a key that a fiber of m's own subtree may provide. Such a fiber is registered by an activation of m or of one of m's descendants, hence at a step after the L-Begin of m; that L-Begin has $\gamma \vDash d_m$ as a premise, so a fiber providing the key is Active already before it, and clause (2) of Definition 58 leaves the key no second possible provider. The fiber that would close the cycle is therefore never registered, and the edge is absent from $\mathrm{dom}(F_\gamma)$. A well-founded recursion has one solution, and the clauses read the four fields alone. □

The last clause reads p, the keys a component may provide, whereas the target reads $\mathrm{dom}(\sigma_\gamma)$, the keys its fibers have installed, and Definition 43 relates the two by $\mathrm{dom}(\sigma_n) \subseteq p_n$ alone. The support set therefore over-approximates the Active fibers in general, and the condition that closes the gap is the following.

Definition 69. A component (d, p, e) is *total on its provision* when an activation of it that finishes has installed every key of p, so that $\mathrm{dom}(\sigma_n) = p_n$ at every Active fiber instantiating it.

Like independence (Definition 60) this is a condition on the components alone, mentioning no lifecycle state and no step, and independence already bounds how far it can fail: were a component to install a key only at context states another component's effects reach, its forward map would not commute with that component's, so the keys a fiber installs are fixed by its component rather than by the schedule. What totality adds is that the fixed set is all of p rather than a proper subset of it.

Lemma 70. (Support at quiescence.) Let $\prec$ be acyclic, let $\mathrm{quiet}(\gamma)$, let no fiber of γ be failed, and let every component of γ be total on its provision (Definition 69). Then the support set is the set of Active fibers:

$$A = \{n : \theta_n = \mathsf{Active}(-,-)\} \tag{64}$$

Proof. Write A' for the right-hand side. No fiber being failed, the quiet of Definition 49 leaves $\mathsf{Inactive}(\bot)$ and Active as the only states and reads

$$n \in A' \iff \mathrm{target}_n(\gamma) \neq \bot$$

By Definition 46 the right side holds exactly when $\neg\tau_n$ and every $k \in d_n$ lies in $\mathrm{dom}(\sigma_\gamma)$, and $\mathrm{dom}(\sigma_\gamma) = \bigcup_{m \in A'} p_m$ by Definition 69. The middle clause is the one the target no longer carries, and registration supplies it: a fiber with $\pi_n \neq \mathsf{root}$ is registered only by an activation of π_n, and if $\pi_n \notin A'$ then π_n is not Active, so its accumulator has run and retired n by Definition 47, giving τ_n. Hence A' satisfies the clauses of Definition 67, and Lemma 68 gives them one solution, so $A = A'$. □

Lemma 71. (Transposition.) Let the steps be pairwise independent and F^t well formed, and let steps t and $t+1$ act on distinct fibers m and n.

1. If both apply an activation rule, namely L-Begin, L-Iter, or L-Finish, and step $t+1$ is applicable at γ^t, then step t is applicable at the state step $t+1$ produces from γ^t, and the two orders reach the same γ^{t+2}.

2. If step t applies an activation rule at m, step $t+1$ an orchestration rule at n, and step t does not register n, then the same holds of the two.

Proof. For (1), by Table 1 the step of m writes θ_m and, within $\Psi^t \in \mathfrak{M}(e_m)$, the table σ_m and the effect part. It therefore leaves θ_n and i_n alone, and by the second condition of Definition 60 leaves the inverse and the continuation that i_n yields alone as well, so only the premises of step $t+1$ that mention target_n remain to be checked. Its retirement half cannot fall, no activation rule writing a τ. Its resolution half cannot move either: step $t+1$ being applicable at γ^t puts every $k \in d_n$ in $\mathrm{dom}(\sigma^t)$, and clause (2) of Definition 58 makes the fiber providing such a k the only one that can, so $k \notin p_m$ and no write of σ_m reaches a key of d_n. The same argument in the other direction leaves step t applicable. Finally $\Psi^t \in \mathfrak{M}(e_m)$ and $\Psi^{t+1} \in \mathfrak{M}(e_n)$ commute by the first condition of Definition 60, and the two edits write control fields of distinct fibers, so the composite is the same in either order.

For (2), the orchestration step has $\Psi^{t+1} = \mathrm{id}_\Gamma$ by Table 1, so the two state maps commute outright, and its edit^{t+1} writes τ_n or $\mathrm{dom}(F_\gamma)$ at n alone, which the activation step neither reads nor writes: the premises of the latter read θ_m, i_m, τ_m, and target_m, and an O-Insert of a fresh n moves no target, a fresh fiber providing nothing, whereas an O-Retire or O-Remove of n leaves σ_γ where it was, n being $\mathsf{Inactive}$ in the one case and unaffected in its table in the other. So step t remains applicable. Conversely each premise of the orchestration step is either read at n, which step t does not write, or is one of the two premises of O-Insert that a smaller registry only relaxes, whence its applicability at γ^{t+1} gives its applicability at γ^t; here step t not registering n is what keeps n present at γ^t where O-Retire and O-Remove require it. □

Lemma 72. (Deletion.) Let the sequence of steps be pairwise independent, let every component be total on its provision (Definition 69), let it reach a quiescent γ^T at which no fiber is failed, let $[b,u]$ be an episode of n that closes, let no episode of any m with $n \prec m$ close in the sequence, and let no fiber n registers during $[b,u]$ have an episode. Write R for the names those registrations draw. Then deleting the steps that act on n in $[b,u]$, together with every step acting on a name of R, leaves a sequence of steps reaching a state $\approx$-equal to γ^T and $\simeq$-equal to it outside R.

Proof. The deleted steps leave the state where they found it. Let $t_1 < \cdots < t_l$ be the steps of $[b,u]$ that act on fibers other than n. Corollary 62 reads

$$\gamma^{u+1} \approx \big(\Psi^{t_l} \circ \cdots \circ \Psi^{t_1}\big)\big(\gamma^b\big)$$

whose right side is what the surviving steps of $[b,u]$ produce on their own, $\gamma^{b-1} \approx \gamma^b$ and their edits writing control fields of fibers other than n that the deletion does not touch. By Table 1 the deleted steps of n write no field but θ_n, which Lemma 54(4) restores to $\mathsf{Inactive}(\bot)$ at u, no fiber being failed, and which it held at γ^{b-1}.

An invariant carries the suffix. Write γ'^t for the state the surviving steps reach at the point corresponding to t. We claim, for every $t > u$, that $\gamma^t \approx \gamma'^t$, that every name of R is vestigial at γ^t and absent from γ'^t, and that the two states agree on every field of every name outside R. At $t = u+1$ this is the paragraph above together with Definition 47, which leaves each name of R retired by the accumulator that ran at u, $\mathsf{Inactive}(\bot)$ and holding an empty table, the fibers of R having no episode by hypothesis. The induction step is Lemma 57(1) applied at each name of R in turn: a step acting outside R has the same premises at the two states, reaches states again $\approx$-equal, and leaves the entries of R vestigial. A step acting on a name of R is one of the deleted ones, and Lemma 57(2) is why it has to be deleted rather than kept, an O-Retire or O-Remove

of an absent name having no fiber to act on; by (1) again such a step moves no field outside R, so dropping it preserves the invariant. Hence the final states are $\approx$-equal, and equal outside R.

No surviving step loses a premise. A step acting on $m \notin R \cup \{n\}$ reads n only through $\mathrm{target}_m(\gamma)$ or $\mathrm{relied}_m(\gamma)$. The first depends on n when m declares a key n provides, hence $n \prec m$, and when n registered m, which puts $m \in R$. In the first case m's episode does not close, by hypothesis, so it is open at γ^T, where quiet gives $\omega_m = \mathrm{target}_m^T$ and Lemma 70 puts its values among the Active fibers, which n is not; since a key has at most one possible provider, n provided no key of d_m at m's L-Begin either. The second reads n only through the values of ω_n, and deleting the episode can only make relied false, which relaxes the guard on L-Unload rather than blocking it. What such a step reads of a name of R is covered by the invariant. Pairwise independence is a property of the effect functions, so deleting steps preserves it. □

Theorem 73. (**Confluence.**) Let a sequence of steps reach a quiescent γ^T at which no fiber is failed, let the steps be pairwise independent and every component be total on its provision (Definition 69), and let A be as in Definition 67. Then

1. (*Canonical form.*) γ^T is reached, up to the names whose entries the reduction withdraws, from γ^0 by a sequence that takes the same orchestration steps in their original order, those at a fiber the orchestrator inserted preceding every lifecycle step and each of the rest following the step that registered the fiber it acts on, and that takes, for an enumeration $n_1, ..., n_k$ of A linearizing $\lhd$, one episode of each n_i in that order.
2. (*Confluence.*) Any two such sequences from γ^0 taking the same orchestration steps reach states related, after a renaming as in Lemma 56, by $\simeq$ and by $\approx$.

Proof. For (1), the episodes of the sequence are of two kinds: those that close and those still open at γ^T, which by quiet^T and Lemma 70 are one episode of each fiber of A.

Closing episodes go first, by induction on their number. At each stage pick a closing episode of a fiber n that is $\lhd$-maximal among the fibers whose episodes still close; one exists by Lemma 68 and the finiteness of N. The three hypotheses of Lemma 72 are then met. No m with $n \prec m$ has a closing episode, by maximality. And no fiber n registers during $[b, u]$ has an episode: such a fiber is retired by the accumulator that ran at u (Definition 47) and by Lemma 54(5) stays retired, so its target view is $\bot$ and Lemma 70 puts it outside A, whence it has no episode open at γ^T; and $\lhd$ relates it to n through its parent pointer, so by maximality it has no closing one either. The lemma removes the episode, together with the steps of the names it registered, leaving γ^T where it was up to those names. The measure drops by one, so no closing episode remains.

A fiber outside A takes no lifecycle step. It has no open episode at γ^T, by Lemma 70 and quiet^T, and no closing one now remains, so it has no episode at all and is $\mathsf{Inactive}(\bot)$ throughout; L-Begin is the only rule that applies there, and applying it would open an episode.

Orchestration steps go next. An orchestration step at a fiber the orchestrator inserted moves one place earlier past a lifecycle step of a different fiber by Lemma 71(2), which applies because a step of a fiber of A registers no such name: registrations draw fresh names, whereas the name here is one an O-Insert of the original sequence introduced. With a lifecycle step of the same fiber there is nothing to exchange, an O-Insert of n already preceding every step of n and an O-Retire or O-Remove of n applying only outside A, which takes no lifecycle step. Moving each to the front in turn preserves their relative order. An orchestration step at a fiber some activation registered cannot go to the front, its premises requiring that fiber to be present, so it stays where the registration put it; it acts outside A by the paragraph above and therefore commutes with everything between it and the registration by the same clause of Lemma 71.

Episodes are sorted and made contiguous, by induction on $|A|$. Let n_1 be $\lhd$-minimal in A. Then $d_{n_1} = \varnothing$ and $\pi_{n_1} = \mathsf{root}$, since Definition 67 puts a provider of a key of d_{n_1} and the fiber registering n_1 in A while $\lhd$ puts both below n_1. So target_{n_1} reads no field of another fiber and, no orchestration step remaining to write τ_{n_1} and no fiber below n_1 remaining to retire it, is constant. Every step acting on n_1 is an activation step, no episode closing, and its remaining premises read θ_{n_1} and i_{n_1}, which by Table 1 only n_1 writes; each is therefore applicable at every earlier state, and Lemma 71 moves it one place earlier without moving the endpoint. The number of steps of other fibers preceding a step of n_1 drops by one at each application, so the episode of n_1 becomes an initial contiguous block. The argument repeats on $A \setminus \{n_1\}$ over the suffix that follows the block, where n_1 is Active throughout and takes no further step, so it too contributes a constant target. The enumeration this produces linearizes $\lhd$ by construction.

For (2), both sequences reduce by (1) to a canonical one, and the two reductions run over the same A up to a renaming. Definition 67 reads τ, π, d, and p, of which the last three are written once with a fiber's entry (Lemma $54(5)$), so what has to be seen is that the same names come into existence carrying the same d, p, and π, and that the same names are retired. Insertions the two sequences share by hypothesis. Registrations they share as well: an activation of a fiber of A registers, at each of its iterations, the component the iterator names there, which the second condition of Definition 60 holds fixed across interleavings, so the tree of registrations below an A-fiber is a function of that fiber's component; the names those registrations draw are not shared, and it is here that Lemma 56 is applied, matching the two trees by a bijection. And a retirement is either an orchestration step, shared, or the O-Retire an accumulator takes, which retires exactly the names the same activation registered. Two enumerations linearizing $\lhd$ differ by transpositions of incomparable episodes, which Lemma 71 again leaves the endpoint unchanged by, so the two canonical sequences agree. With the termination of Theorem 66, the lifecycle relation therefore has unique normal forms. □

Failure is excluded from the statement because it is a genuine source of divergence, and the calculus should not be read as denying it: whether a step raises depends on the state it ran against, so one schedule may fail a fiber where another completes it, and the two quiescent states then differ in that fiber's lifecycle state. They do not differ in anything else, by Corollary 62, which puts a failed fiber's contribution to the state at nothing.

In the base calculus of Section 4.2 the same theorem holds, and the proof needs no substitution beyond dropping one clause. L-Unload carries no guard there, so the last paragraph of Lemma 72 is vacuous; the rest of that lemma appeals to quiet^T alone, which the base calculus supplies unchanged.

The theorem is what licenses reasoning about a Cordis application as though it were statically assembled. An orchestrator that adds a component, removes it, replaces a provider, and reverts the replacement is guaranteed to arrive at the state it would have obtained by writing the final composition down at the outset, and a component author reasoning about which coeffects are in scope may reason about the quiescent state alone. It also delimits the guarantee: it speaks of the state, not of the emissions the system produced along the way, which is the distinction Section 6.1 draws between an acquisition, tracked inside the boundary, and an emission, which crosses it.

5. Implementation and Case Study

This section presents *Cordis*, which realizes the formal models of Section 3 as a practical programming abstraction. Cordis is a *meta-framework* of spatiotemporal composability: unlike application frameworks that target a specific domain (e.g., web routing, ORM, UI rendering), it prescribes no concrete scenario; its sole responsibility is to supply universal dynamic composition semantics. The implementation is layered into three tiers: (1) the *core library* (Section 5.1) implements the effect and coeffect systems directly; (2) the *component loader* (Section 5.2) extends the core with configuration reconciliation and hot module replacement; and (3) application frameworks such as *Koishi* (Section 5.3) build domain-specific functionality on top of the former two tiers.

5.1. Core Library

Table 2 summarizes the correspondence between theoretical constructs and their runtime counterparts. In particular, we use the runtime names introduced below throughout this section, reserving the theoretical symbols for the formal correspondence. We also write `@@name` for a framework-internal symbol key, so the brackets in `ctx[@@store]` denote symbol-keyed access to an opaque slot on the context, rather than indexing into a string-keyed map.

Theory (Section 3, Section 4)	**Implementation**
Γ_∞	`ctx`, the first-class context
$\gamma \in \Gamma$	the context tree together with everything the running system has touched
$\mathfrak{E}_\Gamma$, $\mathfrak{E}_\Gamma^{\text{iter}}$	Effect callback returning / yielding inverses
$\text{effect}_\Gamma(e)$	`ctx.effect(callback)`
Σ, Σ^{iso}, Σ^{inter}	`ctx[@@store]`, `ctx[@@isolate]`, `ctx[@@intercept]`
$\text{get}(k)$, $\text{set}(k, v)$	`ctx.get(key)`, `ctx.set(key, value)`
$\text{isolate}(k, r)$	`ctx.isolate(key, realm)`
$\text{intercept}(k, \nu)$	`ctx.intercept(key, metadata)`
$\langle d, p, e, \pi, \sigma, \tau, \theta \rangle$	`fiber`, the instantiation of a component in $\mathfrak{C}_\Gamma$
$\text{dom}(F_\gamma)$	enumerated through `ctx.registry`
$n : \mathfrak{N}$	`fiber.uid`
$d : \mathfrak{D}_\Gamma$	`fiber.inject`
$p : \mathfrak{P}_\Gamma$	the component's `provide`
$e : \mathfrak{E}_\Gamma^*$	`fiber.apply`
$\pi : \mathfrak{N}$	`fiber.parent.fiber.uid`, the fiber owning the context it was instantiated on
derived realization (Definition 27)	`fiber.ctx`, the child context the fiber runs in
θ (Definition 44)	`fiber.state`, the lifecycle state, whose `LOADING` is Reloading and whose `FAILED` is Inactive(ξ)
recover, accumulator g	`fiber.dispose`, the accumulator
ω (Definition 44)	`fiber.committed`, the committed view
$\text{provider}_k(\gamma)$	an `Impl` whose provider fiber is `ACTIVE`
$\text{target}(\gamma, n)$	`fiber.target`, recomputed by `refresh` (Algorithm 5), where $\perp$ is `INACTIVE`
Future, inertia (Section 4.3.3)	`fiber.inertia`, the handle of the transition in flight
O-Insert, O-Retire (Definition 47)	`ctx.use` and the inverse of its `callback` (Algorithm 4)
O-Remove	the fiber dropped from its runtime, with `uid` cleared
L-Begin, L-Iter, L-Finish	`execute`'s iteration loop (Algorithm 1)
L-Divert	the `guard` failing at an iteration boundary (Algorithm 1), or `reload` chaining into `unload`
L-Leave	`refresh` marking the fiber `UNLOADING` (Line 10)
L-Unload	`unload` and its inertial chaining (Algorithm 5)
guard on L-Unload	`unload` awaiting the notified dependents (Line 25)
L-Raise	the error recorded on the fiber, with its `target` set to $\perp$

Table 2 | Theory-to-implementation correspondence

The remainder of this section builds the core library from the bottom up. Section 5.1.1 realizes revertible effects, the sole primitive through which a context is mutated; Section 5.1.2 realizes reactive coeffects over it; Section 5.1.3 composes both into the component lifecycle; and Section 5.1.4 exposes the context-level operations built on them.

5.1.1. Effect Tracking

This section realizes revertible effects (Section 3.1). Every context mutation in Cordis flows through a single primitive, `ctx.effect`: coeffect provision, component instantiation, and every other context-mutating operation reduces to a `ctx.effect` call, so any operation performed through the context is automatically tracked and recovered upon component unloading. Operationally, `ctx.effect` is the realization of $\mathrm{effect}_\Gamma^{\mathrm{iter}}$ (Definition 52): it takes a callback of type $\mathfrak{E}_\Gamma^{\mathrm{iter}}$ and lifts it to $\mathfrak{E}_{\partial\Gamma}^{\mathrm{iter}}$, yielding a dispose closure that, when invoked, recovers the effect. Cordis accepts both $\mathfrak{E}_\Gamma$ and $\mathfrak{E}_\Gamma^{\mathrm{iter}}$ through this one operation (ad-hoc polymorphism); we take the iterator form as representative, since a plain effect function is the degenerate iterator that yields a single inverse. What the operation does not check is the witness that $\mathfrak{E}_\Gamma^*$ carries: the callback supplies an inverse, and that the inverse recovers the effect it accompanies is an obligation on the component author rather than a property the runtime verifies. Theorem 61 is where the calculus appeals to it, and Section 6.1 is where the obligation is delimited.

Algorithm 1 shows the construction of `ctx.effect`. We write $f \circ g$ for the disposer that runs f after g, and `id` for the no-op; prepending each new inverse therefore yields LIFO recovery.

Algorithm 1 Effect tracking

```
1  async function execute(callback, guard)
2      iter ← callback()
3      inverse ← id
4      while guard()
5          (value, done) ← await iter.next()
6          if value then inverse ← value ∘ inverse
7          if done then break
8      return inverse
9  function effect(ctx, callback)
10     armed ← true
11     task ← execute(callback, () ↦ armed)
12     async function dispose()
13         if not armed then return
14         armed ← false
15         recover ← await task
16         recover()
17     ctx.dispose ← dispose ∘ ctx.dispose
18     return dispose
```

The engine `execute` drives the callback as an effect iterator ($\mathfrak{E}_\Gamma^{\mathrm{iter}}$, Definition 51) and folds the inverse yielded at each step into a single composite. Before each step it consults a caller-supplied `guard`; once the guard trips, iteration stops and only the inverses accumulated so far remain. This is the step-boundary interruption of Section 4.3.2: the $\mathsf{Maybe}(\mathfrak{E}^{\mathrm{iter}})$ continuation is realized by the iterator's `done` flag together with `guard`.

`ctx.effect` is a thin wrapper over `execute` that adds two things. First, self-disposal: the guard reports the `armed` flag, and the returned `dispose` flips `armed` to false, which simultaneously halts any in-flight iteration and makes recovery fire at most once. Firing twice would apply an inverse at a state no application of the effect produced, where nothing holds it to reverting anything. Second, parent composition: `dispose` is prepended to the enclosing context's accu-

mulated inverse `ctx.dispose`, so a child effect's inverse is itself an effect on the parent, which is the recursive structure of $\partial^2\Gamma$. The component level (Section 5.1.3) reuses the same `execute` with a guard that tests the stability of `fiber.target` instead of `armed`.

5.1.2. Coeffect Operations

This section realizes reactive coeffects (Section 3.2). All coeffect operations act on three symbol-keyed slots that each context carries:

- `@@store`: the value store $\sigma : (r : R) \rightharpoonup \mathcal{V}_r$ from realm symbols to typed values;
- `@@isolate`: the realm table $\rho : \mathrm{Map}(K, R)$ from coeffect keys to realm symbols;
- `@@intercept`: the interception table $\iota : (k : K) \to \mathcal{M}_k$ assigning each key its metadata.

The first two compose into the two-layer resolution $k \to \rho(k) \to \sigma(\rho(k))$: `ctx.get(key)` (Algorithm 2) reads the realm symbol $\rho(k)$ from `@@isolate`, then the bound value $\sigma(\rho(k))$ from `@@store`. The ρ indirection lets isolation redirect a key to an independent binding, whereas `@@intercept` is consulted only when a binding is accessed, adjusting how it is used rather than what it resolves to. We realize these operations in two parts: (1) *provision and notification*, which install or retract bindings and propagate the change to dependents; and (2) *isolation and interception*, which reshape how a key resolves.

Provision and notification. Since $\mathrm{set}(k, v)$ has type $\mathfrak{E}_\Sigma$ (Section 3.1), coeffect provision is a `ctx.effect` call and inherits its automatic tracking and recovery. Algorithm 2 implements `ctx.set(key, value)`, the concrete $\mathrm{set}(k, v)$: the callback binds a value into the store under the realm symbol $\rho(k)$, and the returned dispose function removes it. Both installation and removal invoke `notify` to propagate the change to dependent components.

Algorithm 2 Coeffect operations

```
1  function get(ctx, key)
2      realm ← ctx[@@isolate][key] ▷ ρ(k)
3      return ctx[@@store][realm] ▷ σ(ρ(k))
4  function set(ctx, key, value)
5      function callback()
6          realm ← ctx[@@isolate][key] ▷ ρ(k)
7          ctx[@@store][realm] ← value ▷ σ[ρ(k) ↦ v]
8          notify(ctx, [key])
9          return function()
10             delete ctx[@@store][realm] ▷ σ ∖ ρ(k)
11             notify(ctx, [key])
12     return ctx.effect(callback)
```

Algorithm 3 propagates each binding change to dependents by testing, for each live fiber, whether a changed key appears in its `fiber.inject` and resolves to the same realm; if so, it calls `refresh` (Section 5.1.3) to re-evaluate that fiber against the new state, and it returns the fibers it re-evaluated so that a caller can wait for them. This is the reactive classification of Definition 26: a change that flips satisfaction activates or deactivates the fiber, and `refresh`'s idempotence renders a neutral change harmless. The interaction of this re-evaluation with diverse control flows is developed in Section 5.1.3.

Algorithm 3 Reactive notification

```
1  function notify(ctx, keys)
2      affected ← ∅
3      for fiber in all_fibers do
4          for key in keys do
5              if key ∈ fiber.inject and fiber.ctx[@@isolate][key] = ctx[@@isolate][key] then
6                  refresh(fiber)
7                  affected ← affected ∪ {fiber}
8                  break
9      return affected
```

A binding counts as available to a dependent only while the fiber that installed it is `ACTIVE`, so `refresh` resolves each declared key against an active provider rather than against the store alone. This is the *provided by* relation of Definition 46, and it is what makes a withdrawal visible to dependents one step before it happens: a provider that has entered `UNLOADING` has stopped providing, so its dependents recompute an unsatisfied target view and begin their own teardown while its bindings are all still in place.

Isolation and interception. The two operations do structurally the same thing: each derives a child context that adjusts one inherited table for `key`, leaving the parent untouched, so recovery is implicit: discarding the child context suffices, with no explicit inverse to run. `ctx.isolate(key, realm)` overrides the realm mapping ρ with `realm`, or a freshly generated symbol by default (realizing isolate, Definition 29), so two contexts that assign different symbols to the same key resolve to independent bindings. `ctx.intercept(key, metadata)` merges `metadata` into the interception table ι (realizing intercept, Definition 31): following that definition, the new metadata is combined with whatever the context already carries for `key` and takes priority over it.

5.1.3. Component Lifecycle

A component is instantiated as a *fiber* by `ctx.use`. This section gives the fiber (introduced in Section 5.1) operational meaning as the inertial state machine of Section 4.3.3. Two fields drive the algorithm below: `fiber.parent`, the parent context of `fiber.ctx` that forms the component hierarchy (the recursive structure of Γ_∞, Section 3.3.1), and `fiber.inertia`, a handle to the in-flight asynchronous transition (or null if idle).

Algorithm 4 shows component instantiation. A *component* pairs a coeffect specification `component.inject` (d) with an effect function `component.apply`; instantiation binds the component's `config` into `fiber.apply` (Line 9), the config-applied effect function (e) that the lifecycle then runs. The `callback` function (Line 2) is the effect tracked in the *parent* fiber: when executed, it initiates the child's lifecycle by calling `refresh` (Algorithm 5); when recovered, it forces the child's target to $\bot$ and triggers `unload`. This is the registration primitive of Definition 47, with `callback` as its O-Insert and the closure `callback` returns as its O-Retire: an instantiation is an ordinary tracked effect of the parent, so unloading a parent cascades to its children.

Algorithm 4 Component instantiation

```
1  function use(ctx, component, config)
```

```
    function callback()
        refresh(fiber)
        return function()
            fiber.target ← ⊥
            unload(fiber)
    fiber ← Fiber(parent: ctx, inject: component.inject)
    fiber.ctx ← ctx[fiber ↦ fiber]
    fiber.apply ← () ↦ component.apply(fiber.ctx, config)
    ctx.effect(callback)
    return fiber
```

Algorithm 5 realizes the inertial state machine of Section 4.3.3, in which `reload` and `unload` are inertial: once entered, a transition runs to completion before the system responds to a target-state change. It uses two auxiliary lookups over the coeffect store: `resolve(inject)` returns the bindings the declared keys currently resolve to, and `provided(fiber)` returns the keys whose binding this fiber installed. The `refresh` function recomputes `fiber.target` from the coeffect store and, if the fiber is not already in a transition, initiates either a `reload` or `unload` task[2]. The `reload` function records the current target and executes the component's effect function `apply`. Upon completion, it checks whether the target still matches: if so, the fiber enters `ACTIVE`; if not (regardless of whether the new target is $\perp$ or a different set of providers), it chains into `unload`. Symmetrically, `unload` recovers all tracked effects in LIFO order and then either enters `INACTIVE` or chains into `reload`. This mutual recursion implements the inertial property: once a transition begins, it completes before any new transition can start.

Algorithm 5 Component lifecycle

```
1  function refresh(fiber)
2      target ← target(γ, n)
3      if target = fiber.target then return
4      fiber.target ← target
5      if fiber.inertia then return
6      if target ≠ ⊥ then
7          fiber.state ← LOADING
8          fiber.inertia ← create_task(reload(fiber))
9      else
10         fiber.state ← UNLOADING ▷ out of service before any inverse is scheduled
11         fiber.inertia ← create_task(unload(fiber))
12 async function reload(fiber)
13     target₀ ← fiber.target
14     fiber.committed ← resolve(fiber.inject) ▷ commit the view
15     recover ← await execute(fiber.apply, () ↦ fiber.target = target₀)
16     fiber.dispose ← recover ∘ fiber.dispose
17     if fiber.target = target₀ then
18         fiber.state ← ACTIVE
```

[2] `create_task` schedules an async function to run concurrently and returns a handle to it (stored in `fiber.inertia`). We write it explicitly for language independence: with eager scheduling (e.g., TypeScript promises), the call is implicit and the returned promise is the handle, whereas with lazy scheduling (e.g., Python coroutines, Rust futures) the host must spawn the task for it to progress.

```
19          notify(fiber.ctx, provided(fiber))
20          fiber.inertia ← null
21      else
22          fiber.state ← UNLOADING
23          fiber.inertia ← create_task(unload(fiber))
24  async function unload(fiber)
25      await all(notify(fiber.ctx, provided(fiber)).map(f ↦ f.await())) ▷ drain dependents
26      await fiber.dispose()
27      fiber.dispose ← id
28      fiber.committed ← ⊥
29      if fiber.target = ⊥ then
30          fiber.state ← INACTIVE
31          fiber.inertia ← null
32      else
33          fiber.state ← LOADING
34          fiber.inertia ← create_task(reload(fiber))
```

`fiber.target` is computed by resolving each declared key against the current coeffect store and tupling the `uid` of the fiber that provides it, so it is a digest of $\mathrm{target}(\gamma, n)$ (Definition 46). Identifying a binding by its provider rather than by its value is what makes a single comparison against the recorded target sufficient: a `uid` is drawn fresh and never reused, so a provider that is replaced cannot be mistaken for the one it replaced, even when the two provide equal values. Since `notify` (Section 5.1.2) recomputes the target on every coeffect change, a fiber reloads precisely when one of its declared keys comes to be provided by a different fiber. A provider that overwrites its own binding in place is therefore not observed; a component that wants its replacement to propagate withdraws the binding and installs it afresh.

The algorithm operates at two complementary levels. At the *transition* level, `reload` and `unload` check the target at completion, enabling inertial chaining across transitions. At the *iteration* level within each transition, the effect execution (Algorithm 1) checks the target at each iteration boundary, enabling partial rollback within a single transition. These two mechanisms correspond to the inter-transition chaining of Section 4.3.3 and the intra-transition staleness check that Theorem 64 rests on.

Three lines carry the coeffect ordering of Theorem 63, and where each of them sits is what makes the ordering hold. `reload` commits the resolved view at Line 14 and `unload` discards it only after every inverse has run, so a fiber reads the same bindings for as long as it is loaded, its own teardown included. `refresh` marks the fiber `UNLOADING` at Line 10 before the transition task is created, which is the L-Leave step: the fiber stops providing, and the dependents recompute against that before any of its inverses is scheduled. `unload` then waits at Line 25 for each notified dependent to reach `INACTIVE`, which is the guard on L-Unload; `notify` admits a dependent only when its declared key resolves to the same realm symbol as the provider's, which is the runtime form of the guard's demand that the dependent see the key *from this fiber* rather than merely declare it. The wait sits ahead of the whole recovery rather than inside one of the inverses being waited on, since `fiber.dispose` initiates a fiber's effects concurrently and a wait placed within one of them would leave the rest unordered. Termination follows Theorem 66: a fiber only ever waits on dependents that have already stopped being satisfiable, and a dependent that is itself a provider waits the same way for its own, so the provider graph is traversed on demand rather than analyzed in advance.

5.1.4. Context Access

The coeffect operations of Section 5.1.2 form a *reflective* API: a coeffect is written with `ctx.set(key, value)` and read with `ctx.get(key)`, both keyed by name. Cordis layers a second, more native way to extend and consume the context on top of this reflective API: *property access*. A component can access a coeffect as the property `ctx[key]`, as if it were native structure of the context, rather than through a method call. In TypeScript, Cordis realizes this with a `Proxy` whose `get` trap mediates every property access. Algorithm 6 shows how a context resolves such an access to a coeffect, atop the primitive `get` of Section 5.1.2.

Algorithm 6 Proxy-mediated context access

```
1  function resolve(ctx, key)
2      fiber ← ctx.fiber
3      repeat
4          if key ∈ fiber.committed then return fiber.committed[key]
5          if key ∈ fiber.inject then throw INACTIVE_ACCESS
6          if fiber = root then throw UNDECLARED_ACCESS
7          fiber ← fiber.parent.fiber
```

Algorithm 6 walks the fiber chain upward from the accessing context: at the first fiber whose committed view binds `key`, the access is authorized and that binding is returned; if the walk reaches a fiber that declares `key` without having committed it, the fiber is not loaded and the access fails; and if it reaches the root without any declaration, the access is rejected as undeclared. This is where the proxy differs from the bare `ctx.get`: `ctx.get(key)` is a lookup against the store that returns the bound value or nothing and never fails, whereas the proxy resolves against the accessing fiber's own view and enforces the coeffect specification d at the point of use. Reading the view rather than the store is also what Theorem 63 rests on, since it is what keeps a dependency readable to a component whose teardown was triggered by that dependency going away.

This rejection is a *runtime* check performed at the point of access. Because a component's coeffect specification d is declared statically, the same violation is in principle detectable at compile time, by resolving each `ctx[key]` against the declared d before execution; Section 6.4 discusses how a host language's type-level dependency declarations and compile-time metaprogramming can carry out exactly this mediation.

5.2. Component Loader

The core library equips *component developers* with imperative primitives for dynamic composition, such as `ctx.effect`, `ctx.use`, and `ctx.set`. A separate concern arises for *application orchestrators*, who assemble pre-existing components into a running system and adjust the composition over its lifetime. The *component loader* addresses this concern by introducing a declarative configuration layer: the orchestrator specifies the desired composition as a persistent data structure, and the loader translates changes to this specification into the corresponding imperative fiber operations.

5.2.1. Declarative Configuration

Section 4 decomposes a running system into fibers, each an instantiation of one component. Everything an instantiation needs can be declared, so an orchestrator can describe a whole system as a *declarative configuration*: a persistent record that the loader realizes as fibers and keeps in step with them.

Entries. A configuration consists of *entries*. Each entry specifies a fiber and manages it, and the binding runs in both directions: the loader responds to a change in an entry's fields by adjusting the fiber, and a component that revises its own configuration or disables itself has the change written back to its entry.

Definition 74. An *entry* declares a single fiber, recording:

- `id` — a stable identifier, used as the reconciliation key when its group's child list changes;
- `url` — the URL of the component module to instantiate;
- `isolate` — an isolation annotation applied to the entry's context;
- `intercept` — an interception annotation applied to the entry's context;
- `config` — the configuration bound into the component to form its effect function `apply`;
- `disabled` — whether the entry is administratively turned off.

An entry can serve as a faithful specification because what supports a fiber is exactly what an entry records. The support set of Definition 67 reads τ, π, d, and p and nothing else, and an entry gives all four: `disabled` gives τ, the entry's parent in the tree gives π, and `url` selects the component which declares d and p. The fields the support set leaves unread are the fiber's runtime state, which an instantiation does not need either, and Lemma 70 identifies the support set with the Active fibers of a quiescent state (Definition 49) as far as each component installs every key it declares (Definition 69).

These entries form a *configuration tree* that is the authoritative record of what the system loads. An entry may be a leaf mapping to a single fiber, or its component may in turn load further components, making the entry a branch node. Cordis provides components for such grouped and nested loading: `@cordisjs/group` takes a list of child entries as its configuration and loads them as a subgroup, and `@cordisjs/include` loads an external configuration file (YAML or JSON) and grafts its entries in as a nested subtree. Both are ordinary components resting on the registration primitive of Definition 47 (Algorithm 4), so a nested tree stays within the calculus and the results below hold of it.

Reconciliation. When an entry's record changes, the loader reconciles incrementally rather than tearing the fiber down and rebuilding it wholesale. Reconciling this way is sound for reasons the metatheory supplies.

- Theorem 73 makes the quiescent state a function of the final configuration alone: whatever instantiations and retirements the loader performs on the way, and in whatever order, the system quiesces where a load of the final configuration from scratch would have left it. Which components end up loaded is read off the declarations only as far as each of them installs every key it declares (Definition 69); a component that declares a key and installs it under some configurations alone is one the loader can still reconcile, but the set of loaded components then answers to those configurations as well.
- Theorem 66 proves that the system does quiesce, so a reconciliation is complete once its instantiations and retirements have been issued.
- Corollary 62 puts a departing fiber's contribution to the state at nothing, so rebuilding one entry withdraws what its fiber installed and leaves the fibers around it as they were.

- Theorem 63 lets the entries be instantiated together, with no load order for the orchestrator to arrange: a fiber whose declared keys are not yet provided waits at its L-Begin, and one whose provider leaves is deactivated ahead of it. A dependency therefore constrains when a fiber activates rather than when its module is fetched and evaluated, so the loader loads modules concurrently, where bringing up a large configuration spends its time.

On top of the fiber that an entry declares, the loader dispatches on which of the entry's fields changed and applies the least disruptive operation for each.

- `id`, `url` — rebuilds the entry, since its identity or its component has changed;
- `isolate` — reassigns the entry's realms (Algorithm 7);
- `intercept` — updated in place, as interception metadata is consulted at read time and needs no reload;
- `config` — handed to the component, which decides how to apply the new payload, typically by diffing it against the previous one and reloading only on a material change. In particular, an `@cordisjs/group` entry's `config` is its list of child entries, so it applies the update as a keyed diff over child `id`s, creating, removing, or updating each child; since updating a surviving child re-enters this same per-field dispatch, group reconciliation and entry update recurse together down the tree;
- `disabled` — unloads the fiber when set and reloads it when cleared.

Managed realms. Isolation in the core derives a child context overriding the realm table ρ at one key (Section 5.1.2), which suffices while the context tree stands still. An entry may be moved between groups at runtime, so the loader manages realms of its own, and the `isolate` field selects between two scoping rules per key. A value of `true` asks for a *local* realm, private to the entry and tagged by its `id`, which the entry carries with it wherever it moves; a string asks for a *global* realm shared by every entry naming that string, so moving such an entry changes which entries it shares a binding with rather than which realm it belongs to. A realm is discarded once no entry names it.

Reassigning an entry's realms turns on which keys changed realm, whether the entry is itself the provider at a changed key, and which dependents to notify. The middle question is the hard one, since a realm symbol may be shared by several fibers of which only one is the provider. The loader answers it with *delimiters*: one symbol δ_k per key, under which each context stores a tag of its own. A delimiter is written on a context and inherited by its descendants, so the entry's tag and the provider's agree exactly when the two were derived within one isolate scope for k, which is the case in which the binding at k is the entry's own and has to move with it.

Algorithm 7 Isolation realm reassignment

```
1  function patch_isolation(entry, ρ′)
2      ρ ← entry.ctx[@@isolate]
3      store ← entry.ctx[@@store]
4      Δ ← {k | ρ(k) ≠ ρ′(k)} ▷ keys whose realm changes
5      for k in Δ do
6          entry.ctx[δ_k] ← fresh tag
7          diff[k] ← (ρ(k), ρ′(k), entry.ctx[δ_k], store[ρ(k)].fiber.ctx[δ_k])
8      entry.ctx[@@isolate] ← ρ′
9      reload(entry.fiber)
10     for k in Δ do
```

```
        (s_1, s_2, d_1, d_2) ← diff[k]
        if d_1 = d_2 and store[s_1] and not store[s_2] then ▷ the binding is the entry's own
            store[s_2] ← store[s_1]
            delete store[s_1]
    function affected(fiber, k)
        (s_1, s_2, d_1, d_2) ← diff[k]
        return fiber.ctx[@@isolate][k] ∈ {s_1, s_2} and (fiber.ctx[δ_k] = d_1) ≠ (d_2 = d_1)
    notify(entry.ctx, Δ, affected) ▷ in place of the realm test of Algorithm 3
```

The test turns on one property of delimiters. The tag under δ_k is written on the entry's context and inherited by every context derived from it, and it is drawn afresh at each reassignment, so for a context γ'

$$\gamma'[\delta_k] = d_1 \quad \Longleftrightarrow \quad \gamma' \text{ is derived from the entry's context} \tag{65}$$

Write $\mathrm{own}(\gamma')$ for that condition, of which $d_2 = d_1$ is the instance at the provider. The reassignment moves the contexts satisfying own from s_1 to s_2 and leaves the others where they are, and by the loop above it moves the binding to s_2 exactly when the provider satisfies own. A dependent sees the binding while its own realm at k is the realm the binding sits in. Where own agrees on the dependent and the provider, both move or neither does, so the dependent sees the binding afterwards exactly when it saw it before. Where own separates them, one side moves and the other stays, so the dependent gains or loses the binding. The inequality is that separation, and the membership test drops the dependents resolving k in neither realm, which no part of the move reaches.

5.2.2. Hot Module Replacement

Hot module replacement (HMR) applies the revertible-effect pattern at the *module* level: when source files change, typically during development, the system replaces the affected modules in-place without restarting the process. Because a fiber already bounds all of its component's effects and coeffects, a module that is itself a component can be replaced through fiber operations alone: disposing the old fiber recovers everything the component installed, and a new fiber instantiated from the reloaded module reinstalls it. HMR therefore needs no developer-annotated acceptance boundaries, as opposed to Webpack [46] or Vite [47] HMR.

The `@cordisjs/hmr` component provides the HMR *engine*, which operates in three phases.

Phase 1: Module classification. The engine takes two inputs: the *stashed* set (file URLs whose contents have changed since the last reload) and the *externals* set (modules that cannot be hot-replaced and instead trigger a full restart). Writing `get_imports(url)` for the modules that `url` directly imports, it classifies the changes' dependency subgraph, marking each module `accepted` or `declined`:

Algorithm 8 Module classification

```
function classify(stashed, externals)
    accepted ← stashed
    declined ← externals
    pending ← ∅
    for url in stashed do
```

```
        pending ← pending ∪ (get_imports(url) \ (accepted ∪ declined))
    repeat
        progress ← false
        for url in pending do
            if get_imports(url) ∩ accepted ≠ ∅ then
                accepted ← accepted ∪ {url}
                pending ← pending \ {url}
                progress ← true
            else if get_imports(url) ⊆ declined then
                declined ← declined ∪ {url}
                pending ← pending \ {url}
                progress ← true
            else
                pending ← pending ∪ (get_imports(url) \ (accepted ∪ declined))
    until not progress
    declined ← declined ∪ pending
    return (accepted, declined)
```

Seeded with the imports of the stashed files, the fixed point accepts a module once one of its imports is accepted and declines one once all of its imports are declined; any module left undecided, caught in an import cycle, defaults to declined.

Phase 2: Stale-entry detection. Using `accepted` and `declined`, the engine then filters the component entries down to the *stale* ones, whose dependency tree reaches a changed module. It walks each entry's tree with `get_dependencies`, which collects the transitive imports of a module while respecting `declined` as a boundary:

Algorithm 9 Stale-entry detection

```
function get_dependencies(root, declined)
    deps ← ∅
    function traverse(url)
        if url ∈ deps or url ∈ declined then return
        deps ← deps ∪ {url}
        for child in get_imports(url) do traverse(child)
    traverse(root)
    return deps
function detect(entries, accepted, declined)
    stale_entries ← ∅
    for entry in entries do
        tree ← get_dependencies(entry.url, declined)
        if tree ∩ accepted ≠ ∅ then
            accepted ← accepted ∪ tree
            stale_entries ← stale_entries ∪ {entry}
    return stale_entries
```

An entry is stale exactly when its tree intersects `accepted`; that tree is then folded into `accepted`, so every stale module along it is invalidated in the next phase.

Phase 3: Transactional reload. Finally, the engine reloads the stale entries. It invalidates the accepted modules' caches[3], backing up each removed module to enable rollback, then re-imports each stale entry's component module by its `url` and swaps in a fresh fiber:

Algorithm 10 Transactional module reload

```
function reload(ctx, accepted, stale_entries)
    backup ← invalidate_caches(accepted)
    try
        for entry in stale_entries do
            entry.fiber.dispose()
            entry.fiber ← ctx.use(import(entry.url), entry.config)
    catch error
        restore_caches(backup)
        for entry in stale_entries do
            entry.fiber.dispose()
            entry.fiber ← ctx.use(backup[entry.url], entry.config)
        throw error
```

The transactional guarantee ensures that the system never enters a half-reloaded state: if any module fails to import (e.g., due to a syntax error), the caches are restored and every stale entry is rebuilt from `backup[entry.url]`, the previous component whose cache was just restored, undoing the swaps already made.

5.3. Case Study: Koishi

Koishi is an open-source chatbot application framework built on Cordis[4]. Over four years of development, it has accumulated over 4000 community-contributed plugins[5], ranging from instant-messaging (IM) adapters and database drivers to administrative consoles and end-user features. Its scale and diversity make it a representative validation of Cordis's dynamic composability in a production setting.

Expressiveness and generality of the meta-framework. Koishi runs as a server-side bot whose every feature is realized as a plugin over the context primitives of Section 5.1; Koishi itself contributes only the chatbot-domain vocabulary. The same model reappears in a wholly different runtime: Koishi's web console is a second, independent Cordis application whose plugins compose the primitives of the browser and its user interface rather than those of the server. The disparate settings above establish two properties of the model of Section 3. (1) It is expressive: its primitives suffice to carry a complete production system, the host framework supplying only domain vocabulary. (2) It is general: it fixes how effects and coeffects compose while leaving their meaning to each application, and so presupposes neither a particular domain nor a particular runtime.

Temporal composability without cognitive overhead. The plugin systems surveyed in Section 1.2.1 cannot unload an individual extension's effects without restarting the extension

[3]On Node.js, this means clearing the caches of both the ES module and CommonJS module systems, since a module imported through the ES loader can appear in both.

[4]Koishi currently uses Cordis v3. This paper presents Cordis v4, which refines the effect and coeffect semantics and redesigns the loader; the core compositional model is shared across both versions.

[5]Koishi uses the term *plugin* for the concept this paper formalizes as *component*.

host. Koishi routinely performs this operation: an orchestrator disables a plugin from the console and its effects are withdrawn in place; during development, the HMR engine re-applies edited plugins on save while preserving cache state and live connections elsewhere in the system. Cordis makes such removal not merely possible but effortless for the plugin author. Because effects performed through the context are tracked and their inverses composed automatically (Section 3.1), even an inexperienced author obtains ordered cleanup for a plugin's context-mediated effects without writing an uninstall path. This achieves the locality of concern whose absence Section 1.2.1 identifies: correctness that would otherwise rest on each author's diligence is instead discharged once, by the abstraction.

Spatial composability across an open ecosystem. In contrast to the plugin systems of Section 1.2.1, where inter-plugin dependencies are largely absent, Koishi's ecosystem exhibits a genuine dependency topology: IM adapters provide access to each messaging platform, database drivers provide persistent storage, and functional plugins declare these as coeffects and access them. Reconfiguring a provider at runtime, such as switching the storage backend or reconnecting an adapter, reactivates only the dependents whose resolved dependency changed (Section 3.2); a plugin whose dependency is unavailable stays inactive until it appears, without erroring. What the case study substantiates is that this composition holds across independently authored code: a plugin and its dependencies are typically written by different authors who coordinate on nothing beyond the coeffect that connects them, so reactive coeffects keep the assembly consistent across an open ecosystem of independent contributors.

Threats to validity. The evidence here is drawn from a single ecosystem in a single host language, so it cannot separate the merits of the paradigm from those of its TypeScript realization or of Koishi's particular domain, and it is observational rather than a controlled comparison against an alternative architecture. What the case study establishes is thus an existence-and-adoption result rather than a quantitative one; measuring the abstraction's overhead and its effect on developer productivity against a baseline remains future work.

6. Discussion

The formal model and implementation presented in the preceding sections introduce a programming paradigm for dynamic composability. This section examines how the paradigm extends to broader engineering concerns, and discusses the design tensions and open problems.

6.1. System Boundary

Every effect in Section 3.1 carries an inverse, and what that inverse amounts to is settled by the *system boundary*. The boundary divides the environment a system runs against into two parts. (1) A location lies *inside* when the system is able to modify it exclusively and to restore the state before that modification, so an operation on it is tracked in Γ and can be recovered later. (2) A location lies *outside* when either ability fails, so an operation on it acts as id_Γ and is therefore neither tracked nor recovered. This section develops the properties of this boundary and their consequences for recovery.

Boundaries from coeffects. A coeffect moves the boundary by reifying an external location: it confines every access to that location to a set of operations it provides, each of which it can supply an inverse for, so operations that acted as id_Γ come to be tracked in Γ and recovered. The

boundary is therefore drawn per location rather than per medium, since both aforementioned abilities are properties of a location, and reification changes how a location is accessed while leaving its medium as it was. For example, a memory region lies inside when the system alone writes it, and outside when other processes write it too; a file lies inside when only the system can reach it, as with a scratch file under a private path, and outside when it is a path other programs read or write. Moving the boundary is itself a trade-off, between whether the environment provides revertible semantics for a location and what supplying those semantics costs on every access. We take up the co-design this suggests in Section 6.7.

Acquisition and emission. An operation that reaches outside the boundary generally proceeds in two stages. (1) In the *acquisition* stage, the operation obtains access and installs a record inside the boundary: `open` installs a descriptor that `close` removes, `malloc` reserves a block that `free` releases, `fork` starts a child process that `kill` terminates. The record itself is part of the coeffect that reifies the location, e.g. an entry in a map it keeps, and installing that entry is a revertible effect. That record is at the same time the *channel* along which data can leave. (2) In the *emission* stage, the operation pushes data through that channel, as with the bytes a `write` hands to the file or the datagram a `send` puts on the wire, and the push acts as id_Γ, leaving the data where other parties may read and write it. The two stages therefore fall on opposite sides of the boundary: the acquisition stays inside it, whereas the emission crosses to the outside.

Withholding and compensation. A system that must nonetheless recover from an emission has two approaches available. One is to withhold an emission until the state that produced it is certain to persist, which is the *output commit problem* of rollback-recovery [48]. The other is *compensation* [49]: an action that restores the state up to an equivalence the application supplies, coarser than the $\simeq$ of Definition 33, as in deleting a file that was created or refunding a charge that was made. Such actions compose in the same LIFO order as inverses do, so the composition of Section 3.1 transfers to them. The metatheory does not: the commutation of Definition 60 is proved against $\simeq$ and has to be re-established against the coarser one.

6.2. Service Multiplexing

Dynamic component platforms such as OSGi [50] organize composition around *services*: units of functionality that a provider publishes under an interface and a consumer binds to. The Cordis coeffect model echoes this notion, with a service corresponding to the interface behind a key. Components that `provide` a service are its *providers*, and components that `inject` a service are its *consumers*. A single service may be implemented by multiple providers, and this multiplicity can be realized in two forms. (1) *Exclusive binding*: several implementations share one interface but at most one is bound at a time; the orchestrator selects which implementation is bound, and switching between them requires unloading one provider and loading another, momentarily perturbing every consumer's dependency. (2) *Service broker*: a central service that acts as the entrypoint for the interface is injected by both the backing providers and the consumers, so that multiple providers coexist and the broker dispatches each request among them. Compared to exclusive binding, the broker absorbs this perturbation: updating a backing provider leaves the broker in place, so consumers see no change to their dependency and no reload is triggered.

The service broker underlies three capabilities: load balancing, rolling updates, and cross-process invocation.

Load balancing. When several providers coexist, the broker distributes requests among them according to a configurable policy (e.g., round-robin, least-loaded, latency-weighted) or

an explicit target named by the consumer. Because providers are ordinary components, they can be added or removed to scale capacity up or down; each provider registers with the broker through a revertible effect, so unloading it reverts the registration and drops it from the broker's routing set automatically.

Rolling updates. Upgrading a service implementation at runtime reduces to a controlled provider transition [51, 52]. To carry out the transition, the new provider is loaded as an additional fiber and registers with the broker; once it becomes `ACTIVE`, traffic is gradually shifted from the old providers to the new one (e.g., by adjusting selection weights), and the old providers are unloaded once they no longer carry in-flight requests. This provider transition turns what is traditionally an infrastructure-level operation (e.g., container orchestration, blue-green deployment) into an application-level composition pattern.

Cross-process invocation. The service broker can also be applied across process boundaries [53]. Each process hosts its own Cordis context with local providers; a coordinating component links them, treating each as a remote provider. Cross-process service access is mediated by an RPC mechanism that preserves the interface, making the distribution transparent to consumers. One caveat is that a cross-process call incurs latency and may fail mid-flight, so exposing it synchronously would block the caller. An interface intended to be exposed across processes must therefore be designed against an asynchronous contract.

6.3. Access Control and Sandboxing

Given an application assembled from independent components, securing the application calls for two complementary mechanisms: (1) constraining what dependencies a component may access, and (2) sandboxing untrusted code from the host environment. Cordis supports the first through dependency declarations and interception; the second requires an external sandbox.

Capability-based access control. The dependency access mechanism (Section 5.1.4) already constitutes a form of access control over proxy-mediated properties: a component can only access dependencies it has declared; an undeclared access raises an error. This is structurally similar to capability-based security [54–56], where authority is conferred by possession of a reference rather than by ambient authority. The `inject` declaration acts as a capability request, and the context proxy acts as a capability mediator. Since these requests are declared statically, the complete set of proxy-mediated capabilities a component requires is known before it runs, letting the orchestrator review and approve them at load time rather than discovering accesses as they happen.

This mediation generalizes to fine-grained policy through the interception mechanism. Access-control metadata can be carried by contexts or declared by components (Definition 30), and the provider consults it when the dependency is invoked to decide whether a request is permitted. For example, a filesystem dependency may carry metadata declaring which paths a component may read or write, and the provider checks each call against the metadata. Because this interception lives on the context rather than in either party's code, an orchestrator can adjust it to constrain any component's access to a dependency without modifying the provider, e.g., granting read-only database access to a community component whereas a core component retains full access. Moreover, since interception affects only how a dependency is invoked, not whether it is satisfied, it can be installed, reconfigured, or removed at runtime without triggering any reload or perturbing the dependency graph.

Sandboxing untrusted components. When a component's code cannot be trusted, language-level access control is insufficient, since a malicious component with access to the host runtime can reach the underlying objects directly, rendering such checks moot. Sandboxing requires an execution boundary beyond the reach of language-level means, such as software fault isolation [57], a separate language runtime, a sandboxed process, or a virtualized container [58]. Whatever the mechanism, the untrusted component runs in its own sandboxed context and reaches host-provided dependencies through a bridge, generalizing the cross-process invocation of Section 6.2: the same transparency argument renders this bridged access indistinguishable from local injection. On the host side, the bridge is an ordinary fiber whose capabilities can be attenuated by the access control described above.

6.4. Language Independence and Selection

Although Cordis is implemented in TypeScript, the context paradigm is language-agnostic: spatiotemporal composability is defined only by its two composability dimensions, and thus can be realized in any language that meets certain requirements along both. We analyze these requirements along each dimension in turn.

Temporal composability. At its most basic, temporal composability requires *closures*: a revertible effect pairs an action with an inverse, and that inverse must be captured as a value, along with the state it restores, so it can be replayed on teardown. Beyond this, a component's code and the side effects of loading it must be introducible and retractable at runtime.

How a language meets this second requirement depends on its execution model. In managed runtimes, this takes the form of a programmatic module registry, where a loaded module can be evicted from the registry and garbage-collected once unreferenced; Node.js, for instance, exposes such a registry.[6] Native code exposes no module registry, so introduction and retraction take the form of explicit dynamic linking and unlinking (e.g., `dlopen/dlclose` on Unix, `LoadLibrary/FreeLibrary` on Windows) [59], i.e., loading object code into a running process and later detaching it. WebAssembly takes one path or the other depending on its embedder: a module instance is reclaimed by the host's collector under a managed embedder (e.g., a JavaScript host), or released when a native embedder drops it (e.g., Wasmtime). Across these mechanisms, the revertible effects model treats loading as an effect on the context, with inverses that undo the registration of symbols, types, or handlers the module introduced.

Spatial composability. Spatial composability requires a mechanism for components to declare their dependencies and for the runtime to provide and inject these dependencies. This reduces to a dependency injection (DI) problem [38], which manifests at two levels that differ across languages: how dependencies are *typed* and how their access is *mediated*.

At the type level, the language should provide a way for developers to express well-typed dependency access. A consumer obtains a coeffect by reading its key from the context, so the context type (Section 3.2.1) must record each key's coeffect. Typeclasses (Haskell) [60] and traits (Rust) [61] achieve this by letting a provider extend the context type from its own module through an `instance` or `impl` [62]. TypeScript's module augmentation [63] likewise lets a provider module merge declarations into the context type.

At the runtime level, dependency access must be dynamically mediated: the coeffect behind a key may change as providers are loaded and unloaded, and may be resolved differently across

[6] CommonJS exposes the module cache via `require.cache`; ES modules provide no public eviction API, though modules can still be managed through engine-internal interfaces.

contexts. The language therefore needs a way to interpose on access transparently, leaving the consumer's code unchanged, e.g., via JavaScript's `Proxy` object [64] or Python's descriptor protocol (`__get__`) [65]. Absent such a primitive, runtime reflection [66, 67] can mediate access dynamically, at the cost of type safety and developer experience.

Across both levels, metaprogramming facilities supply the typing and the mediation together. Annotations [68] and decorators attach metadata to a declaration, which a processor expands into the accessor that mediates access; compile-time metaprogramming (e.g., Rust procedural macros, Scala macros [69], Zig comptime) emits, for each dependency, a typed declaration together with such an accessor, dispensing with a general-purpose interception primitive.

6.5. Mutual Dependencies and Component Granularity

In the reactive coeffect model, a dependency cycle simply leaves the involved components permanently inactive: given two components A and B, if A requires a key provided by B and B a key provided by A, neither's satisfaction predicate can ever become true. Unlike deadlock in concurrent systems, which depends on the schedule and must be detected as it happens, this condition is predictable from the dependency declarations alone, so a runtime can report it when components are loaded.

In practice, most apparently mutual dependencies can be decomposed into finer-grained components that eliminate the cycle. Consider two components: a server (providing a network interface) and an access controller (enforcing authorization policies). The two components interact bidirectionally: the access controller mediates requests arriving at the server, and the server exposes an endpoint for modifying access-control policies. A monolithic design would make each component depend on the other. However, the two interaction directions are logically independent concerns. Decomposing them yields four components: server-core, access-control-core, request-mediation (depending on both cores to apply access control to incoming requests), and policy-management (depending on both cores to expose policy modification via the server). Through this approach, the cycle is eliminated because neither core depends on the other; only the integration components depend on both.

This decomposition is always possible in principle, since every bidirectional interaction can be factored into independent unidirectional bindings, but it increases the number of components: in the general case, given n mutually interacting components, the number of integration components can grow quadratically with n, since each pair of interacting components may require a distinct component for each direction of interaction. This does not affect correctness or runtime performance (components are lightweight), and finer granularity can be beneficial: users gain the ability to load only the specific integration bindings they need, effectively increasing the system's composability. However, it may affect developer experience: more components require more configuration, more naming, and more cognitive overhead in understanding the dependency graph.

Mitigating this granularity cost is an engineering concern rather than a theoretical one. Practical strategies include package bundling (i.e., grouping related fine-grained components into a single installable unit), convention-based wiring (i.e., automatically connecting components whose names or types match a pattern), and scaffold tooling (i.e., generating boilerplate integration components from declarative specifications). These strategies preserve the formal guarantees of the acyclic model while reducing the authoring burden to something closer to the monolithic case.

6.6. Dependency Typing and Versioning

In the formal model, a dependency link is established purely by key identity: a component providing key k satisfies any component declaring k in its dependency set. The type family $\mathcal{V}_k$ ensures type-level agreement within a single compilation unit, but this guarantee breaks down when components are developed and built independently, which is a common scenario in component ecosystems. This breakage leads to two distinct problems.

Interface drift. A provider may modify the interface associated with k (adding fields, changing method signatures, altering behavioral contracts) between versions, while a consumer compiled against an earlier interface continues to declare the same key k. The dependency is satisfied at the coeffect level ($k \in \mathrm{dom}(\sigma)$), yet the runtime value no longer conforms to the consumer's expectations, leading to type errors, method-not-found failures, or silent behavioral divergence [70].

Key collision. Two independently developed providers may use the same key name k to denote entirely unrelated interfaces. Since key identity alone establishes the link, a consumer expecting one provider's interface will accept the other's value without any compatibility check. Unlike interface drift, where the provider and consumer at least share a common lineage, key collision involves no relationship whatsoever between the expected and actual types, making the resulting failures unpredictable and difficult to diagnose.

Both problems point to the same gap: the coeffect model provides only *nominal* linking (by key name) but no *versioned* or *structural* linking (by interface compatibility) [71]. We discuss three approaches to the gap, from most infrastructure-coupled to most language-agnostic.

Key namespacing. Extending the key space from K to $K \times P$, where P identifies the interface-defining package, eliminates key collision by construction: independently developed interfaces with the same local name occupy distinct keys. This is the most direct solution but also the most coupled: it embeds the package namespace into the formal model itself, making the system dependent on an external package registry for key identity.

Peer dependencies. A lighter coupling is to declare version constraints through the host-language package manager [72]. This is the approach Cordis currently adopts. Component dependencies are semantically *peer dependencies*: a component does not bundle its dependencies internally but expects the runtime context to supply them. Package managers with peer dependency support (e.g., npm) can enforce version compatibility: if the version of the package providing a key falls outside a consumer's declared peer range, the incompatibility is caught at install time rather than surfacing as a runtime failure. However, this approach has two limitations: (1) it depends on providers faithfully adhering to semantic versioning, which is an unenforceable convention; (2) package managers typically resolve each dependency to a single version, which prevents loading components from multiple versions of the same package within one application.

Structural compatibility. A fully language-agnostic approach would replace the membership check $k \in \mathrm{dom}(\sigma)$ with a compatibility predicate that verifies the provider's actual interface structurally subsumes the consumer's expectation. This is analogous to structural subtyping [73]: a provider satisfies a consumer if the provided interface is a subtype of the required interface. The challenge lies in defining this predicate language-agnostically: structural compatibility is straightforward for record types (width subtyping) but becomes complex for behavioral contracts (e.g., pre/postconditions [74], effect specifications [22]), and undecidable once parametric polymorphism introduces bounded quantification [75].

These three approaches address different aspects of the problem. Designing a unified dependency model that combines these approaches while preserving the dynamic composition guarantees of the coeffect model remains an open problem.

6.7. Co-Design with Languages and Operating Systems

Section 6.4 identifies the minimum a host language must supply for the context paradigm. This section takes up the converse question, what a language or operating system co-designed with the paradigm can offer beyond that minimum.

Co-design with languages. A language designed around the context paradigm can improve on a library in two respects: the semantics it gives to contexts, and the primitives it gives to effects and coeffects.

Such a language can make the context implicit again while preserving the context semantics of Section 3.3. An imperative language already runs every statement against an implicit context, and that single context neither tracks effects nor resolves coeffects. The context paradigm instead distinguishes multiple contexts, where an operation either modifies the context it runs against or derives another from it (Definition 27). An in-place realization modifies the ambient context, just as an imperative language does. A derived realization instead introduces a separate context, for which the language must provide a construct. Making the context implicit brings both an ergonomic and a safety benefit. (1) In a library realization, every function involving effects or coeffects takes the context as an ordinary argument or a receiver, as in Section 5.1. Where the language supplies the context implicitly, functions no longer need to take it. (2) Every context carries its own lifecycle state and committed view (Section 4.1). A library realization passes a context as an ordinary variable, so a component may reach another component's context by mistake, through a closure or a global variable. An effect it installs there then leaks out of its own lifecycle, and a coeffect it reads there escapes its dependency specification. Making the context implicit closes both.

Such a language can also make effects and coeffects known to its compiler. (1) For effects, an effect iterator (Definition 51) allocates a closure at every step to hold the inverse together with the state it restores. With syntax for performing an effect, a compiler can emit a single state machine for the whole iteration and hold those inverses in its frame. (2) For coeffects, the coeffect specification can be admitted into the type system, with two benefits. First, a dependency cycle is reported at compile time instead of being left to the runtime (Section 6.5). Second, a dependency can be compared by the structure of its type rather than by key identity alone, as row types do [28], which is type-level support for the structural compatibility of Section 6.6.

Co-design with operating systems. Section 1.2.3 observes a coarse-grained substitute for dynamic composability, where the operating system supplies temporal composability at the granularity of a process, and the container orchestrator above it supplies spatial composability at the granularity of a service. An operating system co-designed with the paradigm would support fine-grained composition, by making the coeffect specification a component declares the whole of what it can reach, and by providing its own resources as coeffects.

Such an operating system can supply the sandbox that Section 6.3 defers to a mechanism outside the language. It does so by bounding a component to the dependencies it declares, supplying them when the component is loaded and leaving nothing else reachable from within it, as a WebAssembly module receives its imports from its embedder at instantiation [76]. It

can also provide the coeffect isolation and interception of Section 3.2.3 as abilities of its own, binding a key differently for each component and mediating the accesses it supplies.

Such an operating system can also provide its own resources as coeffects. A resource lying outside the boundary is made revertible where the runtime records each acquisition against the component that made it (Section 6.1), and every runtime keeps a record of its own. An operating system that provides the resource as a coeffect keeps that record once, since it is the party that hands the resource out and can attribute it to the component that asked. Memory and file descriptors are the immediate candidates, and tracking them for the sake of recovery has been done at the kernel interface [77, 78]. Furthermore, an operating system can make revertible some of the operations Section 6.1 can only withhold or compensate for. A system that performs a write to persistent storage transactionally can roll it back [79], and one built on copy-on-write or immutable storage reaches an earlier state by moving a pointer [80, 81].

7. Related Work

Dynamic composability intersects several established research areas. We survey the most relevant lines of work and distinguish our contribution from each of them.

7.1. Effect and Coeffect Systems

Section 2 reviewed effects and coeffects as the theoretical pillars underlying our work. We first situate the monadic effect systems now common in industrial practice, then survey three research lines that extend effects and coeffects in directions relevant to Cordis: recasting algebraic effects as capabilities, giving effects a reversible semantics, and unifying effects and coeffects under a single graded discipline.

Monadic effect systems. One family of libraries encodes effects in the type systems of existing general-purpose languages, representing them as monadic values that a runtime executes. ZIO in Scala [82] models a computation as `ZIO[R,E,A]` and Effect-TS in TypeScript [83] as `Effect<A,E,R>`, a generic type whose parameters describe its result, its typed errors, and the services its context must supply; the fp-ts library [84] encodes the same error and requirement channels through Reader-based monad transformers. Two traits separate these systems from Cordis. First, the tracking is bought with a monadic embedding: a program obtains it only by being written inside the effect type, whereas Cordis tracks effects as an overlay over ordinary host code. Second, a requirement is discharged by interpretation, an installed service that supplies its operations, and when that service is withdrawn what its operations performed remains in place; Cordis instead pairs each effect with an inverse and re-resolves requirements as providers come and go (Section 3.1, Section 3.2).

Algebraic effects as capabilities. Algebraic effects (Section 2.1) make effect operations visible to the type system. The extension closest to our work is Brachthäuser et al.'s Effekt language, which reinterprets effect types as *capabilities* [85, 86]: an effect type expresses what a computation requires from its context rather than what side effects it may produce. This perspective, like ours, treats the context as a mediator of capabilities. Cordis and Effekt differ in two respects. (1) In purpose, algebraic effects make effects visible to enable *modular interpretation*, giving one operation many handler semantics, whereas Cordis makes them visible to enable *tracking and reversion*, pairing every context transformation with an inverse. (2) In setting, Effekt disciplines effects statically at the type level, defaulting to scope-based reasoning in which capabilities are

second-class and confined to their lexical scope, and recovering first-class use through boxing, which lifts that restriction by tracking captured capabilities in types; Cordis instead disciplines effects at runtime, aiming at complete resource recovery on component removal; Section 6.7 takes up what a language that made the context second class in this sense would offer.

Reversible effect semantics. A parallel line gives effects a reversible semantics rather than an interpretive one. Heunen et al. [87] model side effects in a reversible setting by adapting Hughes' arrows to *dagger arrows* and *inverse arrows*, capturing effects such as serialization and mutable store whose operations admit inverses. This is the formal account closest to our revertible effects: both pair each effect with the means to undo it rather than discharging it through a handler. The two differ in where reversibility resides, and in how much of it they demand. Heunen et al. work in a denotational, categorical setting where reversibility is a global property, guaranteed by construction since every computation is invertible, and the inverse is two-sided and recovered from the categorical structure. Cordis tracks inverses at runtime and requires less of them: not that the whole computation be reversible, but that each atomic effect admit a one-sided inverse, supplied by the caller at the point of application rather than derived, from which the inverse of any composite follows by composition (Section 3.1).

Graded types as unified effects and coeffects. Orchard et al. [88] proposed *graded modal types* as an umbrella notion encompassing both effect reasoning (via graded monads) and coeffect reasoning (via graded comonads), realized in the Granule language, demonstrating that a single type system can track both what a computation does and what it needs; more recent work extends coeffects to imperative Java-like languages [89, 90] and to call-by-push-value [91]. All of these operate at the type level: effects and coeffects are static annotations checked at compile time over lexically fixed scopes. Our contribution is orthogonal to this analysis: we lift the same two notions to runtime mechanisms, which lets Cordis handle dynamic composition. Temporal retraction and spatial dependency are re-resolved as the set of loaded components evolves, instead of being settled once over a fixed program text.

7.2. Programming Paradigms

Section 3.3.3 established the context paradigm as a discipline that mediates effects and coeffects through an explicit context. Two established paradigms warrant explicit comparison: one shares our terminology, the other our treatment of crosscutting concerns.

Context-oriented programming. COP [92, 93] equips a language with *layers*—partial method and class definitions that are activated and deactivated at runtime according to the execution context, so that behavior adapts without the base code naming its context dependencies [94]. COP and Cordis coincide in treating context as a first-class, runtime-mutable entity and in activating and deactivating behavior dynamically, but the resemblance is nominal. In COP, "context" denotes the ambient execution situation (e.g., location, user, mode), and activation changes method dispatch within a dynamically scoped extent; a layer neither tracks the side effects it induces nor reverts them, and activation is not governed by dependency satisfaction. In Cordis, the context is the Γ_∞ entity mediating effects and coeffects: activation runs a component's revertible effects and is driven by reactive coeffect satisfaction (Section 3.2), and deactivation reverts them in full. COP varies what behavior runs; Cordis composes and reverts what effects and dependencies a component installs. Their difference is one of trade-off. COP folds activation into the host language's method dispatch, gaining dynamically-scoped layer extents at the cost of language specificity, whereas Cordis, as a language-agnostic overlay, resolves activation reactively over a shared context. Cordis can thus express as a coeffect only

COP's global, value-driven fragment: context-dependent selection among implementations, but not dynamically-scoped activation.

Aspect-oriented programming. AOP [95, 96] modularizes a crosscutting concern into an *aspect*: a *pointcut* that quantifies over *join points* selected in the base program, and *advice* woven in at each. Cordis addresses the same problem of contextual behavior that would otherwise scatter across components, but its analogue of an aspect is a coeffect: a shared point of mediation many components declare a dependence on, so that crosscutting behavior can be reshaped there without editing any of them. The two paradigms then differ on two axes. (1) *Declaration versus obliviousness*: an AOP pointcut is oblivious and quantified, matching arbitrary join points whose code is unaware it is advised, whereas Cordis confines crosscutting to the coeffects each component declares, so its reach is exactly that declared surface. This yields determinacy and traceability: an application orchestrator can inspect and govern what cross-cuts a component at the configuration layer, without reading or analyzing its source, whereas an AOP concern is legible only through the aspects that quantify over it. (2) *Lifecycle integration*: a crosscutting change in Cordis is carried by a component's effects, reverted when the component unloads and propagated reactively to its dependents, so it is one move within the dynamic composition model; dynamic-AOP systems [97, 98] can also weave and unweave at runtime, but as a standalone operation, neither bound to a component's lifecycle nor triggering re-resolution among the advised code.

7.3. Temporal Composability

Temporal composability concerns replacing or removing a component in a running program while recovering the effects it installed. Prior approaches divide by how they treat a departing component's state and effects: carrying state forward to a successor version, recovering effects through developer-authored cleanup, reversing effects automatically within a scope fixed in advance, or reclaiming resources from a record the runtime accumulates by interposing on an interface.

Stateful forward migration. A broad family of systems replaces components in a running program without downtime by carrying their state forward across versions. All observe the same timing discipline: a component may be swapped only once it reaches a safe, interaction-free point. Kramer and Magee established this criterion as *quiescence* [51], which Vandewoude et al. later relaxed to the less disruptive *tranquility* [52]; our rolling-update pattern (Section 6.2) enforces it by draining in-flight requests before unloading a provider. Dynamic software updating (DSU) then migrates state forward through hand-written transformation functions: Hicks et al.'s general-purpose DSU for C [99], Stoyle et al.'s type-safe update points via con-freeness analysis [100], and Hayden et al.'s Kitsune [101] all map old-version data to new-version representations, inheriting heap objects, open files, and connections in place while re-initializing whatever is left unmigrated. The same discipline extends to persistent state: Overeem et al. [102] convert a running event store's data between schema versions through hand-written upgrade operations while keeping the system available. Erlang/OTP [15] takes the same stance at the process level, migrating state through `code_change/3` and recovering from faults by restarting supervised processes rather than reverting their effects; JavaScript's Hot Module Replacement (e.g., webpack [46], Vite [47]) does the same at the module level, handing state forward through the `module.hot` or `import.meta.hot` API across a reload. Compared with Cordis's module replacement (Section 5.2), these approaches migrate in-memory state more gracefully: Cordis reverts the old component's tracked effects and reapplies the new component's from a clean slate, so a component's own in-memory state does not survive a

reload unless placed in a longer-lived dependency, and layering DSU-style forward migration atop revertible effects is future work. Cordis's approach is nonetheless more general in two respects: it needs no hand-written migration functions of the kind DSU and HMR require, and it supports unloading a component entirely and recovering its resources, not merely updating one in place.

Developer-authored recovery. A second family recovers a component's effects through cleanup or compensation logic that the developer writes by hand. Plugin lifecycle conventions (e.g., OSGi [50], Eclipse's extension points, IntelliJ and VSCode) delegate cleanup to developer-written unload callbacks; the Command pattern [103] encapsulates an operation together with an `undo` method for undo/redo stacks; the saga model [49] structures a long-lived transaction as steps each paired with a compensating action; algebraic effect handlers can attach finalizers that run on teardown [104]; and event sourcing [105] retracts state by appending compensating events rather than executing an inverse at all. In all of them the inverse is an unenforced duty, decoupled from the operation, so that a forgotten one leaks resources silently (as documented empirically in Section 1.2.1). React's `useEffect` hook [106] comes closest to pairing an effect with its inverse structurally, returning a cleanup the runtime invokes before each re-execution and on unmount. Its shortfall is composability: a hook may be called only at the top level of a component or another hook, never inside a conditional, loop, or nested function, and its effect body accepts neither an async function nor an iterator. Effects thus cannot be assembled from other effects or interleaved with control flow, leaving nothing from which a composite inverse could be derived. Cordis effects carry no such restriction: they are ordinary operations that compose freely and may run asynchronously, and require a hand-written inverse only for each atomic effect, from which the inverse of any composite is derived by composition, so that assembling existing effects requires writing no inverses at all. This structural pairing of every effect with its inverse makes complete recovery an invariant of the system rather than a matter of developer discipline.

Statically scoped reversal. A third family reverses effects automatically, by construction, but confines reversal to a scope fixed in advance. Software transactional memory [107, 108], descended from hardware transactional memory [109], records a read/write log so that a group of memory operations either commits or aborts, rolling memory back to its pre-transaction state. Reversible computing, from Landauer and Bennett's thermodynamic analyses [110, 111] to reversible languages such as Janus [112], goes further and makes every step of a whole computation globally invertible. Reversible process calculi build backtracking into the semantics itself: RCCS [113] carries a memory alongside each process and admits a step to be taken back when the past it leads to is causally equivalent, and Phillips and Ulidowski [114] derive reversible operators for CCS, ACP, and CSP uniformly while preserving their forward operational semantics. Their causal-consistency criterion is the concurrent counterpart of the order Cordis's recovery follows, an accumulator applying a component's own inverses in last-in-first-out order and the guard of Section 4.3.1 deferring a provider's withdrawal until its consumers have deactivated (Theorem 63). The reach, however, is fixed by the semantics, every action performed remaining undoable, whereas a Cordis component supplies an inverse for each atomic effect and its accumulator brings the context back to where its composition began. Linear types [115], RAII [4], and Rust's ownership system [61] tie a resource's release to a lexical region. Each fixes the scope and reach of reversal statically; Cordis, by contrast, fixes no such scope in advance: it reverts arbitrary context operations over a component's lifecycle, and treats lexical resource management as complementary, appropriate for local resources within a single component.

Interposed reclamation. A fourth family reclaims what a component acquired without the component itself supplying the inverses, by recording its acquisitions at an interface the runtime controls. Nooks [77] wraps every call crossing the boundary between the Linux kernel and its loadable extensions, so that the kernel objects an extension touches pass through an object tracker whose record tells the recovery manager what to release when the extension fails; shadow drivers [78] tap the same calls from the other side, recording the requests and configuration that determine a driver's state so that a restarted instance can be restored to it. Akeso [116] obtains the record by compiler instrumentation instead, dividing kernel execution into nestable recovery domains that log their state changes and cross-thread dependencies, and rolling a faulting request back together with every domain that depends on it. Reclamation thus follows from a record the runtime maintains rather than from cleanup the developer remembers to write, which makes this family the closest systems-level precedent for revertible effects. It differs from Cordis in vocabulary and in reach. The platform fixes what can be recorded, whether as release code per kernel object type, one shadow per driver class, or an inverse per instrumented allocator, so a component may hold only resources the platform already knows how to release; a Cordis component instead introduces effects of its own and supplies an inverse for each atomic one (Section 3.1). Reclamation is likewise bounded by a request that commits or a restart of the same extension, whereas Cordis reverts over a component's whole lifetime and propagates removal to its dependents, which release their own effects in turn (Section 3.2).

7.4. Spatial Composability

Spatial composability concerns how a component's dependencies on others are declared and bound. Prior mechanisms divide by how binding responds to change: wiring dependencies once at initialization, reacting to the availability of whole components, or propagating change at the granularity of individual values.

Initialization-time dependency wiring. Two established mechanisms wire components together at initialization time. Dependency injection frameworks [38] (e.g., Spring [117], Guice, Angular, Inversify) inject dependencies into components at initialization, and UI framework context (e.g., Vue.js's `provide/inject` and React's Context API) passes them along a component tree. Some support dynamic scoping (e.g., Spring's prototype/request scopes, Angular's hierarchical injectors), but neither re-resolves reactively: when a provider is replaced or removed at runtime, existing dependents are neither deactivated nor re-initialized, and none offers lifecycle management of the kind our component state machine provides. Cordis's reactive coeffects (Section 3.2) supply this: the notification mechanism triggers lifecycle transitions whenever the satisfaction predicate changes.

Availability-reactive component models. The closest precedent to our reactive coeffects reacts to service availability. OSGi's Declarative Services and iPOJO [118, 119] let components declare provided and required services, with the runtime automatically activating and deactivating them as services appear and disappear; iPOJO's Gravity project [119] explicitly targets autonomous runtime adaptation to changing service availability, and its provide/require model directly prefigures Cordis's `ctx.provide/ctx.get` pattern. R-OSGi [53] extends the same abstraction transparently to distributed settings via RPC, mapping network failures to service-withdrawal events, a pattern Section 6.2 discusses as an extension of the Cordis model. All these systems recover through a deactivation callback, which is limited in two ways. First, the callback is hand-written, so resource safety rests on developer discipline and a forgotten one leaks silently. Second, the callback is synchronous: should teardown require an asynchronous exchange with the departing dependency, the frameworks offer no protocol to await it, forcing a

blocking wait against a reference that may already be stale. Cordis's reactive coeffects close both gaps: deactivation reverts the dependents' accumulated effects, and its inertial Unloading state (Section 4.3.3) runs asynchronous teardown to completion before acting on further change.

Value-level reactivity. Functional reactive programming (FRP) [120] and its modern incarnations (e.g., signals [121, 122] in SolidJS, Vue's reactivity system, Angular Signals) propagate change at a *value-level* granularity: when a signal changes, derived computations are re-evaluated synchronously or under a scheduler [123]. Cordis's reactive coeffects act at a *component-level* granularity, adding asynchronous lifecycle semantics that value-level propagation does not model. The same granularity difference runs the other way for consistency: propagating in a turn, in an order the dependency graph fixes, lets FRP require that no derived computation read a mixture of updated and stale inputs, which is *glitch freedom* [124], whereas Cordis has no counterpart of a turn, orchestration actions arriving one at a time, and guarantees only that no single transition straddles two resolutions of its coeffects (Theorem 64). The two are complementary rather than competing: a Cordis coeffect can itself carry reactive values, and a component updates on only the parts it actually consumes, refining component-level reactivity into finer-grained reactive coeffects that span both levels.

8. Conclusion

We have presented a formal foundation for dynamic composability by lifting the classical concepts of effects and coeffects to runtime mechanisms. *Revertible effects* address local temporal composability: every context transformation carries an inverse that the runtime tracks, and both tracking and recovery preserve composition, so the context is recovered upon component removal. *Reactive coeffects* address local spatial composability: a component is notified against its coeffect specification whenever the context changes, each change classified as activating, deactivating, or neutral, with coeffect isolation varying what a declared key resolves to and coeffect interception varying how the binding is used. We unify the effect context and the coeffect context into a single *context type*, in which an observational equivalence on the coeffects supplies the effects with independence, constituting a programming paradigm for spatiotemporal composability. Combining these mechanisms into the notion of a *component* then gives a calculus of dynamic composition, whose metatheory carries spatiotemporal composability from a single component to a whole system of interleaved components. We realize this paradigm as the *Cordis* meta-framework, with a core library providing effect tracking and coeffect resolution, as well as a declarative component loader with configuration reconciliation and hot module replacement. The Koishi case study validates the design of Cordis in a production system with over 4000 community plugins.

Beyond human-curated plugin ecosystems, a compelling direction for future validation is self-evolving agent harnesses (Section 1.2.2), where an AI agent generates and replaces its own harness components continuously and with little human oversight. Applying Cordis in such a setting would validate the temporal guarantees of complete recovery under rapid component replacement, as well as the spatial guarantees of dependency coordination under frequent topological change. Such validation would demonstrate the paradigm's applicability as a foundation for recoverable, coordinated, and continuous self-evolution in agent harnesses and other autonomous systems.

References

[1] D. L. Parnas, "On the criteria to be used in decomposing systems into modules," *Communications of the ACM*, vol. 15, no. 12, pp. 1053–1058, 1972, doi: 10.1145/361598.361623.

[2] D. Birsan, "On Plug-ins and Extensible Architectures," *ACM Queue*, vol. 3, no. 2, pp. 40–46, 2005, doi: 10.1145/1053331.1053345.

[3] B. Burns, B. Grant, D. Oppenheimer, E. Brewer, and J. Wilkes, "Borg, Omega, and Kubernetes," *Communications of the ACM*, vol. 59, no. 5, pp. 50–57, 2016, doi: 10.1145/2890784.

[4] B. Stroustrup, *The Design and Evolution of C++*. Addison-Wesley, 1994.

[5] S. Marlow, S. Peyton Jones, A. Moran, and J. Reppy, "Asynchronous Exceptions in Haskell," in *Proceedings of the ACM SIGPLAN 2001 Conference on Programming Language Design and Implementation*, in PLDI '01. New York, NY, USA: Association for Computing Machinery, 2001, pp. 274–285. doi: 10.1145/378795.378858.

[6] L. Cardelli, "Program Fragments, Linking, and Modularization," in *Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (*POPL 1997*), ACM Press, 1997, pp. 266–277. doi: 10.1145/263699.263735.

[7] C. Szyperski, *Component Software: Beyond Object-Oriented Programming*, 2nd ed. Addison-Wesley, 2002.

[8] R. Lopopolo, "Harness Engineering: Leveraging Codex in an Agent-First World." [Online]. Available: https://openai.com/index/harness-engineering/

[9] Anthropic, "Harness Design for Long-Running Application Development." [Online]. Available: https://www.anthropic.com/engineering/harness-design-long-running-apps

[10] L. Wang *et al.*, "A Survey on Large Language Model Based Autonomous Agents," *Frontiers of Computer Science*, vol. 18, no. 6, p. 186345, 2024, doi: 10.1007/s11704-024-40231-1.

[11] Y. Qin *et al.*, "Tool Learning with Foundation Models," *ACM Computing Surveys*, 2025, doi: 10.1145/3704435.

[12] C. Packer, V. Fang, S. G. Patil, K. Lin, S. Wooders, and J. E. Gonzalez, "MemGPT: Towards LLMs as Operating Systems," *CoRR*, vol. abs/2310.08560, 2023.

[13] T. Guo *et al.*, "Large Language Model Based Multi-Agents: A Survey of Progress and Challenges," in *Proceedings of the Thirty-Third International Joint Conference on Artificial Intelligence*, in IJCAI 2024. 2024, pp. 8048–8057. doi: 10.24963/ijcai.2024/890.

[14] T. Cai, X. Wang, T. Ma, X. Chen, and D. Zhou, "Large Language Models as Tool Makers," in *Proceedings of the Twelfth International Conference on Learning Representations*, in ICLR 2024. 2024. [Online]. Available: https://openreview.net/forum?id=qV83K9d5WB

[15] J. Armstrong, "Making Reliable Distributed Systems in the Presence of Software Errors," Doctoral dissertation, 2003. [Online]. Available: https://erlang.org/download/armstrong_thesis_2003.pdf

[16] E. Moggi, "Notions of computation and monads," *Information and Computation*, vol. 93, no. 1, pp. 55–92, 1991, doi: 10.1016/0890-5401(91)90052-4.

[17] G. Plotkin and J. Power, "Adequacy for Algebraic Effects," in *Foundations of Software Science and Computation Structures*, F. Honsell and M. Miculan, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 2001, pp. 1–24.

[18] T. Petricek, D. Orchard, and A. Mycroft, "Coeffects: unified static analysis of context-dependence," in *Proceedings of the 40th International Conference on Automata, Languages, and Programming - Volume Part II*, in ICALP'13. Riga, Latvia: Springer-Verlag, 2013, pp. 385–397. doi: 10.1007/978-3-642-39212-2_35.

[19] M. Gaboardi, S.-ya Katsumata, D. Orchard, F. Breuvart, and T. Uustalu, "Combining effects and coeffects via grading," in *Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming*, in ICFP 2016. Nara, Japan: Association for Computing Machinery, 2016, pp. 476–489. doi: 10.1145/2951913.2951939.

[20] A. Church, "A Formulation of the Simple Theory of Types," *The Journal of Symbolic Logic*, vol. 5, no. 2, pp. 56–68, 1940, doi: 10.2307/2266170.

[21] B. C. Pierce, *Types and Programming Languages*. MIT Press, 2002.

[22] J. M. Lucassen and D. K. Gifford, "Polymorphic Effect Systems," in *Proceedings of the 15th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, in POPL '88. San Diego, California, USA: Association for Computing Machinery, 1988, pp. 47–57. doi: 10.1145/73560.73564.

[23] P. Wadler, "Monads for functional programming," in *Program Design Calculi*, M. Broy, Ed., Berlin, Heidelberg: Springer Berlin Heidelberg, 1993, pp. 233–264.

[24] G. Plotkin and J. Power, "Notions of Computation Determine Monads," in *Foundations of Software Science and Computation Structures*, Berlin, Heidelberg: Springer Berlin Heidelberg, 2002, pp. 342–356. doi: 10.1007/3-540-45931-6_24.

[25] G. Plotkin and M. Pretnar, "Handlers of Algebraic Effects," in *Programming Languages and Systems (ESOP)*, Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 80–94. doi: 10.1007/978-3-642-00590-9_7.

[26] M. Pretnar, "An Introduction to Algebraic Effects and Handlers. Invited tutorial paper," *Electron. Notes Theor. Comput. Sci.*, vol. 319, no. C, pp. 19–35, Dec. 2015, doi: 10.1016/j.entcs.2015.12.003.

[27] D. Leijen, "Koka: Programming with Row Polymorphic Effect Types," *Electronic Proceedings in Theoretical Computer Science*, vol. 153, pp. 100–126, Jun. 2014, doi: 10.4204/eptcs.153.8.

[28] D. Leijen, "Type directed compilation of row-typed algebraic effects," in *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages*, in POPL '17. Paris, France: Association for Computing Machinery, 2017, pp. 486–499. doi: 10.1145/3009837.3009872.

[29] A. Bauer and M. Pretnar, "Programming with algebraic effects and handlers," *Journal of Logical and Algebraic Methods in Programming*, vol. 84, no. 1, pp. 108–123, Jan. 2015, doi: 10.1016/j.jlamp.2014.02.001.

[30] K. Sivaramakrishnan *et al.*, "Retrofitting parallelism onto OCaml," *Proc. ACM Program. Lang.*, vol. 4, no. ICFP, Aug. 2020, doi: 10.1145/3408995.

[31] T. Petricek, D. Orchard, and A. Mycroft, "Coeffects: a calculus of context-dependent computation," in *Proceedings of the 19th ACM SIGPLAN International Conference on Functional Programming*, in ICFP '14. Gothenburg, Sweden: Association for Computing Machinery, 2014, pp. 123–135. doi: 10.1145/2628136.2628160.

[32] T. Uustalu and V. Vene, "Comonadic Notions of Computation," *Electronic Notes in Theoretical Computer Science*, vol. 203, no. 5, pp. 263–284, 2008, doi: 10.1016/j.entcs.2008.05.029.

[33] A. Brunel, M. Gaboardi, D. Mazza, and S. Zdancewic, "A Core Quantitative Coeffect Calculus," in *Proceedings of the 23rd European Symposium on Programming Languages and Systems - Volume 8410*, Berlin, Heidelberg: Springer-Verlag, 2014, pp. 351–370. doi: 10.1007/978-3-642-54833-8_19.

[34] J. Reed and B. C. Pierce, "Distance makes the types grow stronger: a calculus for differential privacy," *SIGPLAN Not.*, vol. 45, no. 9, pp. 157–168, Sep. 2010, doi: 10.1145/1932681.1863568.

[35] M. Abadi, A. Banerjee, N. Heintze, and J. G. Riecke, "A core calculus of dependency," in *Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, in POPL '99. San Antonio, Texas, USA: Association for Computing Machinery, 1999, pp. 147–160. doi: 10.1145/292540.292555.

[36] D. E. Denning, "A lattice model of secure information flow," *Commun. ACM*, vol. 19, no. 5, pp. 236–243, May 1976, doi: 10.1145/360051.360056.

[37] U. Dal Lago and F. Gavazzo, "A relational theory of effects and coeffects," *Proc. ACM Program. Lang.*, vol. 6, no. POPL, Jan. 2022, doi: 10.1145/3498692.

[38] M. Fowler, "Inversion of Control Containers and the Dependency Injection pattern." [Online]. Available: https://martinfowler.com/articles/injection.html

[39] A. M. Pitts and I. D. B. Stark, "Observable Properties of Higher Order Functions that Dynamically Create Local Names, or What's New?," in *Mathematical Foundations of Computer Science 1993 (MFCS 1993)*, in Lecture Notes in Computer Science, vol. 711. Springer, 1993, pp. 122–141. doi: 10.1007/3-540-57182-5_8.

[40] G. D. Plotkin, "LCF Considered as a Programming Language," *Theoretical Computer Science*, vol. 5, no. 3, pp. 223–255, 1977, doi: 10.1016/0304-3975(77)90044-5.

[41] D. R. Ghica, K. Muroya, and T. Waugh Ambridge, "A Robust Graph-Based Approach to Observational Equivalence," *Logical Methods in Computer Science*, vol. 21, no. 2, p. 8:1–8:95, 2025, doi: 10.46298/LMCS-21(2:8)2025.

[42] X. Leroy and S. Blazy, "Formal Verification of a C-like Memory Model and Its Uses for Verifying Program Transformations," *Journal of Automated Reasoning*, vol. 41, no. 1, pp. 1–31, 2008, doi: 10.1007/s10817-008-9099-0.

[43] R. P. James and A. Sabry, "Yield: Mainstream Delimited Continuations," in *First International Workshop on the Theory and Practice of Delimited Continuations (TPDC 2011)*, 2011, pp. 20–32. [Online]. Available: https://homes.luddy.indiana.edu/sabry/files/yield.pdf

[44] A. W. Mazurkiewicz, "Trace Theory," in *Petri Nets: Central Models and Their Properties, Advances in Petri Nets 1986, Part II*, in Lecture Notes in Computer Science, vol. 255. Springer, 1986, pp. 279–324. doi: 10.1007/3-540-17906-2_30.

[45] U. A. Acar, G. E. Blelloch, and R. Harper, "Adaptive functional programming," *ACM Transactions on Programming Languages and Systems*, vol. 28, no. 6, pp. 990–1034, 2006, doi: 10.1145/1186632.1186634.

[46] webpack, "Hot Module Replacement." [Online]. Available: https://webpack.js.org/api/hot-module-replacement/

[47] Vite, "HMR API." [Online]. Available: https://vite.dev/guide/api-hmr

[48] E. N. (M. Elnozahy, L. Alvisi, Y.-M. Wang, and D. B. Johnson, "A Survey of Rollback-Recovery Protocols in Message-Passing Systems," *ACM Computing Surveys*, vol. 34, no. 3, pp. 375–408, 2002, doi: 10.1145/568522.568525.

[49] H. Garcia-Molina and K. Salem, "Sagas," in *Proceedings of the 1987 ACM SIGMOD International Conference on Management of Data*, in SIGMOD '87. 1987, pp. 249–259. doi: 10.1145/38713.38742.

[50] OSGi Alliance, *OSGi Core Release 8*. OSGi Alliance, 2020. [Online]. Available: https://docs.osgi.org/specification/osgi.core/8.0.0/

[51] J. Kramer and J. Magee, "The Evolving Philosophers Problem: Dynamic Change Management," *IEEE Transactions on Software Engineering*, vol. 16, no. 11, pp. 1293–1306, 1990, doi: 10.1109/32.60317.

[52] Y. Vandewoude, P. Ebraert, Y. Berbers, and T. D'Hondt, "Tranquility: A Low Disruptive Alternative to Quiescence for Ensuring Safe Dynamic Updates," *IEEE Transactions on Software Engineering*, vol. 33, no. 12, pp. 856–868, 2007, doi: 10.1109/tse.2007.70733.

[53] J. S. Rellermeyer, G. Alonso, and T. Roscoe, "R-OSGi: Distributed Applications Through Software Modularization," in *Proceedings of the ACM/IFIP/USENIX 8th International Middleware Conference*, in Middleware '07. 2007, pp. 1–20. doi: 10.1007/978-3-540-76778-7_1.

[54] J. B. Dennis and E. C. Van Horn, "Programming Semantics for Multiprogrammed Computations," *Communications of the ACM*, vol. 9, no. 3, pp. 143–155, 1966, doi: 10.1145/365230.365252.

[55] M. S. Miller, K.-P. Yee, and J. Shapiro, "Capability Myths Demolished," technical report SRL2003–2, 2003. [Online]. Available: http://zesty.ca/capmyths/usenix.pdf

[56] R. N. M. Watson, J. Anderson, B. Laurie, and K. Kennaway, "Capsicum: Practical Capabilities for UNIX," in *Proceedings of the 19th USENIX Security Symposium*, 2010, pp. 29–46. [Online]. Available: https://www.usenix.org/legacy/events/sec10/tech/full_papers/Watson.pdf

[57] R. Wahbe, S. Lucco, T. E. Anderson, and S. L. Graham, "Efficient Software-Based Fault Isolation," in *Proceedings of the 14th ACM Symposium on Operating Systems Principles*, in SOSP '93. 1993, pp. 203–216. doi: 10.1145/168619.168635.

[58] A. Barth, A. P. Felt, P. Saxena, and A. Boodman, "Protecting Browsers from Extension Vulnerabilities," in *Proceedings of the 17th Annual Network and Distributed System Security Symposium*, in NDSS '10. 2010. [Online]. Available: https://www.ndss-symposium.org/ndss2010/protecting-browsers-extension-vulnerabilities/

[59] W. W. Ho and R. A. Olsson, "An Approach to Genuine Dynamic Linking," *Software: Practice and Experience*, vol. 21, no. 4, pp. 375–390, 1991, doi: 10.1002/SPE.4380210404.

[60] P. Wadler and S. Blott, "How to Make Ad-hoc Polymorphism Less Ad Hoc," in *Proceedings of the 16th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, in POPL '89. 1989, pp. 60–76. doi: 10.1145/75277.75283.

[61] N. D. Matsakis and F. S. K. II, "The Rust Language and Type System," in *ACM SIGPLAN ML Family Workshop*, Gothenburg, Sweden, Sep. 2014.

[62] D. Dreyer, R. Harper, M. M. T. Chakravarty, and G. Keller, "Modular Type Classes," in *Proceedings of the 34th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, in POPL '07. 2007, pp. 63–70. doi: 10.1145/1190216.1190229.

[63] Microsoft, "Declaration Merging." [Online]. Available: https://www.typescriptlang.org/docs/handbook/declaration-merging.html

[64] T. Van Cutsem and M. S. Miller, "Proxies: Design Principles for Robust Object-oriented Intercession APIs," in *Proceedings of the 6th Symposium on Dynamic Languages*, in DLS '10. 2010, pp. 59–72. doi: 10.1145/1869631.1869638.

[65] R. Hettinger, "Descriptor HowTo Guide." [Online]. Available: https://docs.python.org/3/howto/descriptor.html

[66] P. Maes, "Concepts and Experiments in Computational Reflection," in *Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, 1987, pp. 147–155. doi: 10.1145/38765.38821.

[67] G. Bracha and D. M. Ungar, "Mirrors: design principles for meta-level facilities of object-oriented programming languages," in *Proceedings of the 19th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, 2004, pp. 331–344. doi: 10.1145/1028976.1029004.

[68] R. Rouvoy and P. Merle, "Leveraging component-based software engineering with Fraclet," *Annals of Telecommunications*, vol. 64, no. 1–2, pp. 65–79, 2009, doi: 10.1007/s12243-008-0072-z.

[69] E. Burmako, "Scala Macros: Let Our Powers Combine!," in *Proceedings of the 4th Workshop on Scala*, in SCALA@ECOOP '13. 2013, p. 3:1–3:10. doi: 10.1145/2489837.2489840.

[70] S. Raemaekers, A. van Deursen, and J. Visser, "Semantic Versioning and Impact of Breaking Changes in the Maven Repository," *Journal of Systems and Software*, vol. 129, pp. 140–158, 2017, doi: 10.1016/j.jss.2016.04.008.

[71] P. Lam, J. Dietrich, and D. J. Pearce, "Putting the Semantics into Semantic Versioning," in *Proceedings of the 2020 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*, in Onward! '20. 2020, pp. 157–179. doi: 10.1145/3426428.3426922.

[72] P. Abate, R. Di Cosmo, R. Treinen, and S. Zacchiroli, "Dependency Solving: A Separate Concern in Component Evolution Management," *Journal of Systems and Software*, vol. 85, no. 10, pp. 2228–2240, 2012, doi: 10.1016/j.jss.2012.02.018.

[73] L. Cardelli, "Structural Subtyping and the Notion of Power Type," in *Proceedings of the 15th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, in POPL '88. 1988, pp. 70–79. doi: 10.1145/73560.73566.

[74] B. Meyer, "Applying "Design by Contract"," *Computer*, vol. 25, no. 10, pp. 40–51, 1992, doi: 10.1109/2.161279.

[75] B. C. Pierce, "Bounded Quantification is Undecidable," *Information and Computation*, vol. 112, no. 1, pp. 131–165, 1994, doi: 10.1006/inco.1994.1055.

[76] A. Haas *et al.*, "Bringing the web up to speed with WebAssembly," in *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, ACM, 2017, pp. 185–200. doi: 10.1145/3062341.3062363.

[77] M. M. Swift, B. N. Bershad, and H. M. Levy, "Improving the reliability of commodity operating systems," in *Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP)*, ACM, 2003, pp. 207–222. doi: 10.1145/945445.945466.

[78] M. M. Swift, M. Annamalai, B. N. Bershad, and H. M. Levy, "Recovering device drivers," *ACM Transactions on Computer Systems*, vol. 24, no. 4, pp. 333–360, 2006, doi: 10.1145/1189256.1189257.

[79] D. E. Porter, O. S. Hofmann, C. J. Rossbach, A. Benn, and E. Witchel, "Operating System Transactions," in *Proceedings of the 22nd ACM Symposium on Operating Systems Principles (SOSP)*, ACM, 2009, pp. 161–176. doi: 10.1145/1629575.1629591.

[80] O. Kiselyov and C.-chieh Shan, "Delimited Continuations in Operating Systems," in *Modeling and Using Context (CONTEXT 2007)*, in Lecture Notes in Computer Science, vol. 4635. Springer, 2007, pp. 291–302. doi: 10.1007/978-3-540-74255-5_22.

[81] E. Dolstra and A. Löh, "NixOS: a purely functional Linux distribution," in *Proceedings of the 13th ACM SIGPLAN International Conference on Functional Programming (ICFP)*, ACM, 2008, pp. 367–378. doi: 10.1145/1411204.1411255.

[82] ZIO, "ZIO: Type-safe, composable asynchronous and concurrent programming for Scala." [Online]. Available: https://zio.dev/

[83] Effect, "Effect: A TypeScript library for building robust applications." [Online]. Available: https://effect.website/

[84] G. Canti, "fp-ts: Functional programming in TypeScript." [Online]. Available: https://github.com/gcanti/fp-ts

[85] J. I. Brachthäuser, P. Schuster, and K. Ostermann, "Effects as capabilities: effect handlers and lightweight effect polymorphism," *Proc. ACM Program. Lang.*, vol. 4, no. OOPSLA, 2020, doi: 10.1145/3428194.

[86] J. I. Brachthäuser, P. Schuster, E. Lee, and A. Boruch-Gruszecki, "Effects, capabilities, and boxes: from scope-based reasoning to type-based reasoning and back," *Proc. ACM Program. Lang.*, vol. 6, no. OOPSLA1, 2022, doi: 10.1145/3527320.

[87] C. Heunen, R. Kaarsgaard, and M. Karvonen, "Reversible Effects as Inverse Arrows," in *Proceedings of the Thirty-Fourth Conference on the Mathematical Foundations of Programming Semantics (MFPS XXXIV)*, in Electronic Notes in Theoretical Computer Science, vol. 341. 2018, pp. 179–199. doi: 10.1016/j.entcs.2018.11.009.

[88] D. Orchard, V.-B. Liepelt, and H. Eades III, "Quantitative program reasoning with graded modal types," *Proc. ACM Program. Lang.*, vol. 3, no. ICFP, 2019, doi: 10.1145/3341714.

[89] R. Bianchini, F. Dagnino, P. Giannini, E. Zucca, and M. Servetto, "Coeffects for sharing and mutation," *Proc. ACM Program. Lang.*, vol. 6, no. OOPSLA2, Oct. 2022, doi: 10.1145/3563319.

[90] R. Bianchini, F. Dagnino, P. Giannini, and E. Zucca, "A Java-like calculus with heterogeneous coeffects," *Theoretical Computer Science*, vol. 971, p. 114063, 2023, doi: https://doi.org/10.1016/j.tcs.2023.114063.

[91] C. Torczon, E. Suárez Acevedo, S. Agrawal, J. Velez-Ginorio, and S. Weirich, "Effects and Coeffects in Call-by-Push-Value," *Proc. ACM Program. Lang.*, vol. 8, no. OOPSLA2, Oct. 2024, doi: 10.1145/3689750.

[92] R. Hirschfeld, P. Costanza, and O. Nierstrasz, "Context-oriented Programming," *Journal of Object Technology*, vol. 7, no. 3, pp. 125–151, 2008, doi: 10.5381/jot.2008.7.3.a4.

[93] P. Costanza and R. Hirschfeld, "Language constructs for context-oriented programming: an overview of ContextL," in *Proceedings of the 2005 Symposium on Dynamic Languages* (*DLS '05*), ACM, 2005, pp. 1–10. doi: 10.1145/1146841.1146842.

[94] G. Salvaneschi, C. Ghezzi, and M. Pradella, "Context-oriented programming: A software engineering perspective," *Journal of Systems and Software*, vol. 85, no. 8, pp. 1801–1817, 2012, doi: 10.1016/j.jss.2012.03.024.

[95] G. Kiczales *et al.*, "Aspect-Oriented Programming," in *ECOOP'97 — Object-Oriented Programming, 11th European Conference*, in Lecture Notes in Computer Science, vol. 1241. Springer, 1997, pp. 220–242. doi: 10.1007/BFb0053381.

[96] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold, "An Overview of AspectJ," in *ECOOP 2001 — Object-Oriented Programming, 15th European Conference*, in Lecture Notes in Computer Science, vol. 2072. Springer, 2001, pp. 327–353. doi: 10.1007/3-540-45337-7_18.

[97] A. Popovici, T. Gross, and G. Alonso, "Dynamic Weaving for Aspect-Oriented Programming," in *Proceedings of the 1st International Conference on Aspect-Oriented Software Development* (*AOSD 2002*), ACM, 2002, pp. 141–147. doi: 10.1145/508386.508404.

[98] J. Bonér, "What Are the Key Issues for Commercial AOP Use: How Does AspectWerkz Address Them?," in *Proceedings of the 3rd International Conference on Aspect-Oriented Software Development* (*AOSD 2004*), ACM, 2004, pp. 5–6. doi: 10.1145/976270.976273.

[99] M. Hicks, J. T. Moore, and S. Nettles, "Dynamic Software Updating," in *Proceedings of the ACM SIGPLAN 2001 Conference on Programming Language Design and Implementation*, in PLDI '01. 2001, pp. 13–23. doi: 10.1145/378795.378798.

[100] G. Stoyle, M. Hicks, G. Bierman, P. Sewell, and I. Neamtiu, "Mutatis Mutandis: Safe and Predictable Dynamic Software Updating," in *Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, in POPL '05. 2005, pp. 183–194. doi: 10.1145/1040305.1040321.

[101] C. M. Hayden, K. Saur, E. K. Smith, and M. Hicks, "Kitsune: Efficient, General-Purpose Dynamic Software Updating for C," *ACM Trans. Program. Lang. Syst.*, vol. 36, no. 4, 2014, doi: 10.1145/2629460.

[102] M. Overeem, M. Spoor, and S. Jansen, "The Dark Side of Event Sourcing: Managing Data Conversion," in *IEEE 24th International Conference on Software Analysis, Evolution and Reengineering*, in SANER '17. 2017, pp. 193–204. doi: 10.1109/SANER.2017.7884621.

[103] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*. Boston, MA: Addison-Wesley, 1994.

[104] D. Leijen, "Algebraic Effect Handlers with Resources and Deep Finalization," technical report MSR-TR-2018-10, Apr. 2018. [Online]. Available: https://www.microsoft.com/en-us/research/publication/algebraic-effect-handlers-resources-deep-finalization/

[105] M. Fowler, "Event Sourcing." 2005.

[106] J. Lee, J. Ahn, and K. Yi, "React-tRace: A Semantics for Understanding React Hooks," *Proc. ACM Program. Lang.*, vol. 9, no. OOPSLA2, pp. 471–498, 2025, doi: 10.1145/3763067.

[107] N. Shavit and D. Touitou, "Software Transactional Memory," in *Proceedings of the Fourteenth Annual ACM Symposium on Principles of Distributed Computing*, in PODC '95. 1995, pp. 204–213. doi: 10.1145/224964.224987.

[108] T. Harris, S. Marlow, S. Peyton Jones, and M. Herlihy, "Composable Memory Transactions," in *Proceedings of the Tenth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, in PPoPP '05. 2005, pp. 48–60. doi: 10.1145/1065944.1065952.

[109] M. Herlihy and J. E. B. Moss, "Transactional Memory: Architectural Support for Lock-Free Data Structures," in *Proceedings of the 20th Annual International Symposium on Computer Architecture*, in ISCA '93. 1993, pp. 289–300. doi: 10.1145/165123.165164.

[110] R. Landauer, "Irreversibility and Heat Generation in the Computing Process," *IBM Journal of Research and Development*, vol. 5, no. 3, pp. 183–191, 1961, doi: 10.1147/rd.53.0183.

[111] C. H. Bennett, "Logical Reversibility of Computation," *IBM Journal of Research and Development*, vol. 17, no. 6, pp. 525–532, 1973, doi: 10.1147/rd.176.0525.

[112] T. Yokoyama and R. Glück, "A Reversible Programming Language and its Invertible Self-Interpreter," in *Proceedings of the 2007 ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation*, in PEPM '07. 2007, pp. 144–153. doi: 10.1145/1244381.1244404.

[113] V. Danos and J. Krivine, "Reversible Communicating Systems," in *CONCUR 2004 — Concurrency Theory, 15th International Conference*, in Lecture Notes in Computer Science, vol. 3170. Springer, 2004, pp. 292–307. doi: 10.1007/978-3-540-28644-8_19.

[114] I. Phillips and I. Ulidowski, "Reversing Algebraic Process Calculi," in *Foundations of Software Science and Computation Structures, 9th International Conference (FOSSACS 2006)*, in Lecture Notes in Computer Science, vol. 3921. Springer, 2006, pp. 246–260. doi: 10.1007/11690634_17.

[115] P. Wadler, "Linear Types Can Change the World!," in *Programming Concepts and Methods: Proceedings of the IFIP Working Group 2.2/2.3 Working Conference*, North-Holland, 1990, pp. 561–581. [Online]. Available: https://homepages.inf.ed.ac.uk/wadler/papers/linear/linear.ps

[116] A. Lenharth, V. S. Adve, and S. T. King, "Recovery domains: an organizing principle for recoverable operating systems," in *Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, ACM, 2009, pp. 49–60. doi: 10.1145/1508244.1508251.

[117] C. Walls, *Spring in Action*, 6th ed. Manning Publications, 2022. [Online]. Available: https://www.manning.com/books/spring-in-action-sixth-edition

[118] C. Escoffier, R. S. Hall, and P. Lalanda, "iPOJO: an Extensible Service-Oriented Component Framework," in *IEEE International Conference on Services Computing*, 2007, pp. 474–481. doi: 10.1109/SCC.2007.74.

[119] H. Cervantes and R. S. Hall, "Autonomous Adaptation to Dynamic Availability Using a Service-Oriented Component Model," in *Proceedings of the 26th International Conference on Software Engineering*, in ICSE '04. 2004, pp. 614–623. doi: 10.1109/ICSE.2004.1317483.

[120] C. Elliott and P. Hudak, "Functional Reactive Animation," in *Proceedings of the Second ACM SIGPLAN International Conference on Functional Programming*, in ICFP '97. 1997, pp. 263–273. doi: 10.1145/258948.258973.

[121] G. H. Cooper and S. Krishnamurthi, "Embedding Dynamic Dataflow in a Call-by-Value Language," in *Programming Languages and Systems (ESOP 2006)*, in Lecture Notes in Computer Science, vol. 3924. Springer, 2006, pp. 294–308. doi: 10.1007/11693024_20.

[122] I. Maier and M. Odersky, "Deprecating the Observer Pattern with Scala.React," technical report EPFL-REPORT-176887, 2012. [Online]. Available: https://infoscience.epfl.ch/record/176887

[123] E. Bainomugisha, A. L. Carreton, T. Van Cutsem, W. De Meuter, and others, "A Survey on Reactive Programming," *ACM Comput. Surv.*, vol. 45, no. 4, 2013, doi: 10.1145/2501654.2501666.

[124] A. Margara and G. Salvaneschi, "On the Semantics of Distributed Reactive Programming: The Cost of Consistency," *IEEE Trans. Software Eng.*, vol. 44, no. 7, pp. 689–711, 2018, doi: 10.1109/TSE.2018.2833109.