

Typed Authority Boundaries

AC and DI for Structure-Preserving Technical Workflows

W. Mitchell
Semantis.io

February 2026

Preprint — Semantis Research Series. This manuscript is shared for open review and has not yet undergone peer review.

Abstract

Technical workflows routinely mix conceptual structure, implementation detail, and presentation across documents and tools. This mixing creates failure modes: hidden authority transfers, accidental introduction of new primitives during “translation,” and silent drift between specification and output. We introduce **Authority Contracts (AC)** and **Declared Interfaces (DI)**, two typed artifact forms designed to harden boundaries between structure and rendering.

An AC artifact is an authoritative specification of a structure with explicit scope, invariants, admissible operations, and a declared primitive vocabulary. A DI artifact is a non-authoritative projection / rendering that maps an AC-defined structure into another substrate (e.g., prose, code, figures) without introducing new primitives. Here, *introducing a primitive* is operational: a DI introduces a primitive if it asserts, assumes, or depends on any symbol, rule, bound, regime, or equivalence criterion not declared (or explicitly allowlisted as a permitted derived form) by the governing AC. We provide enforceable rules, conformance manifests, and schema-validated front matter that constrain DI artifacts to remain non-authoritative and require AC artifacts to declare their contract. The result is a reproducibility-oriented method for stabilizing interfaces in documentation and toolchains while making authority boundaries explicit, checkable, and reviewable.

1 Introduction

A recurring failure mode in collaborative technical work is *boundary drift*: the artifact that is supposed to “render” a specification begins to silently modify the specification itself. This can occur when converting notes to code, code to figures, or internal documents to public writing. The drift is often unreviewable because the authority boundary is implicit: reviewers do not know which artifact is permitted to introduce structure and which is permitted only to re-express it.

AC/DI is a minimal response. It makes two artifact types explicit, assigns authority to one of them, and makes the other operationally non-authoritative by construction and tooling:

- AC defines what exists, what is admissible, and what counts as equivalence for t .
- DI renders AC-defined structure into a substrate s and must not create structure.

This manuscript is not a new logic. It is a workflow method: an authority typing discipline plus conformance checks that reduce the probability that “translations” or “exports” become silent edits to the underlying structure.

AC/DI is intentionally not a logic or verification system. It is an artifact-typing discipline that makes boundary violations mechanically detectable with minimal tooling.

2 Contribution

This paper contributes a lightweight artifact-typing discipline for preserving authority boundaries across technical workflows. Specifically, it:

1. distinguishes authoritative structure from non-authoritative rendering through the Authority Contract (AC) and Declared Interface (DI) artifact types;
2. defines primitive introduction operationally so that boundary violations can be reviewed and mechanically checked;
3. identifies common collapse modes in which renderings, translations, code, figures, or public explanations silently acquire structural authority; and
4. presents an enforceability stack based on declared manifests, schema validation, and continuous-integration gates rather than requiring a full formal-verification toolchain.

The central claim is intentionally narrow: a representation of an authoritative artifact should not acquire authority merely because it is useful, executable, readable, or widely reused.

3 Related Work

AC/DI is adjacent to several established software-engineering and security ideas. Information hiding and modular decomposition separate stable design decisions from their external representations [1]. Explicit mediation and least privilege similarly require that access and authority be checked rather than inherited implicitly [2]. Design by Contract makes interface obligations and admissible behavior explicit at software boundaries [3].

The present paper applies a narrower artifact-level distinction: an authoritative specification and a rendering of that specification are different artifact types, and the latter is not permitted to introduce new primitives. This is also compatible with reproducible-computation practice, where explicit inputs, recorded transformations, and reviewable outputs reduce silent workflow drift [4].

AC/DI does not replace these approaches. It supplies a small authority-typing layer for workflows in which prose, code, diagrams, generated artifacts, and tool-mediated translations may otherwise blur the difference between defining structure and presenting it.

4 Model and Notation

4.1 Structure and substrate variables

We distinguish:

- t : a *typed structure* governed by an AC (state, operators, invariants, regimes, observables, equivalence criteria).
- s : a *target substrate* for representation (e.g., Markdown prose, code, plots, LaTeX, diagrams).

We use the following typing schemata:

$AC\langle t \rangle$: authoritative specification of structure t .

$DI\langle t \rightarrow s \rangle$: declared rendering of t in substrate s , non-authoritative.

Authority is a property of the artifact type, not of the author or tool. The boundary rule is explicit:

- AC is authoritative for t .
- DI is not authoritative for t .

4.2 Artifact identity and governance

Every DI must name the AC that governs it (by identifier and version). In particular:

- DI is governed by exactly one AC at evaluation time.
- If multiple ACs exist, DI must declare which AC is governing (no implicit merges).

This governance rule prevents “accidental multi-authority” where a rendering quietly mixes two incompatible contracts.

5 Definitions

5.1 Primitive (AC-declared)

A **primitive** is any element of the structure definition that can change the admissible behavior or interpretation of t . AC primitives include, at minimum:

- 1) **State variables**: declared state components, domains, and units/kinds.
- 2) **Operators / update laws**: transition functions, mappings, rules, algorithms, inference operators.
- 3) **Invariants / constraints**: required properties, conservation laws, admissibility constraints.
- 4) **Parameters and bounds**: parameter names, domains, and allowed bounds or ranges.
- 5) **Regimes / events / detectors**: mode partitions, triggers, event definitions, regime labels.
- 6) **Observables**: telemetry keys, measured outputs, readouts, reporting channels.
- 7) **Equivalence criteria**: relations such as $\sim$ that define sameness or class membership, as declared by the governing AC.
- 8) **Type judgments**: declarations that assign kinds to symbols (e.g., “ x is state,” “ M is observable operator”).

An AC must explicitly declare its primitive vocabulary (Section 5).

5.2 Derived form (AC-allowed)

A **derived form** is any expression computed from AC primitives by a rule that AC explicitly permits. Derived forms are not new primitives if (and only if) they satisfy all of:

- (D1) The derivation rule is declared in AC as permitted.
- (D2) The derived form introduces no new free symbols beyond AC primitives.
- (D3) The derived form does not add new constraints, bounds, or regimes beyond those declared in AC.
- (D4) The derived form must not be used to justify any new constraint, bound, regime, or equivalence interpretation in the DI.

Derived allowlists are reviewable commitments: if a transformation changes admissibility, bounds, regimes, or equivalence, it must be elevated into AC as a primitive change (with versioning), not treated as a derived convenience.

5.3 Introducing a primitive (DI-prohibited)

A DI artifact **introduces a primitive** if it does any of the following relative to its governing AC:

- (P1) Introduces a new symbol and assigns it a structural role (state/operator/invariant/regime/equivalence/parameter bound).
- (P2) Adds or tightens a constraint or bound not declared by AC (including “reasonable defaults”).
- (P3) Defines a new regime/event/detector or redefines regime boundaries.
- (P4) Defines or modifies an equivalence criterion (e.g., defines a new $\sim$ or changes its parameters).

- (P5) Treats a derived quantity as if it were a primitive (violates D1–D4).
- (P6) Asserts “must/shall/therefore” statements about t that are not entailed by AC declarations (authority transfer in prose).
- (P7) Introduces a new operator in code or pseudocode that affects semantics rather than presentation (e.g., a filtering step that changes admissibility).

Operationally: if the DI cannot point to an AC clause that declares the relevant symbol/rule/bound/regime/equivalence (or explicitly allowlists it as a derived form), then the DI is introducing a primitive.

6 Threat Model and Failure Modes

6.1 Threat model

AC/DI defends against workflow failures where non-authoritative artifacts accidentally become authoritative, including:

- Specification rewritten during documentation, publication, or visualization.
- Tool outputs that embed hidden constraints (“plotting defaults,” “cleanup steps,” “normalizations”).
- “Helpful edits” that introduce new regimes, bounds, or equivalence interpretations.
- Multiple stakeholders “merging” contracts implicitly through prose or code.

The primary threat is routine drift under time pressure. The same boundary also helps in adversarial settings, but adversarial security is not the main claim of this paper.

6.2 Failure modes (typical)

AC/DI is designed to prevent or detect:

- **FM1 — Authority transfer:** DI prose or code asserts new requirements for t .
- **FM2 — Hidden parameter bounds:** DI introduces or tightens bounds (“assume $x \in [0, 1]$ ”) not declared in AC.
- **FM3 — Regime injection:** DI introduces a detector or mode partition absent in AC.
- **FM4 — Equivalence drift:** DI silently changes what counts as “same” (e.g., adds tolerance, changes metric).
- **FM5 — Derived-to-primitive promotion:** DI presents a computed convenience quantity as a structural entity.
- **FM6 — Multi-contract mixing:** DI combines structures from multiple contracts without declaring governance.

AC/DI does not prevent every semantic misunderstanding, but it makes these failures explicit and checkable.

Equivalence is treated as structure: it must be declared in AC, never introduced in DI.

6.3 Minimal Example (boundary violation)

- AC defines a structure with parameter $\alpha \in [0, 1]$ and equivalence $\sim_\varepsilon$.
 - DI renders it as a plot + prose.
 - DI mistakenly says: “Assume $\alpha \in [0.2, 0.8]$ for stability.”
 - Under AC/DI rules, this is a DI primitive introduction (P2: new/tightened bound).
 - Therefore DI is non-conformant and must fail CI.
-

7 Artifact Types and Required Content

7.1 Authority Contract — AC (authoritative)

An AC artifact defines the structure t and its contract: scope, non-scope, dependencies, primitive vocabulary, admissible operations, observables, equivalence criteria, and reproducibility constraints.

AC must include, at minimum, the following sections (headings or equivalent structured blocks):

- Scope / Non-scope
- Dependencies (read-only)
- Primitive vocabulary
- State
- Operators / Update laws
- Parameters (including bounds)
- Events / Regimes (detectors)
- Observables (telemetry keys)
- Equivalence criteria
- Reproducibility notes (seeds / determinism constraints)
- Versioning and change control

AC is the single authoritative source for primitives. If it is not in AC, it is not admissible for t .

7.2 Declared Interface — DI (non-authoritative)

A DI artifact renders an AC-defined structure into substrate s (text, code, visualization) without introducing primitives. DI must include:

- A governing AC identifier + version.
- The target substrate s .
- A conformance manifest:
 - used symbols (subset of AC primitives)
 - used derived forms (subset of AC allowlisted derived rules)
 - transformation class (presentation-only vs. permitted derived reporting)
- A statement of non-authority:
 - DI is not authoritative for t .
 - Disagreements resolve in favor of AC.

DI may reorder, reformat, rename labels, or plot declared observables, but it must not add structure.

8 Rules

We state the core rules as enforceable artifact constraints.

8.1 R1 — No primitive introduction in DI (core safety property)

DI must not introduce primitives as defined in Section 3.3. A DI is non-conformant if any P1–P7 condition holds.

This rule is evaluated relative to the governing AC and is therefore versioned: DI conformant to AC v1 may be non-conformant to AC v2 if primitives change.

AC/DI constrains authority transfer across artifacts; it does not attempt to prove correctness of the AC itself.

8.2 R2 — AC must declare its primitive vocabulary and allowlisted derived rules

AC must include:

- An explicit list of primitives (symbols + kinds).
- An explicit list of permitted derived rules (if any).

If AC omits this inventory, DI conformance is uncheckable and must fail validation.

8.3 R3 — Authority boundary and conflict resolution

AC is authoritative for t . DI is not authoritative for t . If DI output conflicts with AC specification, AC prevails. DI must not “patch” AC by prose or code; the patch must be made as an AC revision.

8.4 R4 — Templates referenced, not copied

Artifact shape is stabilized by referencing canonical templates (by identifier) rather than duplicating them. Copies drift; references remain governed.

9 Machine Hardening: Front Matter, Schemas, and Conformance Manifests

AC/DI is intended to be checkable by cheap tooling. The method uses:

- YAML front matter (optionally validated by JSON Schema)
- required heading checks (AC)
- conformance manifest checks (DI)
- CI gates that can fail on boundary violations

The examples below are reference schemas: projects can adopt the minimal fields immediately and incrementally enrich primitive inventories and derived allowlists as enforcement needs grow.

These headers are illustrative; projects may implement equivalent fields provided closure (AC inventories) and conformance (DI manifests) remain checkable.

9.1 Example AC front matter

The following is an example AC header. Note the explicit primitive inventory and allowlisted derived rules.

```
artifact_kind: AC
is_authoritative: true

# Bind the abstract type parameter t:
t_id: erg
t_version: 1.0

governance:
  change_control: ac_only
  conflict_resolution: AC_prevails

required_sections:
  - Scope / Non-scope
  - Dependencies (read-only)
  - Primitive vocabulary
```

```

- State
- Operators / Update laws
- Parameters (bounds)
- Events / Regimes (detectors)
- Observables (telemetry keys)
- Equivalence criteria
- Reproducibility (seeds / determinism constraints)

# Primitive inventory (operational):
primitives:
  state:
    - name: x
      kind: real_vector
    - name: theta
      kind: angle
  operators:
    - name: U
      kind: update_law
  invariants:
    - name: I1
      kind: constraint
  parameters:
    - name: alpha
      kind: scalar
      bounds: [0.0, 1.0]
  regimes:
    - name: R0
      kind: regime
      detector: D0
  observables:
    - name: y
      kind: telemetry_key
  equivalence:
    - name: "~"
      kind: equivalence_relation
      defined_in_section: Equivalence criteria

# Allowlisted derived rules (optional):
derived_allowlist:
  - name: normalize
    rule: "z := (y - mean(y))/std(y)"
    scope: presentation_only
  - name: tolerance_bucket
    rule: "bucket(y, eps)"
    scope: reporting_only

reproducibility:
  deterministic: true
  seeds_required: true

```

The following is an example DI header. Note that DI declares the governing AC and provides a conformance manifest.

```

artifact_kind: DI
is_authoritative: false

# Bind the abstract type parameters:
t_id: erg
t_version: 1.0

```

```

target_substrate: md_public

# Governing AC reference:
governing_ac:
  id: erg
  version: 1.0

# Conformance manifest (operational):
conformance:
  used_primitives:
    - x
    - theta
    - y
    - "~"
  used_derived:
    - normalize
  transformations:
    - class: presentation_only
      description: "Reformat headings and rename labels only."
  prohibited_actions_acknowledged:
    - "no new operators"
    - "no new bounds"
    - "no new regimes"
    - "no new equivalence criteria"

schema_id: erg_to_md_v1
payload_schema: "<project-defined payload schema>"

```

10 Enforceability Stack and CI Gates

AC/DI enforcement is intentionally staged: the baseline guarantee comes from inventories and manifests; scanning and extractors provide additional evidence.

10.1 Three-tier enforcement stack

1) Manifest closure (hard fail)

DI's `used_primitives` and `used_derived` must be subsets of AC's `primitives` and `derived_allowlist`. If any symbol appears in DI but not in AC inventories, the DI fails.

2) Payload scan (soft → hard)

Scan DI payloads (prose, code blocks, figure captions, math blocks, config snippets) for symbol tokens not declared in the DI manifest. Initially this can warn; for mature pipelines it can hard fail. This is the appropriate place for “term-based linting”: it is secondary evidence, not the primary guarantee.

3) Language-aware extractors (optional modules)

If the DI substrate supports it, add cheap extractors:

- code: AST-based identifier extraction
 - LaTeX/Markdown: math-block + inline-code symbol extraction
 - figures: caption/label token extraction
- These modules improve signal quality without changing the core contract.

10.2 CI gates (minimal set)

- **G1 — AC completeness:** fail if AC lacks primitive inventory, equivalence criteria, and required contract sections.
 - **G2 — DI closure:** hard fail if DI manifest is not a subset of AC inventories.
 - **G3 — No new bounds/regimes/equivalence:** fail if DI introduces bounds, regime labels, or equivalence definitions not declared by AC.
 - **G4 — Conflict resolution discipline:** if DI conflicts with AC, the remedy is an AC revision + version bump; DI must not patch the contract.
-

11 Limitations and Scope

AC/DI provides hardening, not completeness.

- AC/DI does not guarantee semantic equivalence of renderings across substrates; it guarantees only that DI does not *introduce* structure beyond AC.
 - AC/DI cannot fully detect implicit assumptions expressed without symbols (e.g., rhetorical inference). Payload scanning helps but is not proof.
 - If AC is underspecified, DI conformance can be mechanically satisfied while still yielding ambiguous interpretations. The remedy is to strengthen AC, not to strengthen DI.
-

12 Discussion

AC/DI is best understood as a typed authority boundary for technical workflows. It converts implicit authority decisions into explicit, checkable artifacts:

- AC declares admissible structure and serves as the authoritative source of primitives.
- DI is a declared rendering that is structurally non-authoritative and conformance-checked.

The method scales by incremental adoption: teams can begin by enforcing front matter and primitive inventories, then add DI manifests and CI gates as drift pressure increases. The primary value is cheap failures at review time and preventing silent structural edits from entering public artifacts.

13 Conclusion

We presented AC and DI: typed boundary artifacts that harden structure/rendering boundaries in technical workflows. AC defines authoritative structure t via an explicit contract and primitive inventory. DI renders t into substrate s while remaining non-authoritative and conformant: it must not introduce new primitives, bounds, regimes, or equivalence criteria beyond those declared (or allowlisted as derived) in AC. By combining explicit authority typing with lightweight validation and CI gates, AC/DI offers a practical, reproducibility-oriented method for preventing structure leakage across interfaces.

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.

- [2] J. H. Saltzer and M. D. Schroeder, “The Protection of Information in Computer Systems,” *Proceedings of the IEEE*, vol. 63, no. 9, pp. 1278–1308, 1975.
- [3] B. Meyer, “Applying ‘Design by Contract’,” *Computer*, vol. 25, no. 10, pp. 40–51, 1992.
- [4] G. K. Sandve, A. Nekrutenko, J. Taylor, and E. Hovig, “Ten Simple Rules for Reproducible Computational Research,” *PLoS Computational Biology*, vol. 9, no. 10, e1003285, 2013.