States & Invariants

THE METHOD

Two spec pages define the system: first the variables page (every variable, bounded, with a starting value, plus the messages), then the actions page (a guard and an effect per message). A claims page states the invariants and liveness promises. Two runnable models prove the claims. The production code is a translation of the actions, and a test harness proves the code matches the model. Every change starts at the spec pages and flows down

The running example in every row below — a document system: files f1/f2/f3 that upload and convert, a name (n1 or n2) set by rename or by background detection, an archive and a summary built by background jobs, a status (new / submitted / deleted), and three waits a user can hold (download archive, get summary, submit)

1.1 WRITING THE SPEC

variable — one stored field with bounded values and a starting value. The document system has 13: name, the three files, archive, summary, three job flags, status, three requested flags. We used to say fact

state — the value of all 13 variables at one moment. The checker counted 283,951 reachable ones

state predicate — a yes/no computed from variables, stored nowhere. The three here: nameDetectingNeeded, fileArchivingNeeded, summaryGeneratingNeeded. We used to say derived condition

message — the thing that arrives: upload f1, conversion finished f2. The document system has 11 commands and 16 events; with f expanded to the three files that is 31 concrete actions

action — a message's rule: guard plus effect. The whole model is just the list of all 31. We used to say rule

guard — when the action may happen. Reads only state — variables directly or through a predicate, nothing else

effect — what the action writes — only variables, never "sends" anything. A follow-up happens because the new state enables another action's guard

command vs event — a command is asked by a caller and can be refused (guard false = the caller gets an error). An event is reported by the world and may have to wait (guard false = not yet, nobody is told)

waitable command = two actions — the command records a requested flag; a completing event fires when the wait is over. That is where submit completed, archive downloaded, summary retrieved come from

need vs permission — need lives in the predicate: "the result is wrong". Permission lives in the guard: "not now, later yes". "No file converting" moved out of all three needed-predicates for this reason; "status is new" stayed in — it means never, not wait

parameterized message — conversion failed f is one line in the spec and three concrete actions in the model — one per file

effect branches and computes — an effect may branch on the old state (delete f cancels jobs only if f was converted) and may compute from the current state instead of listing cases (archive finished writes "each file: is it converted")

bound everything / small scope — names limited to n1/n2, files a fixed record f1/f2/f3 — an array of files creates fake distinct states. If a race exists, it shows at 2-3 of anything; 10 files finds nothing new. Pick values so traces read: detection always finishes with n1, so n1 reads as automatic and n2 as manual

why f3 exists — with two files, "two converted AND one still converting" is unreachable — exactly the state where "detection waits for conversions" does anything. The test for a bound: can every rule still be exercised

store provenance — the summary records fromName, the name it was built from. Without it the stale-summary race cannot even be written down

derive, don't store — anything computable from other variables (needed, waiting) is a predicate, not a variable. But a request someone waits on needs a stored requested flag

split values the rules treat differently — absent vs removed: document emptied needs at least one removed file, so a doc that never got files stays while a doc whose last file was removed auto-deletes

an async job is two actions — a start event sets the job flag, a finish event requires it. Between the two, every other enabled action may run — that gap is where the bugs live

the conflict rule — two jobs conflict when one writes a variable the other reads (or both write it). Summary reads the name, detection writes it — serialized: summary never starts while detection runs. Archive reads only files, detection writes only the name — parallel is safe. A who-reads-what table gives the verdict; no enumeration needed

eager cancel — two ways to handle stale work: lazy (let the job finish, compare at the end, discard if stale) or eager — chosen here: every write that changes what a running job reads also clears the job's flag, in the same effect — atomic, no gap. Payoff: a job that reaches its finish read fresh state the whole way, so no stored snapshots anywhere. Refined by the liveness check: rename to the name the doc already has cancels nothing

a failed job just clears its flag — the outside machinery can fail (converter down, builder down). The failed event only clears the job flag; the needed predicate is still true, so the job restarts — same shape as a cancel. No error variable: nobody waits on a job directly. The price shows in liveness, hence "a job does not fail forever" as strong fairness

serve on the result, not the job flag — "fileArchiving is false" is true in the gap between a cancel and the restart. The serve guards compare the artifact itself: archive matches converted files; summary matches current name + converted files

refuse up front what could hang — a wait covers only work already in flight, never "maybe a future upload fixes it". download archive refuses when nothing is converting or converted; get summary refuses a name-less doc

close a hang with an event, not a guess — upload one file, submit: completing needs a name, detection needs two converted files, nothing else brings a name — the wait hangs on future user action, which the refuse-up-front rule forbids. The fix was a new event: default name applied lends the converted file's name; detection may later overwrite it

a terminal state freezes staleness — no job ever runs on a submitted doc, but download and get summary stay allowed there — so submit completed also waits for the archive and the summary to match, or a later request would wait forever. Found by walking the spec, before any tool ran

no silent give-up — delete document releases all three requested flags with an error. A wait never just disappears

Considerations — a section of the spec itself holding why each non-obvious guard and effect is the way it is — so the guards don't look arbitrary in a month

1.2 CHECKING

state machine / transition system — what the two spec pages are: variables, guards, actions. Not an FSM — an FSM has a handful of named states and no variables; here states are combinations of 13 variables, which is why a checker walks them instead of a drawing

model checking — enumerate every reachable state, check the claims at each, answer every failure with a trace. Writing the spec is the first half of formal methods; this is the verification half

the checker loop (BFS) — a queue seeded with the start state, a visited set of stringified states, mark visited at push time (at dequeue, duplicates flood the queue), a forward read index instead of shift. ~80 lines of machinery; everything else is the model

ending vs stuck — both are dead ends (no enabled action). Nobody waiting = ending; a requested flag true = stuck. The document system: 1 ending, 0 stuck

invariant (safety) — one boolean function of a state, checked in every reachable state. The "if condition, then claim" inside is just how most useful ones read. The document system has 15; all hold

counterexample — the shortest trace to a violating state, rebuilt by walking a parent map back from it; BFS finds states by fewest steps, so the first hit is shortest. An empty trace means the start state itself violates

refuted belief — a false invariant kept in the model on purpose because it documents a real gap. Example: "when no job runs and no file converts, the archive matches the converted files" — false in the gap between a cancel and the restart. Each must keep failing; one that stops failing prints a warning: the model changed

guard-enforced vs emergent — a claim a guard makes unbreakable is a worthless invariant — it cannot fail. The valuable kind emerges from several rules working together and could fail if one of them changed

invariant fishing passes — where the 15 came from: what must hold in each terminal status; provenance matches at quiet time; what each flag promises about other variables; the claim each deliberate guard clause protects; every guarantee Considerations defends

reachability question — "can we ever get here?" asked by claiming the state is unreachable and reading the trace that disproves it. Same machinery as invariants, pointed at curiosity

liveness — invariants are checked on photos; liveness is about the movie — every photo can be fine while the user is never served. Its counterexample is a loop in the state graph, not a state

fairness — an assumption, not a claim: an action that stays ready eventually fires. Without it every liveness claim fails on the nonsense run where the system never acts. Commands get no fairness — the user owes nothing

weak vs strong fairness — weak: enabled from some point on without interruption — eventually fires (job starts, serves). Strong: keeps becoming enabled, even with gaps — eventually fires (job finishes: "a job does not fail forever"). The failure events get no fairness — nothing may force one

assumptions vs obligations — both are fairness grants in the model, split on purpose. Assumptions: outside machinery we trust (the converter answers, detection returns). Obligations: promises our own code must keep (needed jobs start, satisfiable waits get served)

the three ways out of a liveness violation — accept it as a product decision and record the verdict ("uploads extend the wait"); change the spec (the fix found here: rename to the doc's current name must cancel nothing); or shrink the claim to the promise actually kept. A permanently red check trains people to ignore red — checks assert the truth, prose records the ambition declined

the quiet-user move — the shrink done here: "every wait ends" is false by design (endless uploads postpone the serve), so the kept promise is "once the user goes quiet — from some point on, files and name never change — every wait ends". Proving it exposed the same-name-rename bug

diameter — the deepest BFS level of the state graph: 25 here. Every reachable state sits at most that many steps from the start, so it is the exact bound that makes a bounded verify exhaustive — deeper adds nothing, shallower misses states

two independent translations — a TypeScript model and a Quint model, translated separately from the same spec pages — each is a check on the other. The cross-check: 200 random Quint traces, every transition must be explained by some TypeScript action's guard and effect

what the checker can never do — liveness (needs the movie: cycle detection plus fairness — the hand-off to a standard tool like Quint); anything about the real code; anything not written down — an unmodeled message does not exist for it

1.3 FROM SPEC TO CODE

atomic action in code — every model action is one synchronous block: no await between a check and its write. Single-threaded JS makes such a block atomic for free

the four-section rule — state private; reads = the predicates word for word; named sync actions as the only writers; an async layer that never writes and only calls actions between awaits. Then an await inside a decision cannot compile

pump — the event guards as one loop that re-fires after every state change — "a follow-up happens because the new state enables another guard", as code. It adds one thing the model doesn't have: a priority order (the model says any enabled action may fire; the code must pick one)

waiter queue — the model's one requested boolean becomes a list of pending promise callers. Serve = resolve one; delete = reject all — the no-silent-give-up rule as code

background work is tracked, never orphaned — never `void` an async call; an async function handles its own failure inside, then is called bare. Every spawned job lives in a tracked set with a when-idle that awaits them all — tests and shutdown can always drain to quiet

generation counter — code cannot un-schedule a promise, so cancel = flag false plus a counter bump; a completion carrying a stale counter is a cancelled run arriving late and writes nothing. A structured-concurrency library can delete the counters: cancel = halt the task, and a halted task physically cannot deliver a late result

conformance — proof the code matches the model: step both through the same trace, compare full state after every step. The model is the oracle — the thing that says what correct is

fake env — every slow call (convert, detect, build) is a stored resolver the test finishes by hand — so "conversion finished f1" is a test step, not luck

refusal probe — drive a command exactly when the model refuses it; the code must answer with an error — a throw, or a rejected promise for the waiting commands

model-based testing — random command sequences driven against model and code together (fast-check); a failure shrinks to a minimal sequence with a replayable seed. It immediately found a real harness bug the hand-rolled driver had missed

drain to quiet — after each random run, stop driving commands and let every job and conversion finish: every issued wait must be settled. The quiet-user liveness claim, replayed on the real code

exhaustive small-depth conformance — every sequence of drivable moves up to a depth (5-7; depth 7 is about 900K sequences, under a minute). Within the bound, agreement is proven, not sampled

runtime invariant hook — the code takes a check callback called with its state after every pump; the tests pass the model's 15 invariants into it, so the spec guards the code inside every test run. It caught a real stale-read bug on its first trace

coverage the model defines — 100% means all 31 model actions fired and every status was reached — the model, not the code, says what complete is. This run: 31 of 31

code-only tests — what the model abstracts away gets its own labeled tests, never a silent skip: env rejections mapping to the failed events, retries, many concurrent waiters served or released together

refinement — the field's word for a true proof that code implements a spec — heavy machinery almost nobody uses. Conformance testing with controlled scheduling is the working standard; exhaustive small-depth is the step up the most serious shops take

1.4 THE WIDER FIELD

formal methods - the umbrella name: writing a precise spec (the two pages) plus verifying claims about it. Verification comes in two strengths - model checking (enumerate every state within bounds, what this board does) and proof (covers all sizes, costs hand-guided work). This board is the first road end to end

temporal formula - how liveness is written in a real tool: always (true in every state of the movie), eventually (true at some point), and leads-to (whenever A holds, B holds some time later). "Every wait ends" is: requested leads-to served. Invariants only need always; liveness needs the rest

stuttering - a step where nothing changes. Every spec must stay true when the system pauses - the checker adds these steps for free. It is also why liveness needs fairness: without fairness, pausing forever is a legal movie and every "eventually" fails

primed variables - TLA+'s way of writing an effect: name is the value now, name' is the value after the step. "rename to n2" is written name' = n2. An effect function that returns a new state is the same idea in code

inductive invariant - an invariant strong enough that one step from any state satisfying it lands back inside it - not just true in the reachable states. Proofs need this form, and getting there usually means strengthening the claim with helper clauses. A checker that visits reachable states directly never needs one

symmetry reduction - when two values are interchangeable in every rule (nothing distinguishes f1 from f2), states that differ only by swapping them count as one. How industrial checkers keep state counts down; our 283,951 counts such twins separately

the three tool lanes - exhaustive on a model: TLA+, Quint, P - proves the design, says nothing about the code. Model-based sampling on real code: fast-check - runs in CI against what ships. No model at all: deterministic simulation - fake clock, seeded scheduler, replayable interleavings (FoundationDB, TigerBeetle). They answer different questions; 3-spec-driven used the first two

theorem proving - the proof road: Dafny, Coq, TLAPS. No bounds - the result holds for every size - but each step of the argument is guided by hand. Dafny's fit is one gnarly sequential algorithm with pre- and postconditions; it has no notion of three jobs in flight and answers a failed proof with a hint, not a trace

untimed vs timed model - the default is untimed: no clock, time is only the order of events. An event that "may fire at any moment" already covers every duration - a queue's visibility timeout is such an event, and a stored duration number is a variable no rule can read. A timed model (timed automata, UPPAAL; or a tick variable in TLA+) is earned only when a claim needs "how long", not "in what order". Durations themselves (the timer fires, once, after the configured time) are code-only tests with a fake clock

where timed is earned - the claim itself compares durations. A pacemaker: pace within 900ms of a beat but never within 200ms - the window is the correctness. Raft: heartbeat interval must be much shorter than election timeout, or followers start spurious elections - untimed cannot say "shorter than". Lock leases: the holder acts only while the lease lasts, so clock skew between machines must stay under a bound. Bus arbitration (CAN, CSMA backoff): who wins is defined by who waits how long. Absent from the list: business workflows, queues, jobs - there a timeout only means "may happen later", which is order, and untimed wins

the checker per edit - quint typecheck plus quint run take seconds, so they can run after every model edit - the checker becomes the reviewer while you think, not a later step. The errors it catches mechanically are the ones prose review catches slowly: a flipped comparison in an invariant, a field an effect forgot to clear, an effect reading a value the same step just changed (all three happened in the queue-core maxReceiveCount pass)

quint REPL - quint -r queue-core.qnt::queueCore opens the model as a prompt: fire one action at a time by hand and print the state after each - "what happens if the set command runs here" answered live instead of imagined

quint VSCode extension - syntax and type errors on save, inside the editor, before any terminal run - the shortest loop for model edits

Alloy - a modeling tool built around showing possibilities you did not imagine: write the rules and it draws diagrams of reachable states and counterexamples. The field's usual answer to "it is hard to think of all cases". A separate language to learn; Quint already covers the same need in our projects

no tool checks the prose page - a markdown spec page has no checker; its checkable form is the model next to it. Three ways to live with that: edit page and model together and run the checks on every edit; make the model file the spec and demote the page to explanation; or a named process step that mirrors page to model with one owner (the queue-core README does the third)

2. THE PROCESS (12 STEPS)

1 recognize the moment — you need a spec the first time you can name two events that could arrive in either order and cannot say what should happen. In code, the signal is the second hand-made flag around the same thing. Here: a third upload vs a running detection

2 variables and messages — bound everything; derive or store; record provenance; split values the rules treat differently. Modeling decisions, not product decisions yet

3 actions — refusal or wait; need vs permission; for every write, which running job reads what it changes (the cancel table); serve on the result; refuse up front. Every corner case is a product decision — decide it, record it in Considerations. This step surfaces most missed requirements before any tool runs

4 translate into the runnable model — mechanical, names identical to the spec. Fidelity is the only judgment

5 first enumeration, no invariants — read the endings and stuck states first — they surprise before any invariant would. Ask: is anything stuck, do the endings look like the stories you meant, is the count plausible

6 invariants as beliefs form — written during the loop, not before. A counterexample means the spec is wrong or the belief is wrong — the trace tells you which. A refuted belief that documents a real decision is kept and must keep failing

7 repeat until quiet — draft, enumerate, read the surprises, improve the rules, repeat. A full pass with no surprises is what "the spec is done" means

8 liveness in the standard tool — fairness grants are the honesty test: weak for "stays ready, eventually fires", strong for "keeps starting, eventually finishes", none for failures. A violation ends one of three ways: accept and record, change the spec, shrink the claim

9 cross-check the translations — random traces from one model replayed through the other; the BFS-measured diameter gives the exact verify bound

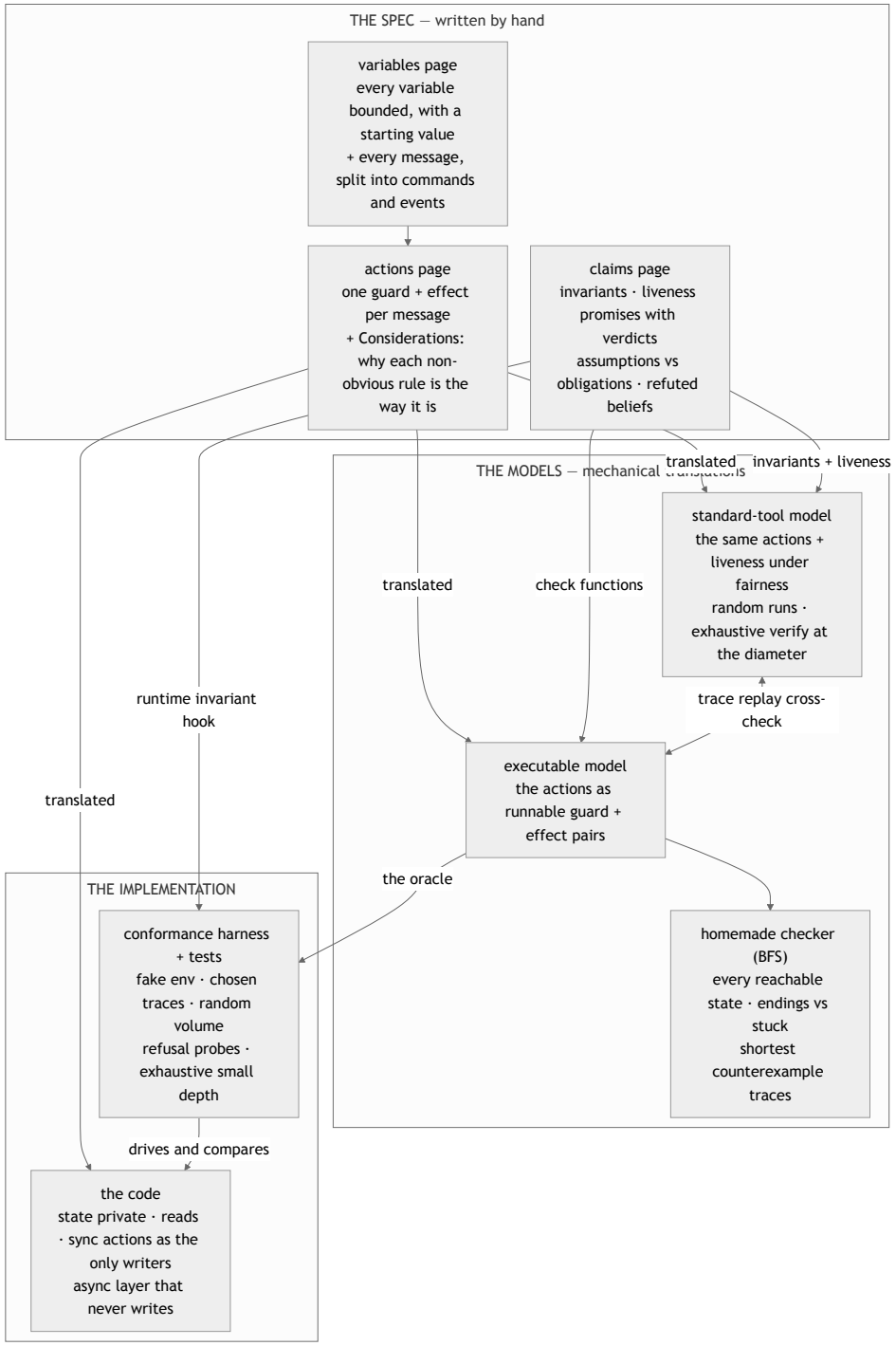
10 implement — the four-section rule. Everything the code adds beyond the model — generations, waiter queues, the pump's priority order — is added consciously and listed

11 conformance — chosen traces for every decided corner case; volume with refusal probes; the liveness drain on real code; exhaustive to a bounded depth; the model's invariants asserted at runtime; coverage the model defines

12 living with it — every rule change starts at the spec and flows down. Two alarms mean the model moved: a failed invariant, and a refuted belief that stops failing

not a waterfall — a finding at any level (a counterexample, a liveness loop, a conformance mismatch, even writing a test) pushes a change back up to the spec; everything below re-derives and re-runs in seconds, which is what makes the looping cheap

The artifacts — what exists when a spec-driven piece is finished



The process — 12 steps, a loop, not a waterfall