States & Invariants

THESIS

Order of operations — writing the small model is what improves comprehension; the checker only answers "did I miss an ordering?" so you can stop looking; TLA+ is those two things with better machinery. Tool third, skill first

1.1 VOCABULARY — CORE

fact — one stored field. In 6-synthetic: the document name, the list of converted files, the pending-detection flag — each is one fact

state — the value of every fact at one moment. One state of 6-synthetic = name + file list + job flags + waits, all at once. The explorer's 525,976 is a count of distinct states

derived condition — a calculation over facts, stored nowhere, recomputed on every look. Example: archive up to date = converting == 0 AND archive != null AND archive == the converted files. Neither a fact nor a state

atomic step — the largest chunk of code that runs with no other actor's write landing inside it. Between two atomic steps, anything can happen. In single-threaded async JS, the code between two awaits is one atomic step — every await is a preemption point

interleaving — one possible ordering of everyone's atomic steps. Two actors with 2 steps each have 6 interleavings: 4! / (2! * 2!) = 24 / 4 = 6

next-state relation — the rule saying, for any state, which states can come next — one entry per event or step allowed in that state. Refusals belong here too: an event that errors is a transition to a state where the error fact is set

reachable state — a state some interleaving can actually produce starting from the start state. Most describable states are unreachable; the explorer counts only reachable ones

state space / reachability graph — all reachable states as nodes, steps as arrows. This is what the explorer computes but does not draw

state space explosion — the count grows multiplicatively with actors and steps. Calibration: the six-event synthetic app had 6,357 reachable situations; the nine-event version 6.2 million; the first attempt blew an 8GB heap

invariant (safety) — a claim that must hold in every reachable state — "nothing bad is ever true". Checkable by looking at each state alone. All 16 rows in the 6-synthetic README table are safety invariants

liveness — a claim that something good eventually happens — "no wait hangs forever". Cannot be checked by looking at one state; needs the run's future. The tally's stuck detector is a liveness check per-state invariants cannot express

fairness — the assumption that a step which stays possible eventually runs. Safety needs no such assumption; liveness does — without it "the job eventually finishes" is false because the scheduler could just never run it

deadlock / stuck state — a reachable state with no possible next step while someone is still waiting. The 8-of-300 hung tally runs ended in one: a retrieval waiting for a summary that could never exist

wedged vs failed — a failed run reached a terminal error everyone can see; a wedged run stays RUNNING forever — a task on infinite retry, a wait on a dead condition. Monitoring that only watches for FAILED misses the wedged class entirely

race condition vs data race — a data race is two threads touching one memory cell unsynchronized, impossible in single-threaded JS. A race condition is order-dependence of the outcome — fully alive in async JS because awaits interleave handlers

logical race — a domain invariant that quietly breaks under an interleaving nobody exercised. Lives in the orderings no test suite samples, so it passes every test and fails in production

check-then-act (TOCTOU) — read a fact, decide, then write — with a gap where another actor's write can land. This write lands between that check and its consequence. In async code the searchable name is atomicity violation across an await

weld — a check and its consequent write with no gap between them — one atomic step. The two named welds in the timelines: wait-exit joined to serve, snapshot-compare joined to write

reentrancy — a reentrant actor starts a new handler before the previous one finishes (they interleave at awaits); non-reentrant means one message runs to completion first. An intra-workflow race requires interleaving, so reentrancy is the enabling condition

snapshot — a copy of the facts a job read at its start, compared at the end; if the live facts changed, the result is discarded and the job reruns. The 6-synthetic background-work rule

cut — a point where timelines wait until each has finished a phase — a barrier. 6-synthetic rests on one real cut, converting == 0, reused by four consumers: one barrier, four users

quiescence / settled — a definition of "nothing is running", needed before any when-idle claim can be checked. Two defensible forms: no busy flag is true, or the doc is deleted or submitted with archive and summary up to date (derived conditions instead of flags)

terminal / absorbing state — a state no transition leaves; any event arriving there is refused. Two kinds of end seen in the real workflow: flag ends that refuse with an error, and a silence end (10-minute idle) that refuses nothing

command vs event — a command is an intent that can be refused (submit, delete — the transition validates and may throw); an event is a past-tense fact you cannot refuse (fileConverted)

confluence — orders don't matter: every ordering of the same events ends in the same state. Non-confluence is the measurable symptom of the whole disease. "Make the order stop mattering" = restore confluence

feature interaction problem — each rule correct alone, wrong in combination; the textbook case is a heater rule and a cooler rule sharing one room. Sibling names for the style: ECA rules (event-condition-action), implicit invocation, trigger cascades

linearization point — the instant at which a step's effect becomes visible to everyone — one atomic transition of the high-level machine. Practical rule: log an event exactly where shared state is updated

run-once flag — a busy flag (archiving, detectingName) prevents a second copy of the same job — it collapses re-entries, not interleavings. The belief that a busy flag makes something safe confuses the two

BFS (breadth-first search) — visit states level by level outward from the start state. The first violation found is at the smallest number of steps, so counterexamples come out shortest

visited set — the set of states already seen, so each state is explored once. States must be serialized to a comparable form (a string) to live in it. The 8GB blowup was this set holding full objects

explicit-state model checking — enumerate every reachable state via the next-state relation, check safety invariants at each. What TLA+'s checker TLC does. Knowing the name unlocks the literature

1.2 VOCABULARY — HAZARDS AND EVIDENCE

one-store vs two-store hazard — one-store: correctness depends only on the workflow's own bookkeeping — tractable. Two-store: correctness depends on staying consistent with a second stateful entity it does not control — the unsolved hard core. Every tool surveyed catches one-store and misses two-store

convention-held vs structurally enforced — an invariant held by convention exists only because every handler happens to do the same lines in the same order. Tests: can it be stated as a property of one component? If a new contributor did the naive thing, would it survive?

provenance grades — attested (a ticket says it happened), structural (analysis shows the ordering reachable), reachable-prod (a recorded history shows the ordering), observed-broken (a history shows the violation), design-intent, un-exercised. Rule: un-exercised is not safe — it is no information

a dangerous ordering is not a broken invariant — the config-toggle race was proven reachable in a real production history, and the outcome that run happened to be harmless. Reachability says the gap exists; violation says something fell in

un-Googleable bugs — the public record corroborates every framework-primitive hazard and goes silent on every invariant-specific one, because a bug defined relative to one workflow's own invariant cannot be searched for. This is why you enumerate your own states

determinism is not race-freedom — replay determinism (same history in, same code path out) says nothing about which orderings of external events are possible. The remaining nondeterminism is exactly the arrival order of signals, completions, cancellations — finite and enumerable

idempotency — an operation safe to run more than once: f(f(x)) == f(x). Under at-least-once delivery every side effect needs it, usually via an idempotency key — stable (never generated inside the retried step) and enforced downstream (in-memory dedup is lost on restart)

property vs invariant — an invariant is one kind of property: always true of every state of a run. Other checkable shapes: roundtrip (decode(encode(x)) == x), metamorphic (know how the answer must change, not the answer), model-based, commutativity — a commutativity failure is a race by definition

oracle (weak vs semantic) — the pass/fail judge. A fuzzer's oracle is weak — crash, hang. A property test's is semantic — it knows what right means. Ordering bugs are silent (wrong counts, lost updates), so a crash-hunting oracle sails past them

interleaving surface — a computable number: handlers times await points per handler; the only atomic regions are await-free statement runs. The real workflow: about 30 update handlers on one coroutine; 4 of 10 recorded histories show a second handler inside another's span

1.3 THE SPEC DISCIPLINE (TLA+ IDEAS, TOOL-FREE)

behavior — one run written as a sequence of states: an initial state, then each adjacent pair connected by one allowed step. The explorer's shortest trace is a behavior. A spec's whole meaning is the set of behaviors it allows

spec = init + next — a complete spec is exactly two things: which states you may start in and which steps are allowed. Invariants and liveness are claims ABOUT the spec, not part of it. This is why no implementation is needed to have a checkable spec

constants vs variables — variables change during a run (files, name, waits); constants parameterize the model (max 2 files, the set of possible names). Checking always runs at small constants — small counterexamples make that enough for hunting

primed and unprimed — an allowed step is a predicate over a pair of states: unprimed x is the value before, primed x' the value after. "converting = 0" is a guard; "archive' = converted files" is the write. One line can state both — which is a weld, written down

action, guard, enabled — an action is a guard plus an effect on the (before, after) pair; it is enabled in a state when some next state satisfies it. Blocking needs no scheduler: a task in a false wait is simply not enabled

nondeterminism is choice, not randomness — a spec allowing several next states says any of these may happen, not one is picked with some probability. The checker takes every branch; the tally rolls dice on one

stuttering — a step in which nothing the spec watches changes; a spec must tolerate them. Consequence: "eventually" claims are false under infinite stuttering unless fairness forbids it — the formal reason liveness and fairness always travel together

the type invariant comes first — the boring invariant: every variable holds a value of its intended shape (converting is between 0 and the file count). It catches mistakes in the MODEL before the model can catch mistakes in the system

always / eventually / leads-to — always P: P in every state of every behavior (safety). Eventually P: P in some state of every behavior (liveness). P leads-to Q: whenever P becomes true, Q eventually follows — "every request ends" is requested leads-to ended

weak vs strong fairness — weak: a step that stays enabled from some point on eventually runs. Strong: a step enabled again and again, even with gaps, eventually runs. Every liveness verdict silently rests on one of the two

inductive invariant — an invariant where any state satisfying it can only step to states satisfying it: start-check plus one-step-check then proves it for all behaviors, no enumeration — the only route to an unbounded guarantee. Most useful invariants are true but not inductive

refinement — every behavior of a detailed spec maps to a behavior of an abstract one, extra steps becoming stutters. The formal version of "the code matches the model"; trace validation is its sampled cousin

symmetry — states differing only by swapping interchangeable values (file A for file B) are one state for checking. The principled name for what interning and receipts-merging did by hand

state claim, step claim, behavior claim — the three shapes a property can take: about one state, about one step (a before-after pair), about a whole run. They map one-to-one onto the explorer's hooks

2. HAND ENUMERATION

list the atomic steps of an actor — worker does: read x, then write x+1. That is 2 atomic steps if separate, 1 if welded. For async code the split rule is: cut at every await, nowhere else

count interleavings of two actors — actor A has a steps, actor B has b steps: (a+b)! / (a! * b!). 2 and 2 steps: 24 / 4 = 6. 3 and 3: 720 / 36 = 20. 5 and 5: 252

enumerate reachable end states — x starts 0; workers A and B each do read x, write x+1 (2 steps each, not welded). Reachable final x: 1 or 2. 6 interleavings, only 2 end states — paths merge

why states are fewer than interleavings — different orderings that produce identical fact values are one state. Facts that influence nothing after being written multiply interleavings without multiplying meaningful states

sampling vs enumeration, with numbers — the same ~100-line six-event app: 300 random-timing runs surfaced about 24 distinct endings; exhaustive enumeration found 6,357 situations and 54 endings. Sampling finds the common endings, never the rare ones — the top ending took 41 of 300 runs, most interesting ones 2-4

small counterexamples, huge proofs — counterexamples show up at tiny bounds (2 agents, 2 cells, 4 operations — witnesses of 2-5 states); proving safety at 3 agents and 9 operations explored over 150 million states. Hunt small; only proofs need the full space

write a next-state table from prose rules — a door: open or closed; push toggles; lock only when closed; push on a locked door errors. Produce the full (state, event) -> next-state table including the refusal row. Closure check: every fact a row mentions is in the state list, every state item has a writer and a reader

find the violating state by hand, then verify by enumeration — claim "x always ends at 2" with the 2-step workers: find the counterexample ordering by hand (A reads, B reads, A writes, B writes), then confirm by listing all 6 interleavings. A hand argument is a guess until enumerated

map the shared surface before enumerating — per shared thing: who writes, who reads, serialized under what, at which await another handler can land, the intended invariant. In the real workflow this took 40 rows down to the 5 handlers that matter. Modeling before that map is how you miss the elephant

fan-in count per fact — count the writers of each fact: in 6-synthetic, files 4, deleted 3, name 3, archive 2, summary 2, converting 1. A fact with many writers is where orders collide — the cheapest risk metric there is

grep for convention-held rules — wide-scope let (shared mutable state); the same call repeated N times (a remember-to-X discipline); boolean flags (illegal-combination candidates); must, should, ensure, before, after, always, never in comments — a rule written as a comment is a rule not enforced by structure

spot ordering preconditions — every runtime guard asking "did an earlier step happen?" is a confession of an unenforced ordering rule: precondition throws, non-null assertions on late-populated fields, readiness flags, comments saying must be called after

diff the endings after a rule change — re-run the enumeration after each new rule and compare ending sets. In 6-synthetic, "a failed conversion removes its file" cured a stuck-forever ending and created a document named with the empty string. A rule's real effect is the diff, not the intent

3. BAD-ORDERING CATALOG

the five symptoms — hand-made gates multiplying, one per surprise; several variables all meaning "what is current", updated at different moments; steps of one action reading live state between its pauses; whichever response lands last wins; refused or missed events vanishing silently

lost update — two actors read the same fact, both write back a value computed from their read; the second write erases the first. The read x / write x+1 example ending at 1

check-then-act gap — a guard passes, then the world changes before the guarded action runs. The old upload door was one of these dressed as a rule

check straddles an await — the decision is split across a pause: two near-simultaneous final signatures each set their own signed flag, then both evaluate "nobody left unsigned" after an intervening await — both see true, the action fires twice. Reproduced in 3-sign-flow

terminal flag set after the terminal action — await the terminal activity, then set the flag: a handler arriving during that await passes the is-terminal check and runs. The mirror image of a weld — flag and action must not be split by an await

stale snapshot write — a job computes from inputs, the inputs change mid-job, the job writes anyway — a result from a world that no longer exists. The 6-synthetic discard-and-rerun rule exists to kill exactly this. Same shape in a UI: refresh workers overwriting shared page slots with no currency check

guard blind to an in-flight phase — a destructive guard counts only work not yet started: delete-when-empty read pending uploads but not uploads already moving, so the document is deleted under a live upload. A destructive guard must cover pending AND in-flight AND committed

gate on one clock, act on another — several variables all mean "what is current", updated at different moments — xplace had three. Most races there were one module gating on one clock and acting on another

slower loader wins — two overlapping loads swap the shared slot only on completion, so whichever loads slower lands last and wins — the newest intent loses to the stalest response. About 11 such spots in xplace; the cure is a generation counter

wait that can never end — a wait whose condition is unreachable from the current state. The submit-empty-doc hang: retrieval waiting for a summary on a name-less submitted doc — no path could ever produce one

wait that ends on a dead premise — the wait does end, on a changed world. Submit waited for running updates and one was a mid-flight delete of the whole document: submit then submitted a deleted document, two terminal ends at once

self-referential wait — a handler drains a counter it is itself counted in. Convert could not wait for running updates because convert was tracked as one — so it skipped the drain submit performs and finalized over live mutations. A handler that drains must not itself be tracked

exit-check ordering — a wait loop checks its exit condition before checking whether its subject still exists. The refuted ALWAYS: retrieval served the summary of a doc that had just auto-deleted, because the exit check ran first

one-shot budget burned before the switch is read — a run-exactly-once counter increments when the trigger fires, before the on-off switch is looked at, never refunded on cancel — one cancelled attempt and "once" silently becomes "never"

capture-at-creation config — a job reads a switch once at start; flipping it later governs only the next start, not the running one. The one hazard proven reachable in a recorded production history

final sweep vs late registration — the closing sweep cancels only what is already registered; work that registers after the sweep passes survives the end. Sibling: waiting for handler functions but not for the fire-and-forget promises they started

default overwrites a decision — automatic behavior lands after a manual one and silently wins. The default name must never apply over a manual rename, pending or running

failure that looks exactly like success — two causes, one indistinguishable ending: a final action that fails after all retries leaves the flag set and the workflow closes looking identical to a normal quiet end. Sibling: an interrupted conversion returning a normal-looking success for a file never saved

silent drop — a refused or missed event vanishes with no trace: a contract signal with no handler registered; an over-capacity add that returns void instead of erroring

crash between the effect and the commit — the step ran but its completion was not recorded, so the retry runs the side effect twice. Why exactly-once needs an idempotency key or two-phase design

lease-expiry double delivery — a visibility timeout is a short-lived lock: near its end, two consumers can legitimately hold the same message. The honest move is a test that expects the duplicate

gate hygiene smells — a lock released by a 50ms timer (leaks locked-forever on error); a flag set and cleared but read by no one; four mutexes in one layer refusing three different ways (null, rejection, undefined) so callers cannot tell refused from empty

serializing everything can deadlock — submit waits for other handlers to finish, so if submit held a global lock nothing could run to release it. A guard protects entry, not work already inside — front guards refuse new timelines, never cut in-flight ones

4. INVARIANT-WRITING PATTERNS

contents provenance — every element of a derived artifact came from an allowed source. "Only converted files are ever archived"; "the summary contains converted files only"

existence dependency — one field implies another, the cheapest class to check and the easiest to forget. "A summary exists only when the doc has a name"

one-shot with a rerun carve-out — bound the settling, not the attempts. "Name detection settles at most once ever — reruns over changed files are part of the same settling." Getting the carve-out wrong is how once becomes never

precedence between writers — phrase who wins as "B's result never lands on top of A's", directly checkable on a transition. "A manual rename is never overwritten by a detection result"

authorized writers only — name the only hands allowed to change a fact. "Once the doc has a name, the name changes only by manual rename or by name detection"

trigger completeness — the consequence must fire, encoded as the negation being impossible. "A doc that loses its last file is deleted" checked as: no reachable state with zero files, not deleted, at least one removed file

terminal cleanliness — one state predicate listing everything that must be gone. "A deleted doc has no files, no archive and no summary"

no zombie work after a terminal — "deleting the document discards all deferred work"

frozen after terminal, with the explicit exception — write the exception into the invariant or it is quietly false. "After submit, the files and the name never change (an already-running recompute or rebuild may still write its result)"

quiescent consistency — the honest form of "always consistent" for background-computed things. "When nothing is running, the archive matches the files" — the clause is what makes it true and checkable

serve-time freshness — checked on the serving move, not on a state. "A retrieved summary matches the doc including the outcome of any conversion running when it was requested" — forbids answering before an in-flight change resolves

non-blocking guarantee — availability stated as an invariant: a step is always enabled. "An upload never waits"

liveness stated plainly — "every user request ends — no wait waits forever"

no silent give-up — "a user wait that ends without its result ends with an error — never a silent give-up." One sentence that forbids the whole silent-drop family

5. DESIGN MOVES THAT SHRINK THE SPACE

one worker, fixed priority — funnel independent jobs into one owner whose loop picks work in a fixed order. Measured: 525,976 states vs 6,209,453 (11.8x fewer), same 16 ALWAYS. The price: background time becomes the sum of jobs instead of the max, and a downloading user waits behind name detection they don't care about

one place that decides — one owner holds all the facts; every event passes through one door, one at a time; deciding is one pure function (state, event) -> (new state, actions). Four names: actor model, Elm architecture / Redux reducer, decide/evolve, statecharts. It cures only if it owns ALL the facts and ALL events pass through — a machine only some events pass is just gate number twenty

the queue orders deciding, not doing — the decide function is microseconds, so serializing it costs nothing; slow work still runs in parallel as actions. Semaphores and queues alone only say wait, not now; they never decide what should happen

no dirty flag — compare instead of notify — each job compares its result against the current facts instead of being told something changed; nobody has to remember to notify anyone. In the rewrite the old hottest fact (a stale flag with 6 writers) became a compare and ceased to exist

order-insensitive state — state whose merge is commutative, associative and idempotent converges regardless of arrival order — there is no wrong order to protect against. Does not fit genuinely sequence-dependent logic like terminal transitions

one cut, reused — when coordination is unavoidable, prefer one barrier many consumers share over many private gates. 6-synthetic rests on a single cut (converting == 0) gating four consumers

owner and choke point — unowned shared state gets a tiny stateful owner holding the value private and exposing only allowed operations; a choke point is one wrapper every mutation must pass through, so a required follow-up cannot be skipped

structured concurrency — a task scope cannot exit until all children finish, which makes drain barriers structural instead of remembered. Fire-and-forget spawning is a form of goto: control leaves while work it started still runs

generation counter / fencing token — version every derived output and accept only the latest — the structural cure for slower-loader-wins. The SQS receipt handle is the same idea as a right-to-act: only the exact handle from this read may delete or extend this message

declared non-guarantees, tested on purpose — write the test that expects the duplicate delivery and the out-of-order arrival, so nobody later "fixes" behavior the platform never promised

the litmus for needing the tool — the moment you write your second hand-made flag, lock, or run-once check around the same thing. The trigger underneath: a process stays open, other events land in the middle, and your code must decide what happens then

duration is the window — with millisecond operations the gap is so narrow that flawed code runs silently for years; 10-30 second calls surface the same disease; a workflow stretches it to minutes

6. BUILD THE CHECKER (THE OWNERSHIP LADDER)

map the shared surface of a small system — per shared thing: writers, readers, serialization, interleave points, intended invariant. This decides what deserves modeling before any modeling happens

draw the trigger map — every arrow from a write to a condition it feeds, plus the chains; per fact, its fan-in. The use: I see this condition — who can change its answer behind my back?

draw one full reachability graph on paper — the door system (3 states) or the 2-worker counter (about 9 states): every state a box, every step an arrow, refusals included. Predict the state count before drawing

write the stories first — before running any checker, write down the bad orderings you already believe in as concrete scenarios; the machine must rediscover every one (the 1-document-draft explorer found all 5, plus one new). A checker that cannot find what you already know proves nothing

write a minimal explorer for the paper system — roughly 50 lines: a queue, a visited set of serialized states, a step function returning successors, one invariant per state. The test: its count equals the paper graph's

serialize states into canonical keys — one comparable string per state, fields in fixed order, small codes; intern unbounded values into integers. Leave out fields no guard ever reads — states differing only there merge and the graph shrinks without losing correctness

take both branches of every uncertain outcome — where the real code rolls dice (a conversion fails 15% of the time), the explorer emits both success and failure moves into the same search. Explore both vs sample one is the explorer-vs-test distinction in one line

model blocking as no enabled move — a task parked in a false wait offers no successor; no scheduler, no queue — enabledness is the wait. Stuck detection falls out: no enabled moves and not an ending

add endings and stuck detection — an ending = all user tasks done and all background work idle; stuck = no successors while some actor waits

keep shortest traces via parent pointers — store parent and label per discovered state; walk back to reconstruct. BFS makes the first arrival a shortest path, so every violation comes with a minimal trace for free. Keep only the first witness per invariant

put each invariant on the right hook — on a state (field predicates), on a move (pre-state, post-state, label — who changed what, what was served), on an ending only (quiescent consistency). Deciding the hook is most of the work of writing it down

refute a hand claim with it — write an ALWAYS claim believed true, run the explorer, read the counterexample path. If the claim survives, tighten it until one falls

check the model against the real code — run the real code many times with random timings and require every sampled ending to appear in the enumerated list. The caveats stay: the model is a copy of an understanding, and the real code has more pause points than the model — the real space is bigger, never smaller

fail honestly at scale — a checker that hits its size cap must say too big, never no violations found. Fix the workload when comparing designs: both versions measured on the identical 9-event run

re-derive a 6-synthetic number — model a reduced 6-synthetic (2 files, upload + convert + rename only, no jobs) in the own explorer; predict the order of magnitude first, then compare against 3-ordering-explorer

point the explorer at rules instead of code — the discovery variant: the same machine over a rules model with UNDECIDED transitions, no implementation existing. The deliverable is not a verdict list — it is a question list

7. REQUIREMENTS DISCOVERY (THE INTERROGATION INSTRUMENT)

the reframe — the enumeration machinery is not a verifier run after the spec is done; it is the requirements-discovery instrument. The requirements you cannot think of live in combinations of rules (the feature interaction problem), and combinations are what a head cannot enumerate and a checker can

a spec needs no code — facts, events, and a draft of the rules as a next-state relation are enough to enumerate. Precedent: Amazon wrote TLA+ specs at design time and found a 35-step bug in DynamoDB's replication design before implementation

UNDECIDED as a transition value — where a rule is not yet decided, do not guess: mark the transition UNDECIDED. The search returns every reachable (state, event) pair that hits the mark, each with a shortest trace — an unfillable decide-page cell, machine-generated and guaranteed reachable

the machine proposes, the owner decides — a trace ending in UNDECIDED reads as the corner-case question an engineer asks a product owner: upload, conversion fails, rename lands, delete arrives — what should happen? The instrument mechanizes the proposing so the human only decides

the convergence loop — seed the rules, enumerate, collect the UNDECIDED questions, read the new endings, decide each answer, write it as a rule, re-run — until the undecided set is empty and the spec settles

endings are an anomaly feed — even with nothing marked UNDECIDED, read every distinct ending and ask: is this one intended? The empty-string-named document and the submitted-and-deleted double ending were both found exactly this way

what it cannot do — it cannot invent the first draft (you seed facts, events, starting rules), and it only asks about what is in the model: a missing event is invisible to it. State explosion, the part a head cannot do, is precisely the part it does

8. TOOLS MAP

TLA+ / PlusCal / TLC — did I miss an ordering, by exhaustive enumeration of a hand-written model. No expressiveness ceiling; the three costs: manual modeling, state explosion, the model-vs-code gap

Quint / Apalache — same question, newer surface: types and a simulator, symbolic backend. TLC bounds the data, Apalache bounds the depth; an unbounded guarantee needs an inductive invariant, which is expert work

P language — the closest mainstream tool to async handlers plus invariants (used on S3 and EBS), but its own language and no TS bridge. The gap is translation, not capability

Coyote / Shuttle / Loom — explore interleavings of the REAL code by controlling the scheduler; right model, wrong ecosystems (.NET, Rust). Loom is exhaustive-and-small like TLC; Shuttle is randomized-and-scalable like DST. The named missing tool: a TypeScript Shuttle

deterministic simulation testing — real code under an injectable scheduler and virtual time, reproducible by seed. Samples: finds races, never proves absence. Temporal is deterministic by construction and ships no such harness; merging the 6-synthetic tally and explorer would be exactly this

fast-check (PBT) — properties over generated inputs, shrinking to a minimal counterexample, seed replay; stateful upgrades: model-based commands and a scheduler arbitrary. The hard limit: it randomizes client dispatch order and cannot inject itself between a workflow's own awaits

statecharts / XState — make illegal flag combinations unrepresentable, today, in TS. Does not touch unbounded collections or cross-store drift. A rewrite style measurable by the same explorer, not a comprehension artifact for this code

workflow-net soundness — Petri-net form where concurrency is native; soundness (option to complete, proper completion, no dead transitions) is decidable and tooled. Control flow only — adding data makes it undecidable

Jepsen / Elle — is the STORE's consistency what it claims, inferred from client observations against a fixed anomaly model. Adjacent, not on target: it tests the store, not the workflow-store interaction

Daikon — guesses likely data invariants from traces: unsound, coverage-bound, not about ordering

trace validation — the model-to-code bridge: log an event at each linearization point, let the checker verify the recorded trace is a behavior of the spec. Found discrepancies in every program tried; finds drift, does not prove conformance

Specula (LLM to TLA+ to TLC) — machine-written model from the code, then TLC. Blind runs on both case studies: caught the one-store hazards, missed every two-store one, zero false positives

the frame over all of it — testing samples orderings and finds bugs; verification enumerates and proves absence; example tests sample zero interleavings