Locus Solum — Quickstart (Parts 1–9)

A contributor-onboarding guide to Ludics (Girard)

Prepared for the Mesh repo • 2025-09-14

Contents

Locus Solum — Part 1 (Contributor Notes)

Scope. Entry point to Girard's **ludics**: from rules of logic to the **logic of rules**. The focus is on *locality* (addresses), *interaction* (plays), and how meaning arises from **what normalizes against what**.

---

1) Shift of viewpoint

Traditional proof theory starts with **formulas** and **inference rules**. Ludics starts with **interactions** between **actions** at **addresses** (loci), and lets logical structure **emerge** from those interactions.

> Slogan: *from rules of logic → logic of rules*.

---

2) Loci (addresses)

A **locus** is an address (place) where a move can occur. Addresses branch (e.g., `σ·1`, `σ·2`), supporting independent sub-interactions and explicit locality.

ASCII — branching addresses

```
σ
├─ σ·1
└─ σ·2
```

---

3) Actions & polarity

An **action** is a move at a locus with a **polarity**:

• **Positive (+)** — offer/open structure (enable a finite set of sub-addresses).

• **Negative (−)** — focus/challenge a specific sub-address.

A special **daimon** `♦` (⊥) can terminate play immediately (technical and conceptual "top").

ASCII — alternation at a locus

```
E:  − σ        (tester requests at σ)
D:  + σ        (design answers / opens)
E:  − σ·1      (tester focuses left branch)
D:  + σ·1.k    (design continues under σ·1; k in the enabled set)
```

---

4) Designs, chronicles, views

A **design** is a proof-like *strategy* built from justified actions at addresses.

• **Chronicle / view**: an observable path (alternating ±) through a design.

• Designs may be infinite but are **locally finite** and **coherent** (justifications line up, no clashes at an address).

ASCII — design as “what I do where”

```
Design D:
  + σ
  + σ·1
  + σ·2      # σ·2 may be a stub (undeveloped) or developed later
```

---

5) Interaction, normalization, orthogonality

Running two designs along shared addresses yields **interaction**. The run **normalizes** when it reaches a terminal configuration (often via `♦`).

• **Orthogonality**: `D ⊥ E` iff the interaction **converges** (normalizes).

• **Behaviour**: a set `B` of designs closed under **bi-orthogonality**: `B = B⊥⊥`.

ASCII — success vs failure

```
Success:
E: −σ ; D: +σ ; E: −σ·1 ; D: +σ·1 ; … ; ♦   ⇒ D ⊥ E

Failure (stuck):
D only develops σ·2; E tests σ·1 → no move at σ·1  ⇒ no ♦ ⇒ D ⊥/ E
```

---

6) Why it matters (orientation)

• **Proofs as programs** at the level of **interaction** (not just λ-terms).

• **Cut = interaction**, **cut-elimination = normalization**.

• **Semantics by tests**: meaning is **what passes all counter-tests**.

> Keep three questions in mind: *at which address? with what polarity? what counter-move would normalize?*

Locus Solum — Part 2 (Contributor Notes)

Scope. The “micro-geometry” of interaction: formalizing **loci**, **polarized actions**, **ramifications/directories**, **chronicles/paths**, and **designs**.

---

1) Loci and ramifications

A positive action may **enable** a **finite set** of sub-addresses (a *ramification*). The collection of immediately enabled children at the root forms the **directory** of a (negative) behaviour.

ASCII — loci & ramifications

```
Root σ
(+ , σ, {1,2})   # positive action opens two children

σ
├─ σ·1
└─ σ·2
```

---

2) Polarized actions (±)

• **`(+ , ξ, I)`** — at address `ξ`, open sub-addresses indexed by finite set `I`.

• **`(− , ξ.i)`** — focus/respond at `ξ.i` (for `i ∈ I`).

Alternation and justification (who enables what, where) govern which chronicles are **coherent**.

---

3) Chronicles, paths, designs

• **Chronicle**: coherent, alternating ± sequence at addresses with proper justifications.

• **Path**: a chronicle that is actually *playable*.

• **Design**: a set of chronicles closed under prefixes and coherence; may be infinite but is locally finite.

ASCII — coherence sketch

```
… + σ  enables {1,2}
   − σ·1  is legal
   − σ·3  is illegal (not enabled)  → incoherent
```

---

4) Directories and additivity (preview)

At a shared base `σ`, **with** behaves as **intersection** of behaviours (`B & C := B ∩ C`); **plus** is its polar dual (`B ⊕ C := (B⊥ & C⊥)⊥`). The *directory* controls which branches are testable at the top.

ASCII — with = intersection (local)

```
Dir(B) = {1,2}
Dir(C) = {2,3}

Shared, testable branch at σ·2 → B & C keeps exactly what passes σ·2-tests.
```

---

5) Implementation nudge

• Represent addresses as dotted strings (`"σ.1.2"`) or lists (`['σ',1,2]`).

• Record actions as `{ locus, polarity: '+'|'-', enables?: number[] }`.

• A simple **interaction driver** alternates moves at the active address; stop on `♦` or incoherence.

Locus Solum — Part 3 (Contributor Notes)

Scope. The dynamics proper: **interaction/normalization**, **orthogonality**, and **behaviours** as bi-orthogonally closed sets (types by tests).

---

1) Interaction as computation

Given a design `D` and a counter-design `E`, **run** them along shared addresses. The alternation discipline (±) and justifications determine the legal next moves.

Pseudo-driver (ASCII)

```
while moves remain:
  pick the next active address ξ on the interface
  if last at ξ was −: D must produce + at ξ (or ♦)
  if last at ξ was +: E must answer − at some ξ.i
  if neither can move coherently: fail (non-orthogonal)
  if ♦ appears: success (convergent)
```

---

2) Orthogonality

`D ▯ E` iff the run **converges** (reaches ♦ or designated success). Orthogonality is **compatibility** of strategies.

ASCII — pass/fail

```
Pass:    E −σ ; D +σ ; E −σ·1 ; D +σ·1 ; … ; ♦
Fail:    E −σ ; D +σ ; E −σ·1 ; D ???   (no move) → stuck
```

---

3) Behaviours via bi-orthogonality

A **behaviour** `B` is a set of designs **closed under tests of tests**:

```
B⊥   = { E | ∀D∈B,  D ⊥ E }
B⊥⊥ = { D | ∀E∈B⊥, D ⊥ E }
B    = B⊥⊥
```

This gives **meaning = interaction-stable set** (types as behaviours).

---

4) Early consequences

• **Separation by tests**: designs are determined (observationally) by the tests they pass.

• **Internal completeness (linear core)**: for additive constructions at a fixed base, bi-closure does not add “mystery” elements—interaction saturates the behaviour.

---

5) Mini end-to-end example

```
D:
  + σ           (+ , σ, {1,2})
  + σ·1         (+ , σ·1, {a})

E:
  − σ
  − σ·1
  − σ·1.a

Run: E −σ ; D +σ ; E −σ·1 ; D +σ·1 ; E −σ·1.a ; D +σ·1.a ; … ; ♦  ⇒ D ⊥ E
```

Locus Solum — Part 4 (Polished Notes)

Theme. Behaviours and connectives **at loci**: additivity as **set-theoretic** operations (local, directory-aware); multiplicatives via **delocalised composition**; internal completeness for the additive core.

4.1 Behaviours recap (local viewpoint)

• A **behaviour** `B` is a set of designs closed under **bi-orthogonality**: `B = B⊥⊥`.

• All constructions in this part are **local to a base locus** (say `σ`), where **directories** (immediate enabled children) control what can be tested at the top.

ASCII — Directory at the root

```
(+ , σ, {1,2,3})   # a positive move enabling children {1,2,3}
Dir(B) = {1,2}     # for a negative behaviour B: tests can target σ·1 or σ·2
```

4.2 Additives: **with = intersection**, **plus = dual**

• **Negative additive (with).** On the same base/directory:

`B & C := B ∩ C`

Reading: a design is in `B & C` iff it passes **all** tests admissible at that base for **both** `B` and `C`.

• **Positive additive (plus).** By polarity:

`B ⊕ C := (B⊥ & C⊥)⊥`.

ASCII — “with = ∩” at a locus

```
Dir(B) = {1,2}       Dir(C) = {2,3}
Shared testable branch at σ·2 only
→ B & C keeps exactly the designs that pass σ·2-tests
```

Laws (local, same base).

• Idempotent, commutative, associative **on the nose** (no up-to-iso packaging).

• Monotone in each argument; `&` distributes over arbitrary intersections (closure under ∩).

4.3 Disjointness & delocation

• **Disjoint** (negative) behaviours: `Dir(B) ∩ Dir(C) = ∅`. Then **incarnation** yields a true product:

`|B & C| ≅ |B| × |C|` (components develop independently).

• If directories clash, **delocate** (rename the base) to make them disjoint, then apply the additive operation.

ASCII — Delocation recipe

```
Clash: both open at ε → {1}
Relabel: ρL(G) opens at ε·L·1;  ρR(H) opens at ε·R·1
Now Dir(ρL(G)) ∩ Dir(ρR(H)) = ∅  → safe to form & / ⊕
```

4.4 Multiplicatives by delocalised composition (sketch)

• `B ⊗ C` composes support over **independent** sub-bases; tests **factor** component-wise.

• Dual `B ⅋ C` via orthogonality.

• Practically: keep addresses separated (`σ` vs `τ`); interaction then proceeds in parallel across loci.

4.5 Internal completeness (additive fragment)

For additive constructions at a fixed base, **bi-closure adds nothing** beyond the interaction-generated elements: once the directory and generators are fixed, the behaviour is saturated. This is the “internal completeness” phenomenon that makes additivity algebraic at loci.

4.6 Worked micro-example

```
Base σ
B requires: respond coherently at σ·1
C requires: respond coherently at σ·2

B & C = B ∩ C
Tester picks σ·1 → must normalize (B condition)
Tester picks σ·2 → must normalize (C condition)
→ designs that pass both remain
```

Locus Solum — Part 5 (Polished Notes)

Theme. Exponentials as **address-level** protocols: contraction/weakening/dereliction realized by **fresh sub-loci** and **daimon**; “logic of rules” = rules as **behaviours over interaction**.

5.1 Why exponentials look different in ludics

• Structural rules are **implemented at addresses** rather than as global typing rules.

• **Copy** (contraction) = re-enter the same interface under **fresh children**; **discard** (weakening) = terminate by **daimon** at that branch; **dereliction** = forget the exponential context for a single use.

• This yields an **operational** exponential: a discipline on *how* you may reuse an address, not a primitive connective in syntax.

ASCII — Copy and discard at addresses

```
Client:    − σ
Server:    + σ ↦ {σ·0, σ·1}    # duplicate the service
Tester:    probes σ·0 and σ·1 independently
Discard:   reply ♦ at an unused branch (weakening)
```

5.2 Freshness and uniformity

• Duplicated sub-loci must be **fresh** (no aliasing of σ·i with σ·j).

• Behaviour must be **uniform** across copies: tests that target different σ·i branches cannot detect illicit asymmetries.

5.3 Rules as behaviours on protocols

View each structural rule as a **behaviour over a protocol** of interaction. Orthogonality enforces the discipline: any admissible tester that focuses a duplicated child must be answered coherently; otherwise the design fails.

5.4 Implementation checklist (engineering)

• Generate **fresh addresses** for each copy; record a bijection back to the parent locus.

• Provide a **driver** that can schedule independent sub-runs on σ·i.

• Include **saturation tests** that pick off each child; acceptance requires convergence on every probed child (or explicit ♦).

5.5 Mini-scenario

```
User asks twice at σ
Design opens σ·0, σ·1
Run 1 on σ·0 normalizes to ♦
Run 2 on σ·1 normalizes to ♦
→ structural use is validated (orthogonal to all such testers)
```

Locus Solum — Part 6 (Girard)

Advanced contributor notes — interaction-first logic (ludics) through the “C-entries” of the lexicon and essays: **Completeness (external/internal), Composition (cut), Connectives (layers), Consensus, Consistency/Convergence, Correctness (proof-nets), Curry–Howard, Creative Subject**. filecite turn3file6

---

1) Completeness (external vs internal) filecite turn3file6

• **External completeness (Π¹/classical reading).** If a closed Π¹ sentence `B` is *true*, then `B` is *provable* (on the intended proof system).

• **Internal completeness (ludics).** Work directly with **designs** and **behaviours**: if a design is accepted by the semantics of proofs (i.e., it **normalizes** against all tests in `B⊥`), then it is representable inside the system; equivalently, **`B = B⊥⊥`** (up to **incarnation**).

ASCII (bi-orthogonality lens).

```
Tests:      B⊥   = { E | ∀D∈B,  D ⊥ E }
Validated:  B⊥⊥ = { D | ∀E∈B⊥, D ⊥ E }  →  internal completeness:  B = B⊥⊥
```

---

2) Composition of strategies (Cut = play composition) filecite turn3file6

• The **cut rule** is read as **running two designs** against each other along a shared locus/interface, then **normalizing** the run.

• In sequent form, composition is the usual cut; in ludics, the object-language is *the interaction itself*.

ASCII (cut at address ξ).

```
D₁ … ⊢  A, Γ        D₂ … ⊢  Δ, A⊥
-----------------------------------  cut on A
          … ⊢ Γ, Δ

Ludics driver at locus ξ for A/A⊥:
  E:  −ξ|A   …   D:  +ξ|A⊥   →  alternate moves at ξ → ♦ (success) ⇒ orthogonality
```

---

3) Connectives as “socialization” layers filecite turn3file6

Girard outlines a 3-layer story for building connectives from interaction constraints:

1. **Associative layer** — strict tensor-like composition (fully associative).

2. **Partial (domain-restricted) layer** — operations are total only on compatible loci, but enjoy completeness on that domain.

3. **Spiritual/delocated layer** — add explicit **shifts**/**delocations** to recover completeness without requiring rigid associativity everywhere.

ASCII.

```
assoc     : (X ⊗ Y) ⊗ Z  ≡  X ⊗ (Y ⊗ Z)
partial   : X ⊙ Y         (only defined when loci compatible)
spiritual : X ⊙_σ Y       (delocation/shift σ • completeness preserved)
```

---

4) Consensus (rule-following via tests) filecituturn3file6

• Ludics is a **game by consensus**: if a run **diverges**, you effectively get a **draw**; to *enforce the rule*, craft **consensus-forcing testers** that make deviations unattractive (they lead to stuck runs elsewhere).

ASCII (tester that “herds” play to σ·1).

```
T: −σ → expects −σ·1 → punishes −σ·2 (drives opponent back to the rule)
```

---

5) Consistency · Convergence · Correctness (proof-nets) filecituturn3file6

• **Consistency** recedes in favour of **normalization discipline** (cut-elimination style).

• **Convergence**: definitionally central — `D  E` iff their interaction **normalizes** (possibly via **daimon** ♦).

• **Correctness (proof-nets)**: **switchings** give a practical sequentialization test; the “Give up” step becomes **daimon** in ludics.

ASCII (switchings idea).

```
Net  --(choose switchings)-->  acyclic & connected ?  yes ⇒ sequentializable
```

---

6) Curry–Howard and the “Creative Subject” filecituturn3file6

• We keep **proofs-as-programs**, but “programs” are now **designs** run *by interaction*; this is Curry–Howard in the **behavioural** key.

• “Creative Subject” is historical context (Brouwer): a reminder that proofs are **activities**, resonating with ludics’ operational stance.

Implementation hints (repo)

• Treat **functions/specs** as **contracts on interaction** (what tests must be driven to success).

• Build test harnesses as **counter-designs** (including **consensus-forcing** ones).

• When adding connectives, pick your **layer** (assoc/partial/spiritual) and model it with explicit **locus** management (shifts/delocations).

Minimal glossary refresh

• **Orthogonality `D ▯ E`** — the run `⟨D|E⟩` **normalizes** (♦).

• **Behaviour** — `B = B⊥⊥` (internal completeness).

• **Delocation / shift** — injective renaming of loci to control compatibility.

• **Switchings** — proof-net test for sequentialization.

Locus Solum — Part 7 (Study Notes)

Focus: separation & incarnation; additives as *local* operations; multiplicatives & exponentials at the level of loci (addresses).

Source: Girard, *Locus Solum: From the rules of logic to the logic of rules* (book manuscript).

> These notes are written for contributors familiar with linear logic and game/interaction semantics. They compress the narrative around *tests*, *orthogonality*, *behaviours*, and *incarnation* as developed in ludics.

---

1) Separation via tests (orthogonality preorder collapses to an order)

• For designs \(D, E\) on the same base, write \(D \preceq E\) iff **every** counter-design that normalizes with \(D\) also normalizes with \(E\):

\(D \preceq E \;:\iff\; D^{\perp} \subseteq E^{\perp}.\)

• In ludics this observational preorder is **separating**: equality of orthogonals forces equality of designs up to observational content. Intuition: tests explore *addresses* the same way programs explore memory; a richer design can be told apart by some tester.

ASCII — intuition for separation

```
Tester T probes address σ·1
E answers at σ·1; D does not
⇒ T ∈ E⊥ but T ∉ D⊥  ⇒ D ⊀ E
```

Consequence: designs are determined by the tests they pass; this prepares **incarnation**.

---

2) Incarnation: extracting the “material” content of a behaviour

• A **behaviour** is a bi-orthogonally closed set \(B = B^{\perp\perp}\) of designs on a base.

• Its **incarnation** \(|B|\) keeps only those designs that are *observationally minimal* in \(B\) — i.e., each trace/action is justified by some test.

• Computation view: incarnation cuts off bureaucracy; what remains are the plays that actually *matter* against all tests in \(B^{\perp}\).

ASCII — behaviour vs incarnation

```
B (closed under ⊥⊥)
├── D₁  (material)   ← kept in |B|
├── D₂  (material)   ← kept in |B|
└── D₃  (with dead branches)   ← pruned by incarnation
```

---

3) Additives are *local* (directories) : with = ∩ and plus = dual

Ludics attaches to each (negative) behaviour a **directory**: the finite set of immediate sub-addresses a first positive action may open at the base locus.

• With:

\(B \mathbin{\&} C := B \cap C\) whenever the base/directory is the same.

Semantics: a counter-design must satisfy both sets of tests; intersection is the correct local product.

• Plus:

\(B \oplus C := (B^{\perp} \mathbin{\&} C^{\perp})^{\perp}\).

Polarity flip: the opponent chooses which branch is probed.

ASCII — “with = ∩” at a locus

```
Base σ with directory {1,2}

B requires success at σ·1
C requires success at σ·2
----------------------------
B & C requires both 1 and 2  (intersection of tests)
```

Disjoint case ⇒ product on incarnations.

If \(B, C\) are negative with **disjoint** directories, then material strategies combine independently:

\(|B \mathbin{\&} C| \;\cong\; |B| imes |C|\).

---

4) Multiplicatives & (de)localised composition

Tensor/par arise when plays proceed on **independent** sub-bases (addresses that do not interfere). Interaction factors component-wise; separation ensures the components are recoverable from their tests.

• Tensor \(B \otimes C\): concurrent support on disjoint subloci.

• Par \(B \parr C\): dual by orthogonality.

---

5) Exponentials by fresh loci (copy/erase structurally)

Structural rules are **addressed**:

• **Copy** duplicates by opening fresh child loci (σ·0, σ·1, …) uniformly.

• **Discard** corresponds to immediate success by **daimon** \(♦\) on unused branches.

ASCII — copying at addresses

```
(+ , σ, {0,1,2})   -- spawn three copies
σ·0, σ·1, σ·2      -- each interacts independently under the same discipline
```

---

6) Takeaways for contributors

• Think *observationally*: reason with tests (counter-designs) rather than syntax.

• Use directories to reason about additives; keep an eye on disjointness for products of **incarnations**.

• When modelling resources, place duplication/erasure at **loci**, not as global rules.

Cross-check & scope

These notes match the mid-book exposition where Girard develops separation, incarnation, and the local reading of additives/multiplicatives in ludics. Exact page numbers vary by edition; wording here is normalized for repo docs.

Locus Solum — Part 8 (Study Notes)

Focus: quantifiers and uniformity; internal completeness for linear connectives; the methodological punchline *from rules of logic to logic of rules*.

---

1) Quantifiers and **uniformity**

Quantified behaviours must be tested in a way that does **not depend** on the particular names/parameters the program sees.

• Idea: a tester for \(\forall X. B(X)\) must behave *uniformly* in the choice of \(X\); in practice one models this by requiring invariance (e.g., PER-style relations or name-freshness regimes) so that no test can “peek” at a private code of \(X\).

• In ludics this is enforced at the **address** level: instantiation introduces fresh subloci, and admissible tests cannot distinguish them beyond the discipline fixed by the behaviour.

ASCII — probing a quantified behaviour

```
Tester picks a fresh name a, probes at σ·a
Uniformity ⇒ same outcome as any other fresh name b
```

---

2) Internal completeness for linear connectives

For the linear fragment, **generators + interaction already saturate the behaviour**: bi-closure adds nothing new.

• Example (sum): any design in \(B ⊕ C\) is observationally just “choose left” or “choose right” at the root; closure under □□ is immediate from the testers.

• Moral: locality at addresses plus separation makes the construction of behaviours algebraic rather than completion-heavy.

---

3) Why “with = ∩” matters (methodological)

Treating **with** as literal set-intersection at a *locus* removes isomorphism bureaucracy (no need to identify rebracketings etc.). Girard’s slogan *“Nature abhors an isomorphism”* reads here as: the **addressing** already fixes the shape; semantics should follow the geometry of loci, not abstract object equalities.

ASCII — intersection vs product

```
At one locus σ:         B & C = B ∩ C       (local)
Across disjoint loci:   |B & C| ≅ |B| × |C| (material/product phenomenon)
```

---

4) Practical guidance for contributors

• When encoding data or protocols, keep quantifier **freshness** and **uniformity** explicit in the address space.

• Prefer local algebra (∩, ⊕, ⊗, ⅋) computed from directories to high-level identifications “up to iso.”

• Use **incarnation** to prune unused branches before proving properties by tests.

Cross-check & scope

The notes summarize the later-book material where Girard treats quantification and the methodological wrap-up, harmonized with standard presentations of Ludics. Exact page numbers vary by edition; wording is adapted for clarity in the repo.

Locus Solum — Part 9 (Synthesis & Outlook)

What this closes: We consolidate the machinery of **ludics**—addresses, actions (±), designs, **orthogonality/behaviours**, directories & additivity, multiplicatives, exponentials, and quantification—into a single operational picture and note the main open directions. This is written as an onboarding “last mile” for contributors.

> Core references within our notes: Parts **1–3** for loci/actions/designs/orthogonality; **4** for local additivity and directories; **5** for exponentials at addresses; **6** for completeness/cut/consensus; **7** for separation & incarnation; **8** for quantifiers/uniformity. fileciteturn6file0 filecitet urn6file1 filecite turn6file2 filecite turn6file3 filecite turn6file4 filecite turn6file5 filecite turn6file7 filecite turn6file8

9.1 Executive summary (one screen)

• **Meaning by tests.** *Designs* (strategies) live at **addresses**; **orthogonality** (= convergent interaction) induces **behaviours** as `B = B⊥⊥`. This is the “logic of rules”: logic emerges from **how** rules play. filecite turn6file0 filecite turn6file2

• **Local algebra at a locus.** **With** is literally **intersection** and **plus** its polar dual, governed by **directories** (top-level ramifications). Disjoint negatives give a **product on incarnations**; clashes are handled by **delocation**. filecite turn6file3

• **Resources as addresses.** Exponentials = disciplined **reuse** at addresses (copy to fresh sub-loci; discard via `♦`), with **freshness** + **uniformity** constraints. filecite turn6file4

• **Cut = composition.** Composition is literally **run-and-normalize** along shared loci; internal **completeness** is expressed as `B = B⊥⊥` (up to incarnation). filecite turn6file5

• **Separation & incarnation.** Designs are **determined by their tests**; **incarnation** extracts the material core that products “on the nose” under disjointness. filecite turn6file7

• **Quantifiers.** Add **uniformity** (fresh-name invariance) so testers can’t peek at parameters. filecite turn6file8

9.2 The whole picture on one page (mapping)

• **Address (`ξ`)** → where play happens (locality & independence). filecite turn6file1

• **Action (±)** → who moves / offer vs. focus; **daimon** `♦` may terminate. filecite turn6file0

• **Design** → set of coherent **chronicles** (alternating ± with justifications). □filecite□turn6file1□

• **Interaction** → normalize `⟨D | E⟩`; **orthogonality** defines success. □filecite□turn6file2□

• **Behaviour** → `B = B⊥⊥` (types by tests). □filecite□turn6file2□

• **Directory** → set of immediate sub-addresses testable at the root (additivity driver). □filecite□turn6file3□

• **Additives** → `& = ∩` (negative), `⊕ = (·⊥ & ·⊥)⊥` (positive). □filecite□turn6file3□

• **Multiplicatives** → compose over **independent** bases (tests factor). □filecite□turn6file3□

• **Exponentials** → copy/erase at addresses (freshness + uniformity). □filecite□turn6file4□

• **Separation & Incarnation** → tests determine designs; material core `|B|` supports **products** under disjointness. □filecite□turn6file7□

• **Quantifiers (∀/∃)** → families as (co)limits with **uniform** testing (fresh names). □filecite□turn6file8□

9.3 End-to-end worked micro-example (design algebra)

Goal. Specify a tiny service that: *(i)* chooses a branch (⊕), *(ii)* must also satisfy a side contract (&) at the same locus, and *(iii)* may serve two clients (exponential copy).

Spec (behavioural):

`Svc := (Color ⊕ Shape) & Audit` at base `σ`, with disjoint directories for `Color/Shape` and `Audit`. □filecite□turn6file3□

ASCII — one interaction (single client).

```
E:  −σ                  # tester arrives
D:  +σ                  # design opens directory {color,shape} and {audit}
E:  −σ.color            # chooses Color
D:  +σ.color.red        # supplies a color
E:  −σ.audit            # asks for audit proof
D:  +σ.audit.ok         # satisfies audit
… ; ♦                   # normalization success ⇒ D ⊥ E
```

Copy to two clients (`σ·0`, `σ·1`).

```
D: +σ ↦ {σ·0, σ·1}     # duplicate service (fresh sub-loci)
E0 probes σ·0.color ; E1 probes σ·1.audit
D answers both coherently; if either gets stuck → non-orthogonal
```

This illustrates **⊕**, **&**, and **!** at the **address** level. □filecite□turn6file3□ □filecite□turn6file4□

9.4 Common failure modes (what to test for)

• **Undeveloped branch at a tested address** → stuck run (no `♦`) = not orthogonal. Write a test that explicitly probes every directory entry. filecite turn6file2

• **Alias in copies** (reuse `σ·0` as `σ·1`) → violates freshness; include testers that distinguish copies. filecite turn6file4

• **Non-uniform quantifier use** → behaviour depends on the actual name; vary the fresh name in the tester. filecite turn6file8

• **Forgot to delocate** when directories clash → additive composition not well-founded; add a renaming step. filecite turn6file3

9.5 Directions beyond this quickstart (very brief)

• **Repetitions/measure-theoretic variants** for modelling non-determinism or probability while preserving separation by tests.

• **Program logics & session types** from behaviours (using directories as protocol menus).

• **Bridging to proof-nets** with richer correctness criteria tied to locality and daimon. filecite turn6file5

9.6 Takeaway for Mesh contributors

• Treat a component as a **design**, its API as a **behaviour**, and testing as **counter-designs** exercising directories and copies.

• Use the **local algebra** (∩/⊕/⊗/⅋) at addresses and add **delocation** automatically when composing.

• Keep **freshness** and **uniformity** explicit in the code paths that correspond to duplication and quantification. filecite turn6file3 filecite turn6file4 filecite turn6file8