Machine-Checked Expressiveness and Failure Transparency for the Dex Durable Workflow Programming Model

Quanzheng Long
Superdurable
long@superdurable.io

Zheng Zheng
Superdurable
zane@superdurable.io

November 2026

Abstract

Dex is a durable workflow framework. An application is a *Flow* of *Steps*; each Step has an optional WaitFor method and an Execute method that returns a *StepDecision*, and the Steps of a Flow share *Attributes*, *Channels*, and *Timers*. This model has no threads, no locks held across waits, and no blocking calls. We ask two questions: can it express ordinary concurrent programs, and does a Flow behave the same when the system fails? We answer both with theorems instead of examples. We define a small concurrent imperative language L with threads, mutexes, condition variables, channels, timers, fork and join, cooperative cancellation, exceptions, and external effects. We give a formal operational semantics of the Dex core, written against the server implementation, and a translation from L into Dex Flows. We prove, in Lean 4, that every well-formed L program and its translation have the same observable finite traces (safety) and the same fair maximal traces with termination, deadlock, and divergence markers (liveness), in a failure-free environment. We then extend the semantics with server crashes, standby takeovers, Worker failures and retries, zombie Workers that keep running after the server gave them up, duplicate results, and client retries. Under explicit hypotheses and in the same fragment, we prove that failures are transparent: erasing the repeated external effects from a trace with failures gives a failure-free trace, and for fair runs the erasure keeps the marker. Composed, the two theorems relate a well-formed L program to its execution on a failing server, up to repeated effects. The two proofs are about 72,000 lines of Lean with no dependency beyond the Lean core and no `sorry` placeholder, and the main theorems depend only on the three standard axioms. We also report how the proofs were engineered: fixed statements proved in parallel work packages, gaps in the simulation invariants that a fuzz of perturbed reachable states finds, and two defects in the real server that writing the semantics exposed.

1 Introduction

A durable workflow engine runs long-lived application logic and keeps its state across process restarts. Engines differ in the programming model they give the application. Dex's model is deliberately small. An application defines a *Flow*. A Flow is a set of *Step* types. A Step has an optional WaitFor method, which returns what the Step waits for, and a required Execute method, which returns a *StepDecision*: go to other Steps, end this branch, or close the Flow. Steps read and write *Attributes*, publish to and consume from *Channels*, and wait on *Timers*. There is no thread that blocks in the middle of a function, no lock that is held across a wait, and no shared memory that is read and written in one atomic step.

A natural question follows: what can this model *not* express? The usual way to answer is by example. The Workflow Patterns [23, 25] are a catalog of 43 control-flow behaviors, and an engine is rated by how directly it supports each one [28]. Dex has such an evaluation: 19 of the 43 control-flow patterns have direct support, 24 have indirect support, and none is unsupported; each rating comes with a runnable example and an integration test against a real server.[1] A pattern catalog measures breadth against the running system.

[1]The evaluation is on the Dex documentation site, under *Correctness and expressiveness, Workflow Patterns evaluation.*

It does not say that some *class* of programs is expressible, and it does not say what "expressible" means when the engine interleaves the pieces of a program in ways the source program never wrote down.

This paper gives the other kind of evidence: two precise claims, each with a machine-checked proof.

> *Every well-formed program of a small concurrent imperative language* L *translates into a Dex Flow that is trace-equivalent to it, assuming no failures.*
>
> *In the same fragment and under explicit hypotheses, erasing the repeated external effects from a finite trace with crashes, takeovers, retries, zombie Workers, and duplicate deliveries gives a failure-free trace, and every failure-free trace remains possible. For fair runs the erasure keeps the termination, deadlock, or divergence marker.*

§3 to §7 develop the first claim, about expressiveness. §8 develops the second, about durability, the reason a workflow engine exists.

What "expressive" means here. L is a conventional shared-memory concurrent language: threads with local variables, shared variables, procedures with recursion, fork and join, mutexes, condition variables, sleep, a timed select on an external channel, cooperative cancellation, exceptions, external effects, and program exit. We define a translation $[\![\cdot]\!]$ from L programs to Dex Flow definitions, a set of observable events on each side, and a projection π from Dex events to L events. The claim is two equalities of trace sets, for every well-formed L program S:

$$\pi\big(\mathrm{FinTr}_{E_{adm}}([\![S]\!])\big) = \mathrm{FinTr}(S) \qquad \text{(safety)} \tag{1}$$

$$\forall w \in E_{adm}.\quad \pi\big(\mathrm{MaxTr}^{fair}_{w}([\![S]\!])\big) = \mathrm{MaxTr}^{fair}_{w}(S) \qquad \text{(liveness)} \tag{2}$$

FinTr is the set of finite observable traces. $\mathrm{MaxTr}^{fair}_{w}$ is the set of traces of fair infinite runs for the input word w, each with a marker: terminated ($\checkmark$), deadlocked (δ), or divergent (∞). E_{adm} is the admissible environment: it starts the program once and then sends plain data on the program's external channels. Equation (1) says that the Flow does nothing the program could not do, and the program does nothing the Flow could not do. Equation (2) adds that the two terminate, deadlock, and diverge under the same conditions, which finite traces alone cannot tell.

So "Dex is expressive" means: the Dex core can simulate a language that has the standard synchronization vocabulary of concurrent programming. The correspondence uses finite sequences of Dex transitions, some observable and some silent: an action of the program corresponds to a short sequence of Dex transitions, and some Dex transitions correspond to no action of the program. The observable traces are preserved. The translation is the constructive content. It shows *how* a mutex, a condition variable, or a cancellable timed receive is built from Steps, Channels, Attributes, and Timers.

Contributions.

1. **A formal semantics of the Dex core** (§2). It is a labeled transition system written against the implementation of the Dex server, with the rules citing the code that implements them. (One rule outside the theorem's fragment follows a fix that was not yet merged at the pinned commit, and says so.) Three adversarial reviews checked it against the code.
2. **The language L and its semantics** (§3), including the choices that make an exact equivalence possible: a timing model, a well-formedness scope, and unforgeable handles.
3. **The translation** from L to Dex (§4): one Step per program point, in continuation-passing style.
4. **The theorems** (§5) and **their proof in Lean 4** (§6): `safety_of_isWellFormed` and `liveness_of_isWellFormed`. No proof contains a `sorry` placeholder, the development depends on the Lean core library only, and `#print axioms` reports `propext`, `Classical.choice`, and `Quot.sound` for both theorems.
5. **Non-vacuity** of both theorems: a decidable static lock discipline that implies well-formedness, witness programs that together use every construct of L, and kernel-checked fair maximal traces for each of the three markers.

6. **An account of the proof engineering** (§7): parallel work packages with fixed statements, the gaps that the provers found in the fixed definitions, the fuzz that now tests those definitions for inductiveness, and two defects that the work exposed in the real server.
7. **Failure transparency** (§8): a failure model written against the server, with crashes, takeovers, Worker failures, zombie Workers, stale results, and StartFlow retries; an abstraction map with ghost history under which every failure step is a stutter; and the Lean theorems `safety_failure_forward`, `liveness_failure`, `endToEnd_safety`, and `endToEnd_liveness`, with the same three axioms. Kernel-checked witnesses apply the safety statement and the forward liveness inclusion with the markers ✓ and δ. Counterexamples show that H1 and H3 are needed and that a re-sent publication is a second request. The H4 counterexample also violates H1, because in the model H4 is a special case of H1.

What is not claimed. The scope is narrow on purpose, and §9 discusses each limit.

- *Failures are a separate theorem, under hypotheses.* By design, the first theorem covers failure-free runs: it fixes what a Flow does when nothing fails, and the second (§8) relates runs with failures to it. The failure theorem covers SYNC Step methods only. It assumes that no method exhausts its retries, that user code ignores the attempt number and similar context, and, for liveness, that failures are finitely many. It keeps the timer order of the first theorem as an idealization. Effects can repeat and overlap; the theorem does not make them exactly-once.
- *A model, not the code.* The Dex side of the theorem is a hand-written model of the server, validated by review against the source. It is not extracted from the implementation.
- *Well-formed programs only.* A program that can call unlock, wait, signal, or broadcast without holding the mutex is outside the domain, as a program with a data race is outside a theorem about data-race-free programs.
- *A restricted environment.* Inputs are one start and then publications to external channels. Remote procedure calls into a Flow, stopping a Flow, and the other client operations are specified in the semantics but are outside the theorem.
- *One timing model.* L counts a fixed number of clock-stamped actions per statement, aligned with Dex. The theorem does not say that a program is equivalent to itself under another cost model.
- *Trace equivalence.* The theorem is not bisimilarity and not full abstraction, and it says nothing about performance.
- *One direction of expressiveness.* We translate L into Dex. We do not claim that every Dex Flow is an L program.

Artifacts. The semantics is `formal/semantics/dex-core-semantics.md` ("the specification" below). The first theorem formalizes its revision 8; revision 9 adds the failure model of Section 11 and leaves every failure-free rule unchanged. The Lean development is `formal/lean/`. File paths in this paper are relative to the repository root, and Lean files are under `formal/lean/DexExpressiveness/`. Numbers for the first theorem are for commit `33e998003` of the development, and numbers for the failure theorem for commit `dd449bd57`, which changes no file of the first proof except for import lines.

2 The Dex programming model and its core semantics

2.1 The model

Table 1 lists the constructs of the Dex programming model. The specification gives rules for all of them. The theorem needs only those in the last column, because the translation of §4 uses no others.

Two properties of the model shape everything that follows. First, application code never blocks. A Step that must wait says so by returning a wait from WaitFor; the engine runs Execute when the wait is satisfied. Second, application code does not touch shared state directly. A method receives a *snapshot* of the Attributes when it starts, and it returns its writes, its publications, and its decision as one response. The

Construct	What it is	Used by the translation
Flow	One run of a Flow definition: the unit of execution and of observation. It starts with a start Step and an input, and it closes with a status and a result.	Yes
Step	A Step execution has an input, an optional WaitFor method, and a required Execute method. These two methods are the only place where application code runs, apart from RPCs.	Yes
StepDecision	What Execute returns: GoTo/GoToMany (create Step executions), DeadEnd (end this branch), ForceComplete and ForceFail (close the Flow), GracefulComplete, ForceCompleteIfChannelsEmpty, CancelSteps, CancelSiblingSteps.	GoTo, GoToMany, DeadEnd, ForceComplete, ForceFail
Wait	What WaitFor returns: Timer, Channel, and SubFlow conditions combined by AllOf, AnyOf, or AnyCombinationOf; or the empty wait.	AnyOf and AllOf over Timers and ForOne Channel conditions
Attribute	A key-value store of one run. An AttributeMap is a family of Attributes; a Step method sees an AttributeMap instance only if its options select it.	Yes
Channel	A FIFO queue of one run. A Channel condition has bounds; the default, ForOne, waits for one message and consumes exactly one.	ForOne only
Timer	A wait condition with a duration in whole seconds.	Yes
SubFlow	A child Flow started by a wait; its result is delivered to the parent.	No
RPC	A client call that runs a handler, reads and writes the run, and may start or cancel Steps.	No
Attribute lock	Mutual exclusion on Attribute keys for the duration of one Step method or RPC.	No

Table 1: The constructs of the Dex core. The specification covers all of them; the Lean model has the ones the translation and the admissible environment use.

engine commits that response later, in one atomic step, unless the Step was canceled in between; then the response is discarded.

2.2 Configurations

The specification defines the system state as partition clocks, a program, the runs of all Flows, engine state, and cross-Flow messages in transit. The admissible environment of the theorem starts one Flow, and the translation uses no SubFlows, so the Lean model (`Dex/Config.lean`) reduces the state to one partition:

$$\Sigma \;=\; \langle\, \mathit{clock},\ \mathit{lastCommitAt},\ \mathit{run} : \mathit{Run}^{?},\ \Theta : [\mathit{Task}],\ \mathit{nextTaskId} \rangle.$$

A run holds a status (*Running* or *Closed* with a close status), the collected outputs, the Step executions, a per-type Step counter, the Attributes, the pending messages of each Channel in commit order, the armed timers with their fire times, and a sequence number *seq* that every commit increments. A Step execution has an ID *T-n* (its type and its number from the per-type counter), an input, options, a phase $\varphi \in \{\mathit{WF}, \mathit{WT}, \mathit{EX}, \mathit{CO}, \mathit{CA}\}$ (WaitFor, Waiting, Execute, Completed, Canceled), an optional *invocation* (a method that is pending or running), an optional registered wait, and optional condition results. A registered wait records *sws*, the sequence number at which it was registered; it orders waiters.

Θ is engine state that the persistent state machine does not hold: the *tasks*, which are method attempts handed to a Worker. A task carries the request the engine built when the method started (context, input, snapshot, condition results) and a phase: *Started*, *Running* with the effects still to emit and the response, or *Returned* with the response.

Action (rule)	Kind	Effect	Events
`tick` [TICK]	environment	The clock advances by $\delta > 0$ microseconds.	none
`request` [FLOW-START]	commit	Enabled when no run exists. Creates the run and its start Step.	$start(T_0, v)$
`request` [PUBLISH]	commit	While the run is running: appends the message and wakes the Channel's waiters. Otherwise the request is rejected and nothing changes.	$pub(c, v)$, or $rejected(pub(c, v))$
`stepStart` [STEP-START]	commit	A pending invocation becomes running. The snapshot is taken, and a fresh task enters Θ.	none
`workerRun`, `workerEffect`, `workerReturn` [WORKER-EFFECT]	Worker	The handler runs; the task emits its external effects one at a time; the task becomes *Returned*.	$fx(e, m, a)$ per effect
`stepComplete` [STEP-COMPLETE-*]	commit	The response of a *Returned* task of a still-running invocation commits: its data effects, then the wait registration (WaitFor) or the decision (Execute).	none, or $closed(\ldots)$
`workerAbandon` [WORKER-ABANDON]	Worker	A task whose invocation was canceled is dropped, before or after any of its effects.	none
`stepCompleteStale` [STEP-COMPLETE-STALE]	Worker	A *Returned* task whose invocation is no longer running is dropped.	none
`timerFire` [TIMER-FIRE]	commit	Fires a non-empty prefix of the due timers, in index order, in one commit.	none, or $closed(\ldots)$

Table 2: The transitions of the restricted Dex system (`Dex.Action`, `Dex/Rules.lean`). The specification also has rules for the other client inputs, RPCs, SubFlows, and a deferred close; the admissible environment and the translation never enable them.

2.3 Rules

A transition is $\Sigma \xrightarrow{\vec{\ell}} \Sigma'$, where $\vec{\ell}$ is the ordered sequence of observable events it produces. A rule marked *commit* is one command of the server's deterministic state machine: it is atomic, it stamps the current clock value, and the clock must have strictly increased since the previous commit. In Lean a rule is a case of the partial function `Dex.step` (`Dex/Rules.lean`): the *action* names the rule and carries its nondeterministic choice, so determinism of each rule holds by construction (`step_deterministic`). Table 2 lists the ten actions.

A Step method is not atomic. This is the central fact of the semantics. A Step method begins with a Start commit ([STEP-START]). For an invocation that stays current and returns, Start is followed by a sequence of Worker transitions and then a Complete commit ([STEP-COMPLETE-WAITFOR] or [STEP-COMPLETE-EXECUTE]). The Worker transitions are the parts of [WORKER-EFFECT], which the Lean model has as separate actions: one `workerRun`, zero or more `workerEffect`, and one `workerReturn`. None of them is a commit. An invocation that is canceled after Start never completes: its task may instead be abandoned ([WORKER-ABANDON]), before or after any of its effects, or dropped as stale once it has returned ([STEP-COMPLETE-STALE]). Any other enabled transition can occur between any two of these. The method reads the snapshot taken at Start. Its writes are applied blindly at Complete; nothing checks that the values it read are still current. So a Step method is *not* a read-modify-write transaction: two Steps that both read an Attribute and both write it can lose an update. The translation is designed around this: it puts at most one shared access in each method, so that a read linearizes at Start and a write at Complete.

Inside one commit. The effect of a commit is defined by deterministic meta-functions (`Dex/MetaFunctions.lean`), run in a fixed order. The ones the proof relies on are these.

- *Data effects*: Attribute writes in order, then Channel publications in order, then a *wake* of the Channels published to.
- *Wake*: every Step that waits on a changed Channel is re-evaluated, earliest registration first (by *sws*). With ForOne waiters only, a Channel is therefore FIFO by registration. Lean: `Dex.wake_servesEarliestWaiter`.
- *Evaluation*: a wait is satisfied when its plan is feasible. AllOf needs every condition. AnyOf takes the first feasible candidate in the order timers, then Channel conditions in declaration order, and consumes only from the winner (`Dex.registerWait_any_firstChannelWins`). A satisfied wait moves the Step to phase *EX* and makes its Execute pending.
- *Registration*: a non-empty wait arms its timers at the commit's timestamp plus whole seconds, records *sws*, and is evaluated once. A timer is never done when its wait registers, even a zero-second one (`Dex.registerWait_timer_blocks`).
- *Decision*: the deciding Step completes; then a close decision closes the run, canceling every active Step, or the moves create new Step executions.

Timers fire late and in order. [TIMER-FIRE] is enabled when some armed timer's fire time is at or before the clock. It fires a non-empty *prefix* of the due timers, in the order (fire time, encoded timer ID), in one commit. Firing can be arbitrarily late: any commit may come between a timer becoming due and its firing, so a due timer can lose an AnyOf race to a message that arrives after its fire time. The specification argues from the timer service's code that the server fires a prefix, and not an arbitrary subset. A later review found one failure-free window in which that is not exact: the server can fire a timer while an earlier-due timer, armed by a command applied just before, is still armed. Both theorems keep the prefix rule as an explicit idealization (§9). An over-approximation would be unsafe here: for an equivalence theorem, extra Dex runs are extra obligations for L.

2.4 Observable events

The Dex event alphabet *Obs* of the restricted system is $start(T, v)$, $pub(c, v)$, $rejected(pub(c, v))$, $fx(e, m, a)$, and $closed(status, outs)$ (`Dex.Obs`). An *fx* event is one external effect of a handler, such as a call to another system. It is emitted by the Worker, outside any commit: an effect can happen with no commit after it, and after the Flow closed, when its Step was canceled in between.

Reads are not observable. Including them would make every intermediate Attribute write visible and would turn the claim into a much stronger one that no compilation of threads into Steps could satisfy.

3 The language L

L is the source side of the theorem. It is meant to be recognizable: a reader who knows POSIX-style threads should find no surprise in its constructs. Its semantics, however, is fixed in detail, because an exact equivalence leaves no room for "implementation-defined".

3.1 Syntax

Figure 1 gives the syntax (`L.Stmt`, `L.Expr`, `L/Syntax.lean`). A program has a list of procedures, a main body with one parameter that receives the program input, and a number of *statically known* mutexes, which exist and are unlocked when the program begins. Expressions are pure and read thread-local variables only. Shared variables are read by load and written by store, one access per statement.

Values are the literals plus three kinds of handles: thread handles, mutex handles, and condition variable handles. A condition variable is named by its mutex and an index, so it belongs to exactly one mutex.

L's observable alphabet Obs_{L} has six events (`L.Event`): $begin(v)$, the program starts with input v; $in(c, v)$, a message arrives on external channel c; $refused(c, v)$, a message arrives after the program ended; $out(a)$, an emitted effect is delivered; $exit(v)$; and $fail(s)$.

$$
\begin{aligned}
\mathit{lit} &::= \mathsf{null} \mid \mathit{bool} \mid \mathit{int} \mid \mathit{string} \\
e &::= \mathit{lit} \mid x \mid \mathsf{staticMutex}(i) \mid \mathsf{cvOf}(e, i) \mid e + e \mid e = e \mid e < e \mid \neg e \\
s &::= \mathsf{skip} \mid s; s \mid x := e \mid x := \mathsf{load}\ g \mid \mathsf{store}\ g := e \mid \mathsf{if}\ e\ \mathsf{then}\ s\ \mathsf{else}\ s \mid \mathsf{while}\ e\ \mathsf{do}\ s \\
&\mid\ x := p(\vec{e}) \mid \mathsf{return}\ e \mid x := \mathsf{fork}\ p(\vec{e}) \mid \mathsf{join}(e) \mid \mathsf{cancel}(e) \\
&\mid\ x := \mathsf{newMutex} \mid \mathsf{lock}(e) \mid \mathsf{unlock}(e) \mid \mathsf{wait}(e_{cv}, e_m) \mid \mathsf{signal}(e) \mid \mathsf{broadcast}(e) \\
&\mid\ \mathsf{sleep}(e) \mid x := \mathsf{recv}(c) \mid \mathsf{select}\ \{\, x := \mathsf{recv}(c) \to s \ \mid\ \mathsf{after}(e) \to s \,\} \\
&\mid\ \mathsf{emit}(e) \mid \mathsf{throw}\ e \mid \mathsf{try}\ s\ \mathsf{catch}\ (x)\ s \mid \mathsf{exit}(e) \mid \mathsf{fail}(e)
\end{aligned}
$$

Figure 1: The syntax of L. x is a thread-local variable, g a shared variable, p a procedure, c an external channel, and i an index.

3.2 Semantics

The semantics is a small-step transition system (`L.step`, `L/Semantics.lean`). A configuration holds a status (not started, running, closed), a clock, the live threads, the shared variables, the owner of each mutex, the FIFO queue of each condition variable, the queued messages of each external channel, the cancel flags, the threads that ended and were not joined, and, after the end, the emits that were issued and not yet delivered. A label is an idle clock tick, an environment request, the next action of one thread, the internal action fire, or the delivery or drop of a late emit.

The definition has two layers. `L/ThreadLocal.lean` says what a statement does to its own thread and which atomic action it asks for, as a value of type `L.Intent`. `L/Semantics.lean` performs that action on the shared state. The translation reuses the first layer verbatim (§4), which is what keeps the two sides aligned.

The semantic choices below are the ones that matter for the equivalence. Each matches what the Dex core does.

Atomic actions. Each single read or write of a shared variable, each issue of an emit, and each synchronization operation is one atomic action. Local computation is invisible. There is no atomic read-modify-write: $g := g + 1$ is a load and a store, and two threads can interleave them.

Mutexes. Threads blocked in lock(m) acquire m in FIFO order of blocking. A mutex that does not exist is never free.

Condition variables. wait(cv, m) atomically releases m and appends the thread to cv's FIFO queue. signal wakes the head of the queue, broadcast wakes all. A woken thread re-locks m, in FIFO order with other lockers. There are no spurious wakeups.

Cancellation is cooperative and deferred. cancel(t) sets t's flag. Thread t observes the flag at a cancellation point (sleep, recv, select, join), and only if that point would block; it then ends without releasing anything it holds, as in POSIX without cleanup handlers. lock and wait are not cancellation points. A canceled thread is joinable like any other ended thread.

Emit is asynchronous. emit(a) is *issued* as one atomic action. The event *out*(a) happens later, after any interleaving of other threads, and the thread continues only after it. If the program ends between the issue and *out*(a), the effect may still be delivered after *exit*, or never.

Exit. exit(v), fail(s), or the return of main ends the program at once: every other thread stops. A thread that ends otherwise just ends, and a program whose threads all end without exit is stuck forever. Input that arrives after the end is *refused*.

Join. A thread that ends leaves one join token. join(t) takes the token if it is there; otherwise it observes its own cancel flag; and otherwise it blocks. A second join of the same thread blocks forever.

Exceptions are data. throw transfers control to the innermost handler on the thread's stack. An uncaught exception whose value is a string s fails the program with *fail*(s).

Halting. L has no static types. A statement whose operand has the wrong type, and an uncaught exception that is not a string, halt the thread: it takes no further action, never ends, and releases nothing.

L step	Stamped actions
begin(v)	1, then 2 more before main's first statement
$s; s$ and skip	0
A statement that does not block (assignment, load, store, if, each test of while, call, return, fork, newMutex, unlock, signal, broadcast, emit, throw, entering try, cancel, exit, fail); the end of a try body or a procedure body	2: begin, complete
lock, recv, join	4: begin the wait, register it, begin the continuation, complete it
wait(cv, m)	10: 2 to release m and enqueue, 4 to wait for the signal, 4 to re-lock m
sleep, select with after	6: 2 for Enter, then 4, of which the register is Arm
in(c, v) while the program runs	1
A batch of $k \geq 1$ consecutive fire actions	1
out(a); delivery or drop of a late emit; *refused*(c, v)	0

Table 3: The timing model of L. A shared read happens at begin, an emit is issued at begin, and every other atomic action happens at complete or at register.

3.3 The timing model

L has an internal clock in microseconds, advanced by idle ticks. A timed operation's deadline is the clock value when it is armed plus a whole number of seconds. Dex computes fire times the same way, from commit timestamps, and those strictly increase from commit to commit. So two runs, one on each side, can carry the same deadlines only if the same number of clock-stamped steps lies between any two arming points. If the counts differ, the order of two deadlines can differ, and a select makes that order observable.

L therefore has a *fixed timing model*: each kind of statement takes a fixed number of *stamped actions*, and the clock must strictly increase between any two stamped actions. The counts (Table 3) are the numbers of Dex commits of the translation.

This was an explicit decision of the model's owner (row L1 of the specification's review log), with this rationale: real programs cannot rely on timing races. A program whose outcomes do not depend on the order of two deadlines means the same under this timing model as under any other. The theorem adopts the one model that is aligned with Dex, which makes the equivalence exact without narrowing what it covers in practice. It is like counting one time unit per instruction in algorithm analysis. The theorem does *not* say that a program is equivalent to itself under another cost model.

In Lean the model is the phase of a thread (`L.Phase`): a statement that does not block goes `ready` → `running` → `ready`; an emit goes `ready` → `emitting` → `emitted` → `ready`; a blocking statement goes `ready` → `waitStarted` → `blocked` or `woken` → `resuming` → `ready`.

3.4 Timers

A timed operation takes two actions. *Enter* gives it the next timer number n from one program-wide counter. *Arm*, later, sets its deadline to the current clock plus the duration and blocks the thread. The timer completes only by the internal action fire, which is enabled at any time no earlier than the deadline and possibly much later. fire fires only the *least* due timer in the order (deadline, n). A batch of consecutive fire actions shares one clock stamp, as one [TIMER-FIRE] commit fires a prefix of the due timers. A blocked timed operation is decided by whichever comes first: its fire, the arrival of its message, or a cancel. A timer is never complete when it is armed, even with duration zero.

3.5 Scope decisions

Three more decisions bound the theorem. Each is recorded in the review log of the specification with its date and its reason.

Well-formed programs (L2). wait, signal, and broadcast require the condition variable's mutex to be held, and unlock requires the mutex to be held by the caller. L gives no behavior to a thread that is about to break a requirement: the thread has no step. The theorem covers the programs in which this never happens.

```
def Thread.BreaksMutexRequirement (config : Config) (thread : Thread) : Prop :=
  thread.phase = .ready ∧ thread.timer = none ∧
  ∃ mutex, requiredMutex thread.core thread.code = some mutex ∧
    config.mutexOwners.get? mutex ≠ some (some thread.core.tid)

def BreaksMutexRequirement (config : Config) : Prop :=
  ∃ thread, thread ∈ config.threads ∧ thread.BreaksMutexRequirement config

def Program.IsWellFormed (program : Program) : Prop :=
  ∀ config, (system program).Reachable config → ¬ BreaksMutexRequirement config
```

`Program.IsWellFormed` is a property of the source program under L's semantics alone, quantified over every input and every schedule, like data-race freedom. The translation plays no part in it.

Unbounded counters (L3). Step numbers and timer numbers are natural numbers in the model. The real server stores a Step number in 32 bits. The order of timers with equal fire times is derived from the byte order of encoded timer IDs, and that derivation is exact for fewer than 2^{32} Step executions of one type per run. The theorem says nothing about a run that exceeds the limit.

Unforgeable handles and literal-only inputs (L5, L6). The proof itself found these two points. In an earlier version a literal could be any value, including the handle of a mutex that newMutex had not created yet. Consider

locker(): lock(h); emit(1) main: fork *locker*(); sleep(0); newMutex; recv(9)

where h is the forged handle of the mutex that main creates. In L the locker blocks forever, because creating a mutex does not serve blocked lockers. In Dex the first token that the translated newMutex publishes wakes the waiting Step, and every fair run emits *out*(1). The liveness statement was false for this well-formed program. The fix is at the source: literals are only null, Booleans, integers, and strings (`L.Literal`); a mutex handle comes only from newMutex or staticMutex; a thread handle comes only from fork. The lemmas `evalExpr_mutex_origin` and `evalExpr_thread_origin` state this.

The counterexample then came back through the environment: the program input was an arbitrary value, so it could be the forged handle. The second decision closes that door: the environment sends plain data only. A request carries a `L.Literal`, and `literal_isNotHandle` proves that no literal is a handle. After both decisions the counterexample cannot be written down, which is a stronger guarantee than a side condition that excludes it.

4 The translation

The translation $[\![\cdot]\!]$ (`Translation.translate`, `Translation/Flow.lean`) compiles each L thread into continuation-passing Steps [3]. It maps a program to one Flow definition with three kinds of Step types (`StepType`):

- `point code`: a *program point*. The Step type is the code left in the current procedure; the Step runs the head statement of that code and moves to the program point of what follows.
- `start`: the entry Step. It publishes one token for each statically known mutex and moves to the first program point of main.
- `timed`: one generic Step type for every sleep and every timed select.

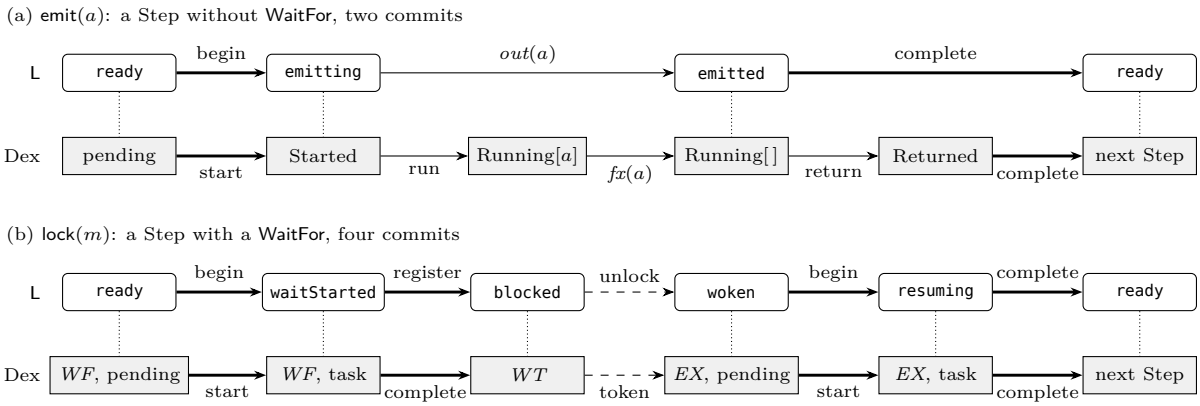


Figure 2: One L statement and its Step. Upper lanes: the phases of the thread (`L.Phase`). Lower lanes: the Step's invocation and task. Thick arrows are clock-stamped: a stamped L action above, a Dex commit ([STEP-START] or a Complete rule) below. (a) An emit is issued at [STEP-START]; $out(a)$ is the Worker's effect; the thread continues at [STEP-COMPLETE-EXECUTE]. The Worker's run and return steps have no L counterpart. (b) A blocked lock is woken by another thread's unlock (dashed): the commit that publishes the token moves the waiting Step from *WT* to *EX*.

The Step *input* is the thread's state: its thread ID, a counter for fresh names, its local variables, and its stack of call frames and exception handlers. A GoTo passes that state on. So each live thread that has not halted is carried, at any moment, by exactly one active Step execution, and the thread's "program counter and stack" are that Step's type and input. A halted thread has no Step: its last Step decided DeadEnd.

One Step per program point. A program point normally uses one Step execution. A statement that does not block has no WaitFor; its Execute takes two commits, [STEP-START] and [STEP-COMPLETE-EXECUTE]. A blocking statement's Step has a WaitFor that returns the wait, which adds two commits. Two kinds of statement use more than one Step execution. wait(cv, m) expands into three: the release, the wait for the signal, and the re-lock. A timed operation, sleep or a timed select, uses an Enter Step at its program point, whose Execute only moves on, followed by a Step of the shared `timed` type. Sequencing and skip are not Steps. These commit counts are the stamped actions of Table 3, and Figure 2 shows the correspondence for the two shapes.

The Execute of a program point. `pointStepDef` runs the thread-local semantics of the head statement (`L.completeStatement`, the same function L's own semantics uses) and encodes the atomic action it asks for as Dex effects and a decision (`encodeIntent`). `encodeIntent` is the mapping table of the specification, one case per row; Table 4 reproduces it.

The translation uses GoTo, GoToMany, DeadEnd, ForceComplete, and ForceFail, and nothing else: no CancelSteps, no GracefulComplete, no RPCs, no SubFlows, no Attribute locks. Every decision it returns is well formed for every input (`translate_execute_isWellFormed`).

Why the encodings are faithful. Each encoding rests on a property of the Dex core from §2.3.

Mutex. A mutex is a Channel that holds one token exactly while the mutex is unlocked. The Channel has only ForOne waiters, so a wake serves them in registration order, and a token that is present when a waiter registers means that no earlier waiter was blocked. This gives L's FIFO acquisition. An Attribute lock would not do: it is held for one Step method, not across a wait.

Condition variable. The waiter list is an Attribute. Reading and writing it in one Execute would normally be a lost-update hazard (§2.3); here it is safe because the Execute runs while its thread holds m, and no other thread touches the list. This is where the mutex requirement enters: L does not let a thread begin such a statement without m, the translated Step has no such check, and well-formedness says that the difference never shows. The wake goes to a per-waiter Channel, so a signal reaches the dequeued waiter even if that waiter has not registered its wait yet: the message stays queued. No spurious wakeup is needed. The waiter

L construct	Dex mapping
if, while, local assignment	A Step whose Execute GoTos the next program point (the taken branch, the loop body, or the loop exit).
Shared read, shared write	One Execute per access: read from the snapshot, or write one Attribute.
Call, return, recursion	Push or pop a frame on the explicit stack in the Step input. No SubFlow.
throw, try	The handler stack is in the Step input; throw is a GoTo to the innermost handler's program point. No Worker error.
fork	GoToMany: the parent's continuation and the child's first program point. The child's thread ID is the parent's ID extended by the parent's counter.
Thread end	Publish one message to the Channel *join*/t, then DeadEnd.
join(t)	WaitFor returns AnyOf(*join*/t, *cancel*/*self*).
exit(v), return of main	ForceComplete with the tagged value {`exit` : v}, which is never null.
fail(s), uncaught string exception	ForceFail(s).
Halted thread	DeadEnd, publishing nothing.
newMutex, unlock(m)	Execute publishes one token to m's token Channel.
lock(m)	WaitFor returns Until(m's token), that is, AllOf with one condition: a ForOne waiter, FIFO by registration.
wait(cv, m)	Three Steps. (1) One Execute, under m: append the thread ID w to the Attribute *waiters*/cv and publish m's token. (2) Wait on the per-waiter Channel cv/w. (3) lock(m) again.
signal(cv)	One Execute, under m: if *waiters*/cv is not empty, remove its head h and publish to cv/h.
broadcast(cv)	As signal, for every waiter in order.
cancel(t)	Execute publishes to *cancel*/t.
recv(c)	WaitFor returns AnyOf(c, *cancel*/*self*). The environment's publications fill c.
sleep(d)	GoTo the `timed` Step (Enter). Its WaitFor returns AnyOf(*Timer*(d), *cancel*/*self*) (Arm).
select with after(d)	As sleep, with the wait AnyOf(*Timer*(d), c, *cancel*/*self*). Only the winner consumes.
emit(a)	An Execute whose only external effect is a.

Table 4: The translation, by construct (`encodeIntent`, `encodeWaitRequest`, `timedStepDef` in `Translation/Flow.lean`). Every Channel condition is ForOne.

list is an AttributeMap instance, so the Step that reads it must select it in its load options; the model applies the selection as the server does, and `readWaiters_snapshot_of_cvOperand` proves that the declared selection is enough.

Cancellation. A cancel is a message on the per-thread Channel *cancel*/t. AnyOf consumes only from the winning condition, so a cancel never eats an input message or a join token. At registration no timer is done, and the Channel conditions decide in declaration order, the operation's own Channel before *cancel*/*self*. That is L's rule that a cancel is observed only if the point would block. The translation never cancels a Step. The only Step cancellation in a translated Flow is the Flow close, after which nothing is observable except effects already in flight, which L's asynchronous emit allows.

Timers. Every timed operation runs in the one Step type `timed`, with one timer at index 0. L's Enter is the commit that creates the Step, so the timer number is the Step number. L's Arm is the WaitFor completion, so the deadline is the fire time. For equal fire times Dex orders timers by encoded timer ID, which for these IDs is numeric order on the Step number. A [TIMER-FIRE] prefix of size k is then k consecutive fire actions of L.

Exit. ForceComplete drops a null output, so the translation passes the tagged value {`exit` : v}, which is never null even when v is L's null. The projection reads v back from the one output of the closed Flow.

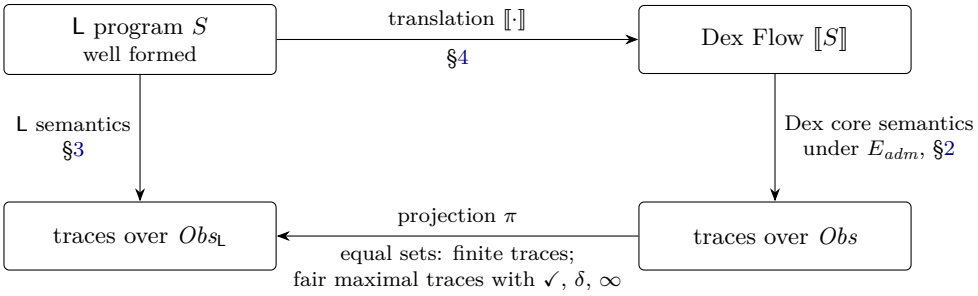


Figure 3: The claim. The traces of the translated Flow, projected to L's alphabet, are the traces of the program.

Dex event	L event
$start(T_0, enc(v))$	$begin(v)$
$pub(c, enc(v))$, c external	$in(c, v)$
$rejected(pub(c, enc(v)))$, c external	$refused(c, v)$
$fx(e, m, a)$	$out(a)$ (the source is dropped)
$closed(Completed, [(e, \{\texttt{exit} : v\})])$	$exit(v)$
$closed(Failed(s), _)$	$fail(s)$
every other event	ε

Table 5: The projection π from Dex events to L events.

5 The theorems

Both sides are instances of one structure, `TransitionSystem` (`Core/TransitionSystem.lean`): an initial state, a partial step function that returns the events of a transition, the request a label attempts (if any), whether a label is a *progress step*, and whether a state is *closed*. Finite traces, runs for an input word, markers, and fair maximal traces are defined once, for any transition system, so the two trace sets of each statement are built the same way. Figure 3 shows the shape of the claim.

5.1 The admissible environment and the projection

The Dex alphabet has raw identifiers and the theorem's environment is restricted, so the statement is over a projection, under an admissible environment.

E_{adm}. An admissible input word is a finite or infinite sequence of *attempted* client requests, accepted and rejected alike. It begins with exactly one request that starts the Flow at the entry Step with the program input. Every later request publishes one message to one of the program's external Channels. The start input and every message are encoded L literals (§3.5). There is no other client operation. In Lean, `Translation.environment` fixes these data and `Dex.Environment.IsAdmissibleWord` states the shape. The L form is `L.IsAdmissibleWord`: one `start`, then `input` requests. The lemmas `isAdmissibleWord_encode` and `isAdmissibleWord_decode` show that the encoded admissible L words are exactly the words of E_{adm}.

Counting attempted requests, and not only accepted ones, matters. A publication after the Flow closed is rejected and produces only a *rejected* event; L produces *refused* for the same input. A word of accepted events only would give the L run that exits and then refuses an input no matching Dex run.

π. The projection $\pi : Obs \to Obs_{\mathsf{L}} \cup \{\varepsilon\}$ (`Translation.project`) is given by Table 5.

5.2 Safety

A finite trace of a system is the event sequence of a finite run from the initial state, viewed through a function that may erase events (`TransitionSystem.IsFiniteTrace`). The two sets of Equation (1) are (`TraceSets.lean`):

```
def IsProjectedDexFiniteTrace (program : Program) (trace : List Event) : Prop :=
  (Dex.system ⟦program⟧).IsFiniteTrace environment.IsAdmissiblePrefix project trace

def IsLFiniteTrace (program : Program) (trace : List Event) : Prop :=
  (L.system program).IsFiniteTrace (fun _ => True) some trace
```

Lean's core library has no set type, so an equality of sets is written as a bi-implication for every element. The safety theorem (`Theorems.lean`) is:

```
theorem safety_of_isWellFormed (program : Program) (isWellFormed : program.IsWellFormed)
    (trace : List Event) :
    IsProjectedDexFiniteTrace program trace ↔ IsLFiniteTrace program trace
```

This is weak trace equality: Dex transitions without a projected event are invisible. Finite traces alone cannot tell a Flow whose last thread ended with DeadEnd, and which stays running forever, from a Flow whose close an unfair scheduler never runs. Liveness closes that gap.

5.3 Liveness

Runs for a word. Clock ticks are always enabled, so no maximal run is finite. A run *for* an input word w is an infinite run whose request transitions are exactly the requests of w, each once and in order (`IsRunFor`). A tick is time passing, not progress; every other transition is a progress step.

Fairness. The rules themselves are unfair: the dispatcher may delay a startable method, the timer service a due timer, and a Worker its response, each without bound. Liveness is therefore stated for fair runs. The conditions are assumptions about the scheduler, the timer service, and the Workers. They are not guarantees that the rules provide. Table 6 states them on both sides. Each of F2 to F5 is a weak-fairness condition of one shape (`EventuallyFails`): a given predicate on the state does not hold forever from any point on.

F4 deserves emphasis. The server does not make a Worker finish. It hands a method to a Worker, and for a canceled method it sends a cancel request; cancellation is cooperative. That every task eventually leaves Θ is an assumption about Workers that run terminating handlers and honor cancellation or run to completion. A Worker that hangs forever on a task gives an unfair run, about which the theorem says nothing.

Markers. A fair run is infinite, because time keeps passing. It is classified by its progress steps (`HasMarker`):

- ✓ (`terminated`): finitely many progress steps, and the program is closed at some point.
- δ (`deadlocked`): finitely many progress steps, and the program is never closed. From some point on only ticks happen: nothing can start, no task is in flight, no timer is armed, and the word is used up. A program whose threads all ended without exit lands here.
- ∞ (`divergent`): infinitely many progress steps.

A ✓ run has finitely many progress steps, so by F4 and F5 no task is left in Θ after the last one. The trace therefore already contains every effect that a canceled Worker emits after the close, and a ✓ trace is never a proper prefix of another fair trace for the same word.

The statement. A fair maximal trace for w is the whole observable trace of a fair run for w, with the run's marker (`IsFairMaximalTrace`). The two sets of Equation (2) are:

```
def IsProjectedDexFairMaximalTrace (program : Program) (word : Sequence L.Request)
    (trace : Sequence Event) (marker : Marker) : Prop :=
  (Dex.system ⟦program⟧).IsFairMaximalTrace Dex.IsFair project
    (word.map encodeRequest) trace marker
```

	Dex (`Dex.IsFair`, `Dex/Observation.lean`)	`L` (`L.IsFair`, `L/Semantics.lean`)
F0	Every request of w is eventually attempted, in order. It follows from "run for w".	The same.
F1	The clock exceeds every bound.	The same.
F2	No timer stays armed and due forever. A timer stops being armed when [TIMER-FIRE] fires it, or when its wait is decided another way or its Step is canceled. For a timer that nothing else disarms, this assumes that the timer service eventually fires it.	No thread stays blocked forever on an armed timer that is due: fire happens, or a message or a cancel decides the wait, or the program ends.
F3	No invocation stays pending forever. A pending invocation ends by starting, or when the Flow closes and cancels its Step. Otherwise this assumes that dispatch eventually hands it to a Worker. In this fragment every pending invocation is startable, because there are no locks.	No thread stays forever in a phase in which its next action has not begun (`bootPending`, `ready`, `woken`).
F4	No task stays in Θ forever without having returned: it emits all its effects and returns, or, if its invocation was canceled, it is dropped after some prefix of its effects. An assumption on the Workers: handlers terminate, and a Worker whose task was canceled eventually stops or finishes.	No thread stays `emitting` forever while the program runs; after the end, a non-empty list of late emits eventually shrinks.
F5	No task stays *Returned* forever: its result commits, or it is dropped as stale. An assumption on the engine: a returned response is eventually proposed as a commit.	No thread stays forever in a phase in which it began an action and has not completed it.
F6	Cross-Flow messages are delivered. Vacuous here: one Flow.	Vacuous.

Table 6: The fairness assumptions. They constrain runs; no rule enforces them.

```
def IsLFairMaximalTrace (program : Program) (word : Sequence L.Request)
    (trace : Sequence Event) (marker : Marker) : Prop :=
  (L.system program).IsFairMaximalTrace L.IsFair some word trace marker
```

A `Sequence` is a finite or infinite sequence. The word is quantified in `L` form and encoded for the Dex side. The liveness theorem is:

```
theorem liveness_of_isWellFormed (program : Program) (isWellFormed : program.IsWellFormed)
    (word : Sequence L.Request) (isAdmissible : L.IsAdmissibleWord word)
    (trace : Sequence Event) (marker : Marker) :
    IsProjectedDexFairMaximalTrace program word trace marker ↔
      IsLFairMaximalTrace program word trace marker
```

`expressiveness_of_isWellFormed` is the conjunction of the two theorems. The theorem names carry the scope, so that it is visible wherever a theorem is used.

Why this equivalence. The specification chose trace equality over weak bisimilarity, which is stronger than the claim needs and distinguishes *where* a nondeterministic choice is made. Trace equality with markers, per input word, still separates normal exit, failure, deadlock, and divergence, and it accounts for effects after the close. The proof does establish step simulations in both directions over one relation (§6); the theorems state only what follows for traces. The theorem is untimed: clock values are not observable. Time still matters, through the order in which timers fire and through F1 and F2.

What the kernel checked. For both theorems, `#print axioms` reports `propext`, `Classical.choice`, and `Quot.sound`: the three standard axioms of Lean, and nothing else. No proof contains a `sorry` placeholder

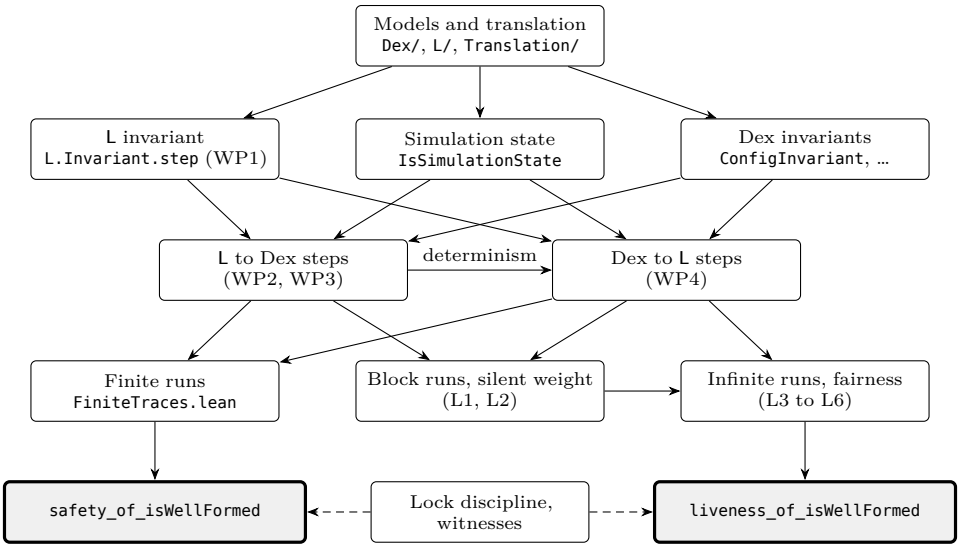


Figure 4: The structure of the proof. WP1 to WP4 are the work packages of safety, L1 to L6 those of liveness (§7). The dashed arrows are non-vacuity (§6.6): the hypothesis holds for concrete programs, and the trace sets are inhabited.

(the word occurs only in comments), and the proofs do not use `native_decide`. The trusted base is the Lean 4 kernel [6] and the definitions a reader must check against the specification: the transition-system core, the Dex model, the L model, the translation, the projection, and the two statements, about 3,450 lines in 14 files.

6 The proof

The proof has the classical shape of a simulation argument [1, 18]: a relation between the states of the two systems, invariants of each side, a step simulation in each direction, and then the lifting of those steps to finite and infinite runs. Figure 4 shows the parts. This section describes them in order and says where each one is in the Lean development.

6.1 The simulation state

Every step lemma takes one hypothesis, `IsSimulationState` (`Simulation/Relation.lean`), with five parts:

```
structure IsSimulationState (program : Program) (lConfig : L.Config)
    (dexConfig : Dex.Config signature) : Prop where
  isRelated : Related program lConfig dexConfig
  lInvariant : L.Invariant program lConfig
  dexInvariant : ConfigInvariant program dexConfig
  tasksDistinct : TasksAreDistinct dexConfig
  timersDistinct : TimersAreDistinct dexConfig
```

`Related` holds only the clauses that mention both sides. The clocks agree, and the last stamped L action carries the timestamp of the last Dex commit: this is the timing model at work. The statuses correspond (not started, running, closed). While the program runs (`RunningRelated`):

- *Data.* Shared variables are Attributes; each condition variable's queue is its waiter-list Attribute; each external channel's queue is its Channel's pending messages.
- *Tokens.* A mutex's token Channel holds one token exactly while the mutex is unlocked; *join*/t holds one message exactly while t has ended and was not joined; *cv*/w holds the wakes of a waiter that has not parked yet. A pending cancel request means the flag is set; a set flag means a pending request, or a thread that has observed it, or an ended thread that no live thread can still fork.

- *Timers.* The armed Dex timers are the armed L timers, with the same number and deadline, and the timer counter is the counter of the `timed` Step type.
- *Threads and Steps.* Every thread that has not halted is carried by exactly one active Step, and every active Step carries one thread (`StepCarriesThread`): the Step has the type, input, and options the translation gives that thread. The thread's phase fixes the Step's phase, invocation, and task (`PhaseMatchesStep`), as in Figure 2. In particular a Step is in phase WT exactly while its thread is blocked, so a publication wakes a Step exactly when it wakes the thread.
- *Order.* Two blocked threads are ordered by their blocking stamps as their Steps' waits are ordered by *sws*. This carries FIFO from one side to the other.

After the close (`ClosedRelated`), the late emits of L are, up to permutation, the effects that the remaining stale tasks have not emitted yet; no Step has an invocation, and no timer is armed.

The invariants. `L.Invariant` (`L/Invariant.lean`) holds the facts about L alone: thread IDs are unique and the next children of a thread are fresh; the next mutex a thread creates does not exist yet; every handle to a *created* mutex, or to a condition variable of one, that a thread can reach names a mutex that newMutex has created (this is where unforgeable handles are used); blocking stamps and timer numbers are distinct; each thread's phase fits its code; a thread that began a statement that requires a mutex holds it until the statement completes. The invariant does not check a static index against the program's number of static mutexes: staticMutex(i) with i out of range names a mutex that never exists, and a lock of it blocks forever on both sides. On the Dex side, `ConfigInvariant` wraps a run invariant (`Dex.Run.Invariant`: unique Step IDs, counted Step numbers, injective *sws*, only the wait shapes the translation uses, and no pending message on a Channel that a waiting Step names) and says that a task holds the response its handler computes; `TasksAreDistinct` and `TimersAreDistinct` say that task IDs and armed timer IDs are unique. Each invariant is proved to hold initially and to be kept by every transition (`L.Invariant.step`, `ConfigInvariant.step`, `TasksAreDistinct.step`, `TimersAreDistinct.step`).

6.2 From L to Dex: named action sequences

The L-to-Dex direction does not say "there exist Dex actions". It names them. `simulatingActions` is a function of the two configurations and the L label:

- a tick, a request, and a batch of fire actions are one Dex action each (`tick`, `request`, `timerFire`);
- a thread that begins an action is the [STEP-START] of its Step;
- $out(a)$ is the `workerEffect` of the thread's task, after the `workerRun` that runs the handler if it has not run;
- a thread that completes an action is the `stepComplete` of its task, after the `workerRun` and `workerReturn` that have not happened yet;
- after the close, the delivery or the drop of a late emit is the `workerEffect` or the `workerAbandon` of the first task that still holds the effect.

So one L label is one to three Dex actions. The step lemma, proved by cases on the label (`l_step_simulatedByActions`, `Simulation/LToDex.lean`), says that these actions are enabled, have the same projected events, and end in a related configuration:

```
def IsSimulatedByActions (program : Program) (lConfig : L.Config)
    (dexConfig : Dex.Config signature) (label : L.Label) (lEvents : List Event)
    (lNext : L.Config) : Prop :=
  ∃ dexEvents dexNext,
    (Dex.system ⟦program⟧).exec dexConfig (simulatingActions program lConfig dexConfig label) =
      some (dexEvents, dexNext) ∧
    dexEvents.filterMap project = lEvents ∧
    Related program lNext dexNext
```

The bulk of the proof is here. Registering a wait (`Simulation/Register.lean`) and completing an action (`Simulation/Completion.lean` and `Simulation/Completion/`) must re-establish every clause of `Related` after a commit that may write an Attribute, publish tokens, wake the earliest waiter, create Steps, or close the Flow.

6.3 From Dex to L: at most one label, by determinism

In the other direction, every Dex step of an admissible run is simulated by at most one L label (`dex_step_simulated`, `Simulation/DexToL.lean`):

```
def IsSimulatedByL (program : Program) (lConfig : L.Config)
    (dexEvents : List (Dex.Obs signature)) (dexNext : Dex.Config signature) : Prop :=
  ∃ labels lNext,
    labels.length ≤ 1 ∧
    (L.system program).exec lConfig labels = some (dexEvents.filterMap project, lNext) ∧
    Related program lNext dexNext
```

Running a handler, returning it, and dropping a task that holds no effect have no L label: the list is empty and the relation is kept. For the actions that L sees, the proof does not repeat the case analysis of the relation. It reuses the other direction:

1. Show that the corresponding L label is enabled.
2. Apply the L-to-Dex lemma. It names the simulating Dex actions.
3. Show that those actions are the given action alone. For a Worker effect or a completion, the given action is enabled only when the Worker steps before it have already happened, so the named list has no prefix.
4. `Dex.step` is a function. The given step is therefore the simulating step, and its result is the related configuration of step 2.

This is why the L-to-Dex statement names its actions: an existential statement could not be matched against a given Dex step. Compiler verification uses determinism for a similar economy: with a deterministic target, preservation proved in one direction suffices [17].

Step 1 is the only place where well-formedness is used. At [STEP-START] of a Step whose statement requires a mutex, the L thread can begin only if it holds the mutex; the hypothesis that `BreaksMutexRequirement` does not hold says that it does. Consequently the two L-to-Dex inclusions, `finiteTrace_l_to_dex` and `fairMaximalTrace_l_to_dex`, hold for every program; only the Dex-to-L inclusions need the scope.

6.4 Safety

Safety follows by induction on a finite run (`FiniteTraces.lean`). The initial configurations are a simulation state (`related_initial` and the initial cases of the invariants). Each step of one side is matched by the step simulation, and the invariants are carried along by their own step lemmas. The Dex-to-L induction also carries reachability of the L configuration, to apply well-formedness, and admissibility of each Dex action, which follows from the admissible prefix. The two inclusions are `finiteTrace_dex_to_l` and `finiteTrace_l_to_dex`.

6.5 Liveness

Each inclusion of liveness builds an infinite run of one side from an infinite run of the other, and then carries over the input word, the trace, the marker, and the fairness assumptions.

Runs in blocks. The two systems do not step in lockstep. One L label is one to three Dex actions, and a Dex Worker step is no L label. A run of one side therefore gives the other side a *run in blocks* (`BlockRun`, `Core/BlockRun.lean`): one finite, possibly empty, list of labels per step of the first (Figure 5). A block run with infinitely many non-empty blocks *flattens* into an ordinary infinite run (`BlockRun.IsFlattenedTo`),

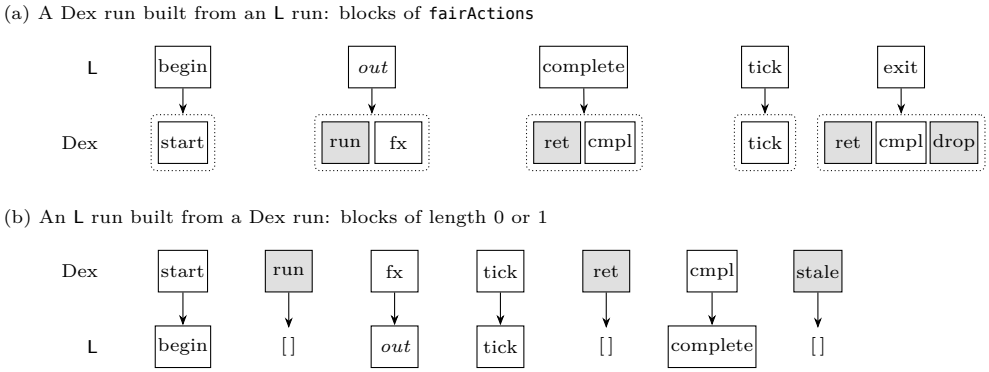


Figure 5: Runs in blocks. Dotted outlines are blocks. Shaded Dex actions are of a *silent kind*: L has no label for them. (a) Each L label becomes its simulating actions, followed, after the close, by the drops of the canceled tasks that hold no effect. (b) Each Dex action becomes at most one L label; silent actions become empty blocks.

and the flattening keeps the trace, the attempted requests, and whether the progress steps are finitely many. This part mentions neither Dex nor L.

Dex to L: the silent weight. Given a fair Dex run, `dex_step_simulated` gives an L run in blocks of length 0 or 1 (`LFollowsDex`). The danger is an infinite tail of empty blocks: the Dex run would have infinitely many progress steps (∞) while the L run has finitely many. The proof rules it out with a measure. The actions that can have no L label are of four kinds: `workerRun`, `workerReturn`, `workerAbandon`, and `stepCompleteStale`. A task can take at most three such steps (run, return, drop), and the *silent weight* of a configuration counts what is left:

```
def Dex.Task.silentWeight {σ : Dex.Signature} (task : Dex.Task σ) : Nat :=
  match task.phase with
  | .started => 3
  | .running _ _ => 2
  | .returned _ => 1

def Dex.Config.silentWeight {σ : Dex.Signature} (config : Dex.Config σ) : Nat :=
  (config.tasks.map Dex.Task.silentWeight).sum
```

Every action of a silent kind strictly lowers the weight (`silentWeight_lt_of_isSilentKind`), a tick keeps it, and only a step that L sees can raise it. So infinitely many progress steps of the Flow include infinitely many that L sees, and the L labels of a Dex step are progress steps exactly when the action is one. The markers then agree. F1 gives infinitely many ticks, hence infinitely many non-empty blocks, so the flattening exists (`exists_lRun_of_dexRun`).

L to Dex: cleaning up after the close. Given a fair L run, the simulating actions give a Dex run in blocks (`DexFollowsL`). One more step is needed for fairness. When the program ends, the Flow close cancels every active Step, and the tasks of those Steps stay in Θ. A stale task that still holds an effect corresponds to a late emit, and L's own fairness delivers or drops it. A stale task that holds no effect has no L step at all, yet F4 requires it to leave Θ. So the Dex run does not use `simulatingActions` alone. It uses `fairActions`: the simulating actions, followed by the [WORKER-ABANDON] of every canceled task that holds no effect (`cleanupActions`). Between two blocks no such task and no *Returned* task is left.

Fairness and markers carry over. For each direction, one lemma per assumption shows that the built run is fair when the given run is. The correspondences are those of Table 6: a thread whose next action has not begun is a pending invocation (F3); a thread that issued an emit or began an action is a task, with an effect exactly when it issued an emit (F4, F5); a due timer is a due timer (F2); a late emit is an effect of a stale task (F4). The markers carry over because closed corresponds to closed in `Related` and because flattening

keeps whether the progress steps are finitely many. The two inclusions are `fairMaximalTrace_dex_to_l` and `fairMaximalTrace_l_to_dex` (`Liveness.lean`).

6.6 Non-vacuity

A theorem with a hypothesis is only as good as the set of objects that satisfy the hypothesis. `Program.IsWellFormed` quantifies over every reachable configuration, so a concrete trace does not establish it, and without a witness the theorems could hold for no program at all.

A static lock discipline. `L/LockDiscipline.lean` defines a decidable, syntactic, sufficient condition, `Program.hasStaticLockDiscipline`: a must-hold lock-set analysis over the statically known mutexes. lock(staticMutex(i)) adds i to the set. unlock, wait, signal, and broadcast need a statically known operand whose mutex is in the set. The two branches of an if or a select must agree. A loop body must give back the set it started with. A called or forked procedure body, a try body, and a handler are checked from the empty set. `L/LockDisciplineSound.lean` proves soundness:

```
theorem Program.isWellFormed_of_hasStaticLockDiscipline (program : Program)
    (isDisciplined : program.hasStaticLockDiscipline = true) : program.IsWellFormed
```

The proof rests on one fact of L: only the owner of a mutex releases it, so the steps of other threads never shrink the set of mutexes a thread holds.

Well-formed programs exist. `Examples/WellFormed.lean` proves `IsWellFormed` for eight programs, each by evaluating the discipline in the Lean kernel. Together they use every construct of L: a two-thread mutex program; a program with sleep, a timed select, recv, cancel, fork, and join; a wait/signal program; a wait-in-a-loop/broadcast program with shared variables; a recursive program; a program with try, a throw across a call, and an uncaught exception; three threads racing in select; and a program that creates and locks a mutex and then fails. `Examples/WellFormedSafety.lean` states the safety theorem at three of them with no hypothesis left, for example:

```
theorem mutexProgram_safety (trace : List Event) :
    IsProjectedDexFiniteTrace Mutex.program trace ↔ IsLFiniteTrace Mutex.program trace :=
  safety_of_isWellFormed Mutex.program mutexProgram_isWellFormed trace
```

The trace sets are not empty either. `Examples/Mutex.lean` shows, by running explicit schedules in the kernel, that both complete traces of the two-thread mutex program are finite traces of the program and projected finite traces of its Flow (`bothTraces_onBothSides`). `Examples/TimedSelect.lean` and `Examples/TimerOrder.lean` do the same for timers, select, cancellation, and two timers with equal deadlines.

Fair maximal traces exist. The trace sets of liveness are defined by the existence of a fair run. Without a fair run of some program, `liveness_of_isWellFormed` would compare two empty sets. `Liveness/RestingRuns.lean` builds fair maximal runs of L. A configuration is *at rest* (`L.Config.isAtRest`) when every thread is halted or blocked without a timer and no late emit is left; then none of F2 to F5 speaks about it. A run that reaches a configuration at rest by a finite schedule, and then takes only steps that keep it at rest while time diverges, is fair (`IsRestingTail.isLFairMaximalTrace`). Two such tails are built: clock ticks alone, and clock ticks alternating with inputs that are refused after the close.

`Examples/FairRuns.lean` gives the witnesses of Table 7. Each finite schedule is checked by evaluation in the kernel. Each L trace is then also a projected fair maximal trace of the translated Flow, by instantiating `liveness_of_isWellFormed` at a well-formed program and an admissible word. So both sides of the liveness equivalence are inhabited for each marker.

One limit of these witnesses must be stated precisely. The ∞ witness has an *infinite input word*: the program exits, and every later input is refused, forever. Those refusals are the infinitely many progress steps. It is not a program that diverges on its own. No witness is checked for a program that runs forever on a finite word, such as a loop that keeps emitting: its configurations never come to rest, so its run is not a finite schedule followed by a regular tail. There is also no general lemma that every reachable configuration of a well-formed program has a fair maximal continuation.

Marker	Program	Input word	L side; Dex side
✓	the two-thread mutex program	the start	`mutexProgram_terminates`; `mutexProgram_terminates_dex`
δ	emit(1); recv(5)	the start	`waitingProgram_deadlocks`; `waitingProgram_deadlocks_dex`
δ	lock(m_0); emit(1); lock(m_0)	the start	`selfDeadlockProgram_deadlocks`; `selfDeadlockProgram_deadlocks_dex`
∞	the two-thread mutex program	the start, then one input forever	`mutexProgram_diverges`; `mutexProgram_diverges_dex`

Table 7: Kernel-checked fair maximal traces (`Examples/FairRuns.lean`). The Dex-side theorems follow from the L-side ones by the liveness theorem.

7 Engineering the proof

This section reports how the result was produced, including what went wrong. We think the method is reusable, and the failures are the informative part.

7.1 Stages and fixed statements

The work had three stages.

Stage 0: the semantics. The specification was written against the server code at a pinned commit, with the rules citing the files and lines that implement them, and every modeling choice marked as an assumption or a decision. There is one deliberate exception: the rule for the commit of an RPC follows a fix that was not yet merged at the pinned commit (the first defect of §7.4), and is marked as doing so. Three adversarial reviews checked it against the code. They produced 21 findings; one was rejected with an argument from the code (the prefix rule of [TIMER-FIRE]), and the others changed the specification. The additions include the fair maximal traces with markers, the admissible environment and the projection, attempted requests as the input word, and the late, ordered fire action of L. The specification ends with a list of 41 numbered assumptions and decisions.

Stage 1: the model and the statements. The Lean model, the translation, and the two theorems were written first, with the theorems reduced to inclusion lemmas that ended in `sorry`: twelve in all. A review compared the statements with the specification before any proof started. That review led to the owner decisions L1 to L4 of §3 and to two corrections of the model: snapshots had loaded every Attribute, where the server loads an AttributeMap instance only when it is selected; and a runtime-only instruction had been exposed as a source statement.

Stage 2: the proofs. The definitions that the proofs share were *fixed*: the relation, the invariants, the shape of each step lemma, and later the liveness definitions. The remaining work was cut into packages with fixed statements (Table 8), which were proved in parallel on separate branches and then merged. A package may add lemmas. It may not change a model file, a fixed definition, or a statement.

Fixing the statements first has an obvious benefit: independent provers cannot drift apart. It also has a cost, which we met at once. A step lemma quantifies over *every* pair of configurations that satisfies `IsSimulationState`, not only over reachable pairs. If the fixed definitions admit an unreachable pair that breaks a step, the lemma is false, and the prover of that package is the one who finds out.

7.2 Six gaps, and a fuzz that finds them

That is what happened. The first fixed definitions were not inductive. The provers found six pairs of configurations that satisfied the definitions, were not reachable, and falsified a step lemma (Table 9). Each gap was closed by one more clause.

A seventh problem was in the shape of a statement, not in an invariant. The L-to-Dex lemmas first concluded with an existential ("some Dex actions simulate the label"). The Dex-to-L direction could not

Package	Content	Main statement
WP1	Every L transition keeps the L invariant.	`L.Invariant.step`
WP2	L to Dex: ticks, requests, timer fires, late emits, the begin of an action, the delivery of an effect.	`l_tick_simulated`, `l_fire_simulated`, …
WP3	L to Dex: the registration of a wait and the completion of an action.	`l_thread_register_simulated`, `l_thread_complete_simulated`, …
WP4	Dex to L: every Dex step.	`dex_step_simulated`
L1	Block runs flatten; the flattening keeps the trace, the requests, and finiteness of progress.	`exists_flattening`
L2	Silent actions lower the silent weight; labels are progress steps exactly when the action is one.	`silentWeight_lt_of_isSilentKind`
L3	The L run of a Dex run: existence, word, trace, marker.	`exists_lRun_of_dexRun`
L4	The L run of a fair Dex run is fair.	one lemma per assumption
L5	The Dex run of an L run: existence, word, trace, marker.	`exists_dexRun_of_lRun`
L6	The Dex run of a fair L run is fair.	one lemma per assumption

Table 8: The work packages. None of them changed a definition of a model file, a definition of `TraceSets.lean`, or a statement of `Theorems.lean`.

reuse them by determinism, because the actions were hidden. The statement was changed to name them (`simulatingActions`, §6.2).

Finding such gaps by proving is expensive: each one stops a package, and the fix touches definitions that other packages rely on. So after the fact we built the test that would have caught them: an *inductiveness fuzz* (`Examples/Inductiveness.lean`), in the spirit of testing a property on random inputs before trying to prove it [5].

1. It runs an L program and its Flow in lockstep, in both directions, on pseudo-random schedules, and samples the reached pairs of configurations.
2. It *perturbs* each pair in about 270 ways: one field of one side (a thread's phase, counter, timer, locals, or code; the cancel flags; the mutex owners; a Step's phase, invocation, wait, or input; a task's phase or snapshot; added, dropped, or doubled Steps, tasks, messages, timers, Attributes), or matching fields of both sides.
3. It keeps a perturbed pair only when an executable Boolean mirror of `IsSimulationState` (`checkState`, `Examples/RelatedCheck.lean`) accepts it. The kept pairs are mostly unreachable, which is the point.
4. From every kept pair it takes every enabled L label with its simulating actions, and every enabled admissible Dex action with its L labels. The other side must be enabled, the events and clock stamps must agree, and the mirror must accept the result.

The result is the honest measure of the method. With the mirror of the *first* definitions, the run `inductiveness-fuzz 120 5 6 2` reports 322 failures, and they cover all six gaps of Table 9. With the present definitions the same run, on 13 programs, perturbs 1,898 pairs into 510,426 copies, keeps 189,045 of them, and reports no failure. A larger run on the finished proof, `inductiveness-fuzz 200 4 6 3`, perturbs 5,876 pairs into 1,820,177 copies, again with no failure. The fuzz found no gap beyond the six. It is evidence, not proof; the work packages prove inductiveness. But the smaller run takes well under a minute, and had it existed before the statements were fixed, none of the six gaps would have reached a prover.

Writing the fuzz also taught a lesson about the tooling. The first mirror of the timer clause checked every binding of the timer list, where the clause itself reads only the first binding of an ID. A mirror that is stricter than its clause *hides* gaps: the duplicate-timer gap was invisible until the mirror was made exact.

Gap	A pair the first definitions allowed	Closed by
WaitFor task with an effect	A thread that began a wait, whose WaitFor task holds an effect: the Worker's effect step has no L step.	`PhaseMatchesStep`: such a task holds no effect.
Task effects	After the close, a stale task holds two effects: abandoning it drops both, L drops one late emit per label.	`IsTaskComputed`: a task holds what its handler computes, at most one effect.
Waiting Step with an invocation	A thread that is not blocked, carried by a Step in phase *WT* with a stored wait: a publication wakes the Step and not the thread.	`PhaseMatchesStep` fixes the Step's phase in every case.
Cancel flag of an unforked child	A flag is set for the next child of a live thread, with no pending request: after fork the child has a flag and no request.	`cancelFlagsAreRequests`.
Boot and static mutexes	While main boots, the mutex owners and the token Channels are unconstrained, and the start Step then publishes the tokens.	`mutexesBeforeBoot`.
Duplicate timer IDs	The timer list holds one ID twice: the relation reads the first binding, the due-timer computation reads both.	`TimersAreDistinct`.

Table 9: The six gaps in the first fixed definitions. Each row is an unreachable pair that the definitions accepted and that breaks a step lemma.

7.3 Testing before proving

The same habit was applied throughout. None of these checks is part of the proof. All of them run when the development is built, and a failure fails the build.

Lockstep checks (`Examples/Lockstep.lean`, `Examples/LockstepChecks.lean`) run a program and its Flow side by side on pseudo-random schedules, in both directions, for 13 programs that together use every construct. After every step the L events must equal the projected Dex events, and the L clock stamp must equal the Dex commit timestamp. Three faults were planted to see whether the build notices. Dropping the cancel of a blocked timed operation failed. Dropping the load selection of the waiter list failed. Dropping the timer-number tie-break *passed*: equal deadlines are rare under random ticks. A dedicated example with two equal deadlines (`Examples/TimerOrder.lean`) was added and fails under that fault.

Relation checks (`Examples/RelatedCheck.lean`) evaluate the mirror of `IsSimulationState` in every pair that the lockstep runs reach, for 14 programs.

Liveness checks (`Examples/LivenessChecks.lean`) tested the liveness definitions before the liveness packages started: that `fairActions` are enabled and end in a settled pair; that the fairness obligations of the two sides correspond; that every silent action lowers the silent weight; and that, in a settled pair, L has a step other than a tick or a request exactly when the Flow has one, which is what the markers ✓ and δ compare. The run `inductiveness-fuzz 200 4 6 3 liveness` checks 1,396,813 pairs, 1,205,008 of them settled and 408,933 settled after the close, with no failure. Each of the four checks was shown to detect a planted fault.

These checks cannot test a statement about an infinite run, and they did miss one thing. A fixed statement of package L2 was false as first written: that every silent action lowers the silent weight. With two tasks of one ID it fails, because an update by ID changes both. The statement now assumes `TasksAreDistinct`, which every configuration of a run satisfies. The checks did not find this because they start from pairs that the mirror accepts, and those have distinct task IDs. The prover found it.

7.4 What the work found in the real server

Writing a rule for every server behavior, with a citation, means reading every caller of every function the rule mentions. Two defects in the real server surfaced this way, before any Lean was written.

- *A cancellation that skipped the graceful-completion check.* A Flow that has requested graceful completion closes once no Step is active. One path that cancels Steps, cancellation by an RPC, did not run that check, so a Flow whose last active Step was canceled this way stayed running forever.

Directory	Files	Lines	Content
`Core/`	4	1,177	transition systems, traces, block runs
`Dex/`	9	3,378	the Dex model, its run invariant, lemmas
`L/`	11	6,033	the L model, its invariant, the lock discipline
`Translation/`	2	433	the translation, the projection, E_{adm}
`Simulation/`	32	15,862	the relation and the step simulations
`Liveness/`	16	6,435	the liveness definitions, packages L2 to L6, resting runs
`Examples/`	13	2,692	kernel-checked runs, witnesses, checks, the fuzz
top level	9	2,076	invariants, finite traces, the theorems
Total	96	38,086	

Table 10: The Lean development at commit `33e998003`. The specification is a further 2,005 lines of Markdown.

- *Consumed messages lost when a waiting Step was canceled.* A Step whose wait had consumed Channel messages, and which was canceled before its Execute committed, discarded those messages. A review finding showed the consequence for a token-based mutex: the token is gone and the mutex stays locked.

Both have since been fixed in the server. The first is outside the translation, which uses neither graceful completion nor RPCs. For the second, the fix changed how the server handles the consumed messages of a canceled Step. A translated Flow cancels Steps only by closing, and we argue that the changed handling is unobservable there; but the model describes the server at the pinned commit, and we have not proved that the current server refines it (§9). The second defect also shaped L: it is why L's cancellation is cooperative and lock is not a cancellation point. The specification also records where the product documentation said less than the code does; the plainest case is "a Step method commits atomically", which is true of the writes and not of the read-modify-write (§2.3).

7.5 Size and effort

Table 10 gives the size of the development, counted with `wc -l`, comments and blank lines included. The definitions that a reader must trust are about a tenth of it. The largest part by far is the L-to-Dex step simulation; its largest file, the registration of a wait, has 1,941 lines.

The development uses Lean 4.35.0 and no library beyond the Lean core: no Mathlib and no Batteries. The list and arithmetic facts it needs are in the core or are proved in the development. Finite maps are association lists, so that every model function evaluates in the kernel and the examples are checked by evaluation. A clean build takes a few minutes at most; the two fuzz runs quoted above took about half a minute and two minutes when we reran them for this paper.

The proofs were produced by AI coding agents working in parallel on the work packages with fixed statements, under the author's direction. The author made all semantic decisions. By their git author dates, the 92 commits under `formal/` up to the revision described here fall between 13:39 and 23:11 on one day: from the first committed draft of the specification to the complete proof took a little under ten hours. That figure excludes earlier preparation: planning, the Workflow Patterns evaluation, and drafting the specification before its first commit.

8 Failure transparency

The theorems of §5 cover failure-free runs. This section adds server crashes, standby takeovers, Worker failures, retries, zombie Workers, duplicate results, and StartFlow retries, and proves the second theorem in the same fragment under the hypotheses of §8.2. For finite traces it gives two inclusions: erasing the repeated external effects of a trace with failures gives a failure-free trace for the word without retries, and every failure-free trace is a trace with failures. For fair maximal traces with markers the first inclusion holds for every admissible word, and the second for words without StartFlow retries. Composed with the first

Rule	What happens
[CRASH]	The server stops. Committed state, timers, the clock, and the zombies stay; every running task becomes a zombie; engine memory is lost. A log write with an unknown outcome is a crash.
[RECOVER]	The server restarts from the log. Every attempt that was Running joins I; it is never handed out again.
[RECOVER-INTERRUPT]	A separate commit per element of I: the attempt fails and goes to Backoff, or is ignored if a fence rejects it.
[ATTEMPT-LOST]	An attempt fails at any time (Worker error, method or heartbeat timeout, disconnect, server error). It keeps its snapshot, arms a retry timer, and its task becomes a zombie.
[ATTEMPT-EXHAUSTED]	The same failure when the retry policy allows no further attempt, or a non-retryable one: the Flow closes as Failed.
[ATTEMPT-RETURNED]	The Worker rejected the task; it goes back to Pending with the same attempt number and snapshot.
[ATTEMPT-HEARTBEAT]	A heartbeat checkpoint commits.
[RESULT-STALE]	A result, failure, or heartbeat whose fence (Step execution, method, attempt number, phase, task ID) no longer matches is ignored. Duplicate proposals are stale.
[RETRY-FIRE]	A due retry timer moves a Backoff invocation back to Pending, with its retained snapshot. It is part of [TIMER-FIRE].
[ZOMBIE-EFFECT]	A zombie emits its next external effect, or is dropped.
[START-RETRY]	A StartFlow with the start's request ID and ignore_already_started_error returns the existing run. It is a stamped log entry and writes no run state.
[ENGINE-REJECT]	After the run was purged, the engine rejects a publication before proposing anything, with the event that apply's rejection would give.

Table 11: The failure rules of DexF (specification §11.3). They emit no event, except [ATTEMPT-EXHAUSTED] (*closed*), [ZOMBIE-EFFECT] (*fx*), [ENGINE-REJECT] (*rejected*), and [RETRY-FIRE] when its command also fires wait timers, which then acts as [TIMER-FIRE].

theorem, it relates a well-formed L program to its execution on a failing server, where each extra $out(x)$ is known only to equal an earlier $out(x)$ in value.

The failure model is Section 11 of the specification (revision 9), written against the server at a later commit than the failure-free rules, again with a citation for every rule. The failure-free rules are unchanged, so the first proof stays valid. In Lean the model is *DexF* (`Failure/`); the failure-free model of the earlier sections is called *Dex* below.

8.1 The failure model

DexF is the Dex model plus four pieces of state and a set of failure rules. The state records whether the server is up; a set Z of *zombie* tasks, Worker tasks that the server gave up but that may still run; a set I of interrupted attempts, which recovery rebuilds; and, per invocation, an attempt number, a phase Pending, Running, or Backoff, a task ID, and a *retained snapshot*. A Step method's attempts share one snapshot: the server takes it at the first start and every retry reuses it. Table 11 lists the rules.

Three choices make the model larger than the code, the safe direction for an inclusion theorem: more DexF runs are more obligations. Failures are nondeterministic and can happen at any time, with no model of how the engine detects them. A zombie can emit effects at any later time, even after a newer attempt of the same method has started. A zombie's result can commit after a crash and before its interrupt, an order that the code makes unlikely.

Durability. The durable point of a commit is its log batch in object storage, written create-only. Recovery replays the same stamped commands through the same apply function as live commits; it is not a special mode. A standby takeover is a crash of the old owner followed by recovery on the new one. The new owner

writes an epoch marker at the first free log position, and replay voids every later batch of the old epoch, so the log has one writer at a time. The old owner may still carry frames to Workers until it stops; those tasks are zombies, and their results cannot reach the new owner.

8.2 Scope, decisions, and hypotheses

We fixed the scope before any proof (decisions D1 to D8 and OD1 of the specification). Table 12 lists each assumption and how the Lean model treats it.

Four entries hold by construction. The Lean model has no failure policy, no ASYNC start, and no purge of a closed run, and its task IDs are minted numbers. The theorem excludes those three features, and the named predicates make the exclusion visible in every statement. H3 is the one real hypothesis on the program. A DexF program is a function from the attempt context to a Flow definition, so a program that reads its attempt number is expressible, and the theorem excludes it explicitly.

D5, finitely many failures, is necessary: no system makes progress under infinitely many crashes. D6 needs no clock agreement between hosts. Stamps increase strictly after a takeover too, because the new owner seeds its watermark from the largest applied stamp. The timer order of OD1 is an idealization. The specification found one window, present without any failure, in which the code fires a timer while an earlier-due timer, armed by a command applied just before, is still armed (Open question 26). We kept the prefix rule for both theorems so that they compose; neither covers runs in that window.

8.3 Repetitions and erasure

DexF tags every *fx* event with a ghost index: $fx(e, m, x)^j$ is the j-th effect of some attempt of method m of Step execution e. An event is *repeated* when an earlier event of the trace has the same source and index. $erase(t)$ removes the repeated events of a trace t (`eraseRepeats`). A trace t is a *repetition* of u when t is u with inserted events, each equal in source and label to an earlier event of u (`IsRepetition`).

First occurrences come in order. The theorem rests on one fact: for each method, the first occurrences of the indexes $1, 2, 3, \ldots$ appear in this order, with the labels of one effect list, even when zombies overlap with newer attempts. Three facts give it.

1. Every attempt computes the same effect list and response. The handler is a function, and its arguments are equal across attempts: the snapshot is retained on every retry path (an ordinary failure, a restart or takeover interrupt, a returned attempt, a retry fire, the next start); the Step input does not change while a method is live; and H3 makes the attempt context irrelevant.
2. An attempt emits its effects in list order.
3. So when index j first appears, some attempt has already emitted 1 to $j - 1$, and each of them has appeared.

The first occurrences are therefore the effects of one *virtual* task of Dex, and every other *fx* event of the method equals the first occurrence of its index. D2 is needed for the first fact: under ASYNC a crash can lose a start that a Worker already received, and the rerun takes a new snapshot, so its effect list can differ.

8.4 The abstraction map

The proof is a forward simulation of DexF by Dex under a function α from DexF configurations to Dex configurations (`abstractConfig`, `Failure/Abstraction.lean`). Every failure transition maps to a stutter, every other transition to Dex transitions with the same events, and the first occurrence of each effect index to a Worker effect of the virtual task.

α forgets attempt numbers, task IDs, heartbeat checkpoints, retry timers, Z, I, and whether the server is up. It decides by the snapshot, not by the phase: an invocation that holds a snapshot, also in Backoff or Pending after a failure, maps to a running invocation with one virtual task; an invocation without one never started and maps to a pending one. This is sound because a retrying invocation looks to apply exactly like a running one: same snapshot, same input, same locals.

The virtual task cannot be read off a live task. The attempt that went furthest may be gone, and two runs can reach the same durable state with different histories, for example an attempt that returned an empty effect list and was then ignored after a failure, against one that failed before it ran. So DexF keeps *ghost* history variables (`Ghost`). In the specification no rule reads them. In Lean the step function reads them only to check the classification Booleans that a label carries, such as whether an effect is a first occurrence, and to update them; α is their only other reader:

- $h(e, m)$, the largest effect index any attempt emitted, and $v(e, m)$, the furthest phase any attempt reached: not run, ran with a given effect list and response, or returned. Both are monotone. The virtual task has emitted the first $h(e, m)$ effects and is in phase $v(e, m)$.
- $par(e, m)$, the handler's arguments at the first start. A Flow close clears an invocation and releases its snapshot while a zombie may still emit its first effect. α then keeps a canceled virtual task built from par, v, and h, exactly like a canceled task of Dex, until the last real task of the method is gone.
- n and ζ: failure commands take run sequence numbers that Dex never sees. ζ maps the sequence number of the k-th applied normal command to k, and n counts them, so α relabels every sequence-derived value of the model, such as the enqueue order of messages. ζ must be state, not a function of it: two runs can reach equal states, one through a normal Step start and one through a heartbeat at the same sequence number, and need different relabelings.

The specification classifies every command by apply's outcome. A command takes a run sequence number exactly when it advances the run's persisted sequence counter: normal and failure commands do, bookkeeping commands do not. *Normal* commands map to Dex commits; *failure*, *bookkeeping*, internal ignored, and post-close housekeeping commands map to stutters; a rejected client request maps to the same Dex rejection with the same event. Worker steps follow the ghosts: a first occurrence maps to a step of the virtual task and a repeat to a stutter. When the last real task of a canceled invocation leaves, its drop maps to Dex [WORKER-ABANDON] and its ignored result, if the method had returned, to Dex [STEP-COMPLETE-STALE]; both are progress steps. The simulation needs seven invariants (`Invariant`, `Failure/Relation.lean`): one committed result per invocation; one effect list per method; every task has emitted a prefix of length at most h; retained state belongs to invocations that started; the abstract state equals the Dex state up to relabeling; every Backoff invocation is pending and has its retry timer armed, while stale retry timers may remain; and interrupts name minted tasks.

8.5 The theorems in Lean

The trace sets are `IsFailureFiniteTrace` and `IsFailureFairMaximalTrace` (DexF runs that satisfy H1 and H4, for an admissible outcome word, and for liveness D6, HP, and the fairness of Table 12), and `IsFreeFiniteTrace` for Dex (`Failure/Observation.lean`). A DexF word may contain StartFlow retries; $w{\downarrow}$ (`eraseRetries`) deletes them. The statements are in `Failure/Statements/Main.lean`.

Safety. Erasing the repeated effects of a finite DexF trace gives a finite Dex trace for the word without retries:

```
theorem safety_failure_forward (program : Program σ)
    (hasNoFailurePolicy : HasNoFailurePolicy program) (isSyncOnly : IsSyncOnly program)
    (ignoresAttemptContext : IgnoresAttemptContext program) (environment : Dex.Environment σ)
    (requests : List (Request σ)) (trace : List (Obs σ))
    (isTrace : IsFailureFiniteTrace program environment requests trace) :
    IsFreeFiniteTrace program.failureFreeFlow environment (eraseRetries requests)
      (eraseRepeats trace)
```

`safety_failure_repetition` adds that the DexF trace, with the ghost indexes dropped, is a repetition of that Dex trace. `freeFiniteTrace_isFailureFiniteTrace` is the converse inclusion $Tr_{free} \subseteq Tr_{fail}$: every Dex trace is a DexF trace with no failure. This is not trivial, because DexF carries more state. It is the only backward statement for finite traces. The converse of the forward statement is false: not every repetition is reachable. An attempt runs from the start and emits its effects in order, so index j never has more copies than index $j-1$. For an effect list $[a, b, c]$, one extra c without an extra a and b before it is a repetition that no DexF run produces.

Liveness. A DexF step is a progress step exactly when its image under the transition map is a Dex progress step (`isProgress_eq_dexProgress`). Failures, stale results other than that of the last task of a canceled invocation, StartFlow retries, and repeated effects are therefore not progress, and infinitely many of them do not make a run divergent.

```
theorem liveness_failure (program : Program σ)
    (hasNoFailurePolicy : HasNoFailurePolicy program) (isSyncOnly : IsSyncOnly program)
    (ignoresAttemptContext : IgnoresAttemptContext program) (environment : Dex.Environment σ)
    (word : Sequence (Request σ)) (isAdmissible : IsAdmissibleOutcomeWord environment word)
    (erasedWord : Sequence (Dex.Request σ)) (isErasure : IsRetryErasure word erasedWord)
    (trace : Sequence (Dex.Obs σ)) (marker : Marker) :
    ((∃ failureTrace, IsFailureFairMaximalTrace program word failureTrace marker ∧
        IsRepeatErasure failureTrace trace) →
      (Dex.system program.failureFreeFlow).IsFairMaximalTrace Dex.IsFair some erasedWord trace
        marker) ∧
    (HasNoRetry word →
      (Dex.system program.failureFreeFlow).IsFairMaximalTrace Dex.IsFair some erasedWord trace
        marker →
      ∃ failureTrace, IsFailureFairMaximalTrace program word failureTrace marker ∧
        IsRepeatErasure failureTrace trace)
```

The forward inclusion holds for every admissible outcome word, markers included. The backward inclusion holds for words without retries, and it can fail for some words with them. A deduplicated StartFlow retry is a log entry, and its stamp strictly advances the partition clock. A word with 1,000,001 retries between two publications therefore forces more than a second between them. A failure-free run can arm a one-second timer of one Step and, before it is due, a zero-second timer of another Step that the second publication starts, and fire the second timer first. With the retries in between, the first timer is due first and takes the only message both wait for, and no DexF run for that word has the failure-free trace. The kernel checks the parts the model can state (`Failure/Examples/RetryTiming.lean`): every StartFlow retry advances the commit stamp, so n retries put at least n microseconds between two publications, and in a failure-free Flow a delay of 1,000,001 microseconds changes the trace. The last step, a statement about every DexF run of a word with a million labels, remains an argument by hand. The specification was corrected to the same two inclusions after the review of the Lean statements.

End to end. `endToEnd_safety` and `endToEnd_liveness` compose these with `expressiveness_of_isWellFormed`. For every well-formed `L` program S, every `L` trace of S is the projection of a finite DexF trace of $[\![S]\!]$, and the projection of every finite DexF trace of $[\![S]\!]$ is an `L` trace of S with inserted $out(x)$ events, each equal in value to an earlier $out(x)$:

```
theorem endToEnd_safety (program : L.Program) (isWellFormed : program.IsWellFormed) :
    (∀ lTrace, IsLFiniteTrace program lTrace →
      ∃ requests failureTrace,
        IsFailureFiniteTrace (Program.ofFlow ⟦program⟧) environment requests failureTrace ∧
        failureTrace.filterMap (fun event => project event.toDex) = lTrace) ∧
    (∀ requests failureTrace,
      IsFailureFiniteTrace (Program.ofFlow ⟦program⟧) environment requests failureTrace →
      ∃ lTrace, IsLFiniteTrace program lTrace ∧
        IsOutRepetition (failureTrace.filterMap (fun event => project event.toDex)) lTrace)
```

The projection drops the source of an effect, so on the `L` side an inserted $out(x)$ is only known to equal some earlier $out(x)$. For fair maximal traces the statement is stronger: the projection of $erase(t)$ is a fair maximal `L` trace with the same marker, forward for every admissible outcome word and backward for words without retries.

What the kernel checked. The proof contains no `sorry` and no `native_decide`. For `safety_failure_forward`, `safety_failure_repetition`, `freeFiniteTrace_isFailureFiniteTrace`, `liveness_failure`, `endToEnd_safety`, `endToEnd_liveness`, and the leaf lemmas `step_simulated`, `invariant_step`, and `effectHistory_of_exec`, `#print axioms` reports `propext`, `Classical.choice`, and `Quot.sound`; for `isProgress_eq_dexProgress` it reports `propext` and `Quot.sound`. The statements of the failure theorem rest on about 1,900 lines of new definitions that a reader must check against the specification (`Failure/Config.lean`, `Rules.lean`, `Repeats.lean`, `Observation.lean`, and the main statements), on top of the Dex model. The abstraction map and the invariants are internal to the proof and need no trust.

8.6 The proof and its work packages

The proof followed the method of §7. The failure model was written against the code first, and repeated adversarial reviews checked it; its review log has 37 entries. Each finding was checked against the code and accepted. Among them were rejected client requests that had been made silent, a purge of closed runs that lets a late StartFlow retry start a second run, an abstraction that was not a function of the configuration, and a canceled invocation whose zombie could still emit its first effect. The Lean definitions and statements were then fixed, reviewed for vacuity and escape hatches, and tested with an inductiveness fuzz before any proof. The proof was cut into seven work packages with fixed statements (Table 13), proved in parallel. No package changed a definition or a statement.

Inductiveness fuzz. The fuzz (`failure-fuzz`) walks pseudo-random DexF runs with failures on a hand-written Flow and on the translations of eleven example `L` programs. At every step it checks the step simulation, the progress rule, and a Boolean mirror of the invariants. It then perturbs every fifth configuration in one place, keeps the copies that the mirror accepts, and checks every enabled step from them. With the arguments `300 5 10 12` it takes 31,160 walk steps with 194 crashes, 324 attempt losses, 189 starts on a retained snapshot, 27 repeated effects, and 118 StartFlow retries; it checks 8,425,297 single steps from 841,930 kept copies and reports no failure. A planted program that breaks H3 makes it report 404 failures. Three gaps in the first invariants were closed before any proof. The fuzz found the first: an interrupt could name a task ID not yet minted. Reviews found the other two, a Step execution ID reused when a Step counter was out of date, and a closed run that kept a running invocation; the extended fuzz now reproduces both.

Non-vacuity. Witnesses show that the hypotheses of `safety_failure_repetition` and of the forward inclusion of `liveness_failure` hold together for runs with crashes, retries, and zombie effects, and apply those statements to them (`Failure/Examples/Nonvacuity.lean`). Each asserts that its schedule is executable and that every step is simulated and keeps the invariants. The backward liveness inclusion and the end-to-end statements have no witness of their own.

- `safetyRepetition_isNonvacuous`: a run of the translation of $\mathsf{main}(x)\colon \mathsf{emit}(1); \mathsf{exit}(0)$ with a StartFlow retry, a lost attempt, a zombie effect, two retry fires, a crash and its recovery, an interrupt, a stale zombie result, and a zombie drop. Its trace has three *fx* events, and *erase* keeps one. It projects to the `L` trace $\mathit{begin}(7), \mathit{out}(1), \mathit{exit}(0)$ with two inserted $\mathit{out}(1)$.
- `livenessForward_isNonvacuous_terminated`: the same run followed by ticks, a fair maximal DexF trace with marker ✓ for a word with a retry.
- `livenessForward_isNonvacuous_deadlocked`: the translation of $\mathsf{main}(x)\colon \mathsf{emit}(1); x := \mathsf{recv}(5)$ with a crash, an interrupt, a lost WaitFor attempt, two retry fires, and two stale zombie results, then blocked forever: a fair maximal DexF trace with marker δ.

A finite schedule followed by ticks forever is a fair maximal run when three Boolean checks pass (`prefixThenTicks_isFailureFairMaximalTrace`). A further kernel-checked run takes the order the model allows and the fuzz never reached: a zombie's result commits after a crash and before its interrupt, which apply then ignores.

Counterexamples. Counterexamples show that H1 and H3 are needed and that a re-sent publication counts as a second request (`Failure/Examples/HypothesisCounterexamples.lean`). Without H3 (`attemptSensitive_isNotRepetition`), attempt a emits a copies of a; the retry emits $\mathit{fx}(2)$, which no failure-free run emits, so both safety statements fail. Without H1 (`exhaustion_isNotFailureFree`), an exhausted attempt closes the Flow as Failed, which no failure-free run of that Flow does, because it never decides ForceFail. The run of `nonRetryableFailure_isNotFailureFree` breaks H4 and also H1: in the model a non-retryable failure is an exhaustion, so H1 implies H4, and no counterexample shows H4 needed on its own. A re-sent publication (`resentPublication_isTwoRequests`) gives two *pub* events, which no failure-free trace of the one-publication word has, even as a repetition. H2, SYNC, and HP hold by construction in the model, and the specification argues from the code why each is needed. All witnesses and counterexamples depend on a subset of the three standard axioms.

Size. Table 14 gives the size of the failure development. It adds 88 files and 33,928 lines to the development of §7.5, and the specification grew to 3,534 lines. By their git author dates, the 88 non-merge commits of this work under `formal/` (95 with the 7 merges of work-package branches), from the first draft of the failure model to the last witness, fall between 10:58 and 16:16 on one day.

9 Limitations and threats to validity

A machine-checked proof removes one risk, an error in the argument. It leaves the others in place. We list them from the most to the least important.

The Dex side is a model, not the code. The semantics is written by hand from the server's source and validated by review. It is not extracted from the implementation, and no tool checks that the server refines it. A mistake in reading the code is a mistake in the theorem's meaning. Three things reduce the risk and none removes it. The rules cite the implementing code, except one RPC rule that follows a fix not yet merged at the pinned commit. Three adversarial reviews checked the rules against that code. And the model's executable form runs the examples and the lockstep checks, although only against itself, not against a running server. The Lean model is, further, a *restriction* of the specification to what the translation and the admissible environment use; each omission is listed in the development with the reason why it cannot matter. The specification also pins a base commit of the server. The server has changed since, including the two fixes of §7.4; the second changes what happens to the messages of a canceled Step, a behavior that a translated Flow reaches only at its close, where it is unobservable. That last sentence is an argument, not a theorem. The failure rules cite a later commit of the server and went through their own adversarial reviews, with the same caveat.

Failures: scope and hypotheses. The failure theorem of §8 replaces the failure-free assumption of the first theorem with a list of narrower ones. Each is a place where a real deployment can leave the theorem.

- *The Lean fragment.* One Flow, a start and publications, and StartFlow retries. RPCs, SubFlows, StopFlow, and Attribute locks under failures are not covered.
- *SYNC only.* An ASYNC Step method can lose a start in a crash and run the same attempt again with a new snapshot, so its effects can differ. That is the real ASYNC behavior, and the theorem excludes it.
- *H1 to H4.* No method exhausts its retries, no Step uses a failure policy, user code ignores the attempt number, the first attempt time, the last error, and the last heartbeat value, and no failure is non-retryable. A long outage can break H1: an interrupt after an outage longer than the total retry duration exhausts the method. Code that resumes from a heartbeat checkpoint breaks H3. The Lean model leaves failure policies and ASYNC starts out, so H2 and SYNC hold there by construction. The theorem does not cover Flows that use those features.
- *Finitely many failures for liveness.* Liveness also assumes that the server comes back after every crash and that every interrupt eventually leaves the interrupted set. Progress is defined through the abstraction map, so the theorem says nothing about an internal livelock of the engine.
- *The clock.* On one node the wall clock must not step back between the timer service's due check and the stamping of its command. The theorem speaks of stamps: a host clock that runs ahead can fire a timer early in real time.
- *Publications retried by the client.* A publication has no request ID. A client that sends it again after a lost response has sent a second request, and the message can be published twice. That is outside failure transparency. Exactly-once publication through RPC request IDs is planned.
- *StartFlow retries after the purge.* The first StartFlow and every retry set ignore_already_started_error, and a retry reuses the first request ID. A retry deduplicates only while the run is still in the Flow directory; after the purge it starts a new run. The theorem assumes that every retry's command is

ordered in the committed log before the purge; a retry that reaches the server first can still be ordered after it. The Lean model, which keeps a closed run forever, makes this hold by construction.

- *Liveness backward only without retries.* Every fair DexF trace erases to a fair Dex trace, but the converse holds only for input words without StartFlow retries, because stamped retries move the clock (§8.5).
- *Timer order is idealized.* Both theorems fire timers as a prefix of the due timers. The server can fire a timer while an earlier-due timer, armed by a command applied just before the scan's command, is still armed. We kept the prefix rule, because no distributed system guarantees the order of timers due at almost the same time. Neither theorem covers runs in that window.
- *One owner at a time.* A standby takeover is in scope. Multi-node partition ownership is not: the server runs one node per cluster, and the model has one log writer at a time.

The theorem guarantees only that every extra external call repeats an earlier one. A retry can run a Step method's external calls again, and an older attempt can make a call late, after a newer attempt's call. An application must therefore make each external call safe to repeat and safe to arrive late, for example with an idempotency key that the external system deduplicates on, built from the Flow run, the Step execution, the method, and the effect. The server's deterministic simulation harness with injected crashes checks that a recovered store equals one that never crashed; it is complementary evidence and is not connected to the model.

Fairness is assumed, not provided. Liveness holds for fair runs, and the fairness conditions of Table 6 are assumptions about components outside the rules. F2 assumes that the timer service eventually fires a due timer that nothing else disarms. F3 assumes that the dispatcher eventually starts a pending method that is not canceled first. F5 assumes that the engine eventually commits or drops a returned response. F4 is the one most worth stating plainly: it assumes that every Worker task eventually settles, by returning or by being abandoned after its invocation was canceled. The server does not guarantee this on its own. Cancellation of a running handler is cooperative, and in the failure-free model there is no method timeout to end a handler that never returns. Under failures a timeout fails the attempt, but the handler can keep running as a zombie, so the failure theorem's liveness keeps the assumption for zombies. Its F4 lets any real task be dropped, so it excludes only runs in which some task never returns and is never dropped: a handler that hangs is allowed once its attempt fails and its zombie is dropped. The first theorem's liveness excludes runs in which a task of a live invocation never returns. The safety theorems apply to every run. The fair runs that the development exhibits (§6.6) cover termination, deadlock, and divergence by an infinite input word. A fair run of a program that diverges on its own, on a finite word, is not exhibited, and no lemma says that every reachable configuration has a fair continuation.

A restricted environment. The admissible environment starts one Flow and publishes plain data to external Channels. It makes no RPC, never stops a Flow, never sets an Attribute from outside, and never deletes a message or skips a timer. The specification gives rules for all of these, but they are outside the theorem. In particular an RPC can observe intermediate state and can fail with a lock conflict, and stopping a Flow cancels Steps at arbitrary points; both would need `L` counterparts.

Well-formed programs only. The hypothesis `Program.IsWellFormed` excludes programs that can break a mutex requirement. For such a program the two sides differ: `L` refuses the step, and the Flow takes it. The scope is semantic and undecidable in general. The static lock discipline of §6.6 is sufficient, not necessary. It tracks statically known mutexes only, and it empties the lock set after a call and after a `try`. So it rejects `unlock`, `wait`, `signal`, and `broadcast` on a created mutex, and a program that holds a lock across a call and unlocks after it, although such programs can be well formed. For them well-formedness must be proved by other means.

The timing model. `L` counts stamped actions exactly as the translation counts commits (§3.3). The theorem is exact for this model and silent about any other. A program whose observable behavior depends on the order of two deadlines that are close together can behave differently under another cost model. We

consider such programs incorrect, but the theorem does not exclude them; it just assigns them the Dex timing. The equivalence is also untimed: it does not say that an event happens within some time, only that the same sequences of events are possible. Durations are whole seconds.

Unbounded counters. Step numbers and timer numbers are unbounded in the model and 32-bit in the server. The order of timers with equal fire times was derived from the protobuf encoding of timer IDs by hand; that derivation is an assumption of the model, and it holds for fewer than 2^{32} Step executions of one type per run.

L is a chosen language. L's semantics was designed so that the equivalence is exact: FIFO mutexes and condition variables, cooperative cancellation at four cancellation points, asynchronous emit, a select with one receive branch and one timeout branch, late timers, and a single-use join. Each choice is defensible and close to common practice, but L is not an existing language, and a different reasonable choice (say, a cancellable lock) would need a different translation and a new proof. The theorem shows that Dex can express *a* conventional concurrent language. It does not show that every feature of every such language has a faithful encoding.

Observations. Only inputs, external effects, and the program's result are observable. Reads of a Flow's state by a client are not, and the events of the Flow's internal Channels are projected away. An observer with access to those could tell a translated Flow from its source program.

Trust in the statements. The kernel checks proofs, not intentions. A reader who wants to rely on the first theorem must read the model, the translation, and the statements, about 3,450 lines, against the specification; the failure theorem adds about 1,900 lines of definitions and statements. The development maps every section and every numbered assumption of the specification to its Lean definition to make that reading possible.

10 Related work

Expressiveness and encodings. Felleisen [7] made "expressive power" of programming languages precise through translations that are local in the syntax. The process-calculus community developed the study of encodings furthest. Palamidessi [20] separated the synchronous from the asynchronous π-calculus by showing that no encoding of the first into the second meets a small set of reasonable criteria, which made the choice of criteria a central question, and Gorla [9] proposed a set of criteria that has become a reference point: compositionality, name invariance, operational correspondence, divergence reflection, and success sensitiveness. Peters [21] surveys the criteria and how they relate. Our result is closest in spirit to the semantic criteria. The two step simulations of §6 are an operational correspondence, in a strong form with named action sequences. The ∞ marker of the liveness theorem gives divergence reflection and preservation, and ✓ with the *exit* and *fail* events plays the role of success. We do not claim the structural criteria in Gorla's form. The translation is one Step per program point and is defined from the statement's thread-local semantics, but it is a whole-program compilation in continuation-passing style [3] with naming conventions for Channels, and its target is an engine with a fixed set of primitives, not a calculus with parallel composition. The equivalence itself is trace equivalence, which is at the coarse end of the linear-time/branching-time spectrum [26]; Milner's weak bisimilarity [19] is finer than the claim needs (§5.3).

Workflow patterns. The Workflow Patterns initiative catalogued recurring control-flow behaviors [25], later revised to 43 patterns with formal definitions [23], and the catalog became the standard instrument for comparing workflow languages and systems [28]. YAWL [24] was designed from the patterns, and Kiepuszewski et al. [14] studied the expressive power of classes of workflow languages formally. Pattern-based evaluation and our theorem answer different questions. A pattern rating asks how *directly* a system supports one behavior, and it is checked against the running system. Our theorem asks whether a *class* of programs is expressible at all, with an exact notion of equivalence, and it is checked against a model. The two are complementary, and Dex has both (§1).

Simulation proofs and verified systems. The proof technique is standard. Forward and backward simulations [18] and refinement mappings with stuttering [1] are the foundations; the distinction between safety and liveness that organizes our two theorems is due to Alpern and Schneider [2]. Among mechanized developments, CompCert [17] proves semantic preservation for a compiler by simulation diagrams, and relies on determinism of the target so that preservation in one direction suffices; we use determinism of each Dex rule for a similar economy in §6.3. seL4 [15] proves that a kernel implementation refines its abstract specification. For distributed systems, Verdi [27] verifies implementations under explicit network fault models, and IronFleet [10] combines refinement with mechanized proofs of liveness. Those projects verify an *implementation* against a specification. We verify a property of a *model* of a system, and the gap between that model and the code (§9) is the gap they close. On the engineering side, Woos et al. [29] report that their proof needed 90 invariants and that keeping it up to date as invariants changed was the main difficulty, and they propose to plan for such change. Our gaps of §7.2 are the same phenomenon, and the inductiveness fuzz is our answer, in the tradition of testing a conjecture on random inputs before proving it [5]. The development is in Lean 4 [6].

The source language. L's synchronization constructs go back to monitors. Condition variables with wait and signal are Hoare's [11]; that a woken waiter must re-acquire the lock in competition with other threads is the discipline of Mesa [16], which L follows, with FIFO queues and without spurious wakeups.

Durable execution and workflow engines. The academic literature on durable execution is mostly about the proposition of our second theorem: that a program behaves under failures as it would without them. Sagas [8] introduced long-lived transactions built from steps with compensations. Ramalingam and Vaswani [22] gave a language-based account of fault tolerance through idempotent, logged steps, with a formal calculus that models failures and duplicate requests. Beldi [30] provides exactly-once semantics and transactions for workflows of stateful serverless functions, and Kallas et al. [13] run microservice applications on a serverless platform with formal guarantees that the generated implementations are correct. Jangda et al. [12] give an operational semantics of serverless platforms and characterize when a simpler, naive semantics coincides with it. These works compare a failure-prone execution with an ideal one for the *same* program, as our failure theorem does. Our first theorem compares two *different* programming models in the ideal setting: threads on one side, Steps on the other. We are not aware of a prior machine-checked theorem that a durable workflow model expresses a conventional concurrent language up to trace equivalence. Our failure theorem differs from the works above in three ways. It is about a model written against an existing server, rule by rule, rather than about a calculus or a system designed for the proof. It allows zombie Workers that keep emitting effects after the server gave them up. And it states what repeats: every extra effect equals an earlier one, and the first occurrences of each method's effects come in order. Mechanized frameworks for crash safety of implementations, such as Perennial [4], address the gap between our model and the code.

11 Conclusion

The Dex programming model gives an application Steps that never block, state that is read from a snapshot and written at a commit, Channels, Timers, and a decision at the end of each Step. We have shown that this is enough to express a conventional concurrent language: every well-formed program of L, with its threads, mutexes, condition variables, timed receives, cancellation, and exceptions, translates into a Flow with the same observable finite traces and the same fair maximal traces, in a failure-free environment. We have also shown that, in the same fragment and under explicit hypotheses, erasing the repeated external effects from a finite trace with crashes, takeovers, Worker retries, zombie Workers, and duplicate deliveries gives a failure-free trace, and that every failure-free trace remains possible; for fair runs the erasure keeps the marker, and the reverse inclusion holds for words without StartFlow retries. The Lean kernel checks both proofs from three standard axioms.

The constructive content is the translation. Each live thread that has not halted is carried by one active Step, whose input is the thread's state; a halted thread has none. A mutex is a Channel with one token. A condition variable is a waiter list in an Attribute and one Channel per waiter. A cancellation is a message that competes with the operation's own condition in an AnyOf. A timed operation passes through an Enter

Step and a shared timed Step with one Timer. None of these needs a lock held across a wait or an atomic read-modify-write, and the proof shows why none is missed.

Three lessons go beyond Dex. First, writing an operational semantics against an implementation, rule by rule with citations, is a review of the implementation in its own right; it found two defects before any proof existed. Second, an exact equivalence forces decisions that informal descriptions leave open, about time, about ill-behaved programs, and about what the environment may send, and it is better to make them explicit than to weaken the theorem until they disappear. Third, fixed statements make proofs parallel, but a fixed simulation invariant must be inductive, and that can be tested cheaply before anyone tries to prove it. We lost time to six gaps that a fuzz of perturbed reachable states finds in under a minute.

The failure theorem adds a lesson of its own. Because every attempt of a method reuses one snapshot and runs the same handler, the first occurrences of its effects come in order, even when zombies overlap with newer attempts. That order is what makes erasure work. It also tells an application what to prepare for: every extra external call repeats an earlier one, possibly late, so each call must be safe to repeat and safe to arrive late.

The Dex side is a model validated by review, and liveness assumes fair scheduling and Workers that settle. The failure theorem covers SYNC methods only, assumes that no method exhausts its retries and that user code ignores its attempt, keeps the timer order as an idealization, and has a backward liveness inclusion only for words without StartFlow retries. Extending the admissible environment and the failure model with RPC entry points, SubFlows, and Attribute locks, and connecting the model to the server's deterministic simulation tests, are the next directions.

References

[1] M. Abadi and L. Lamport. The existence of refinement mappings. *Theoretical Computer Science*, 82(2):253–284, 1991. doi: 10.1016/0304-3975(91)90224-P.

[2] B. Alpern and F. B. Schneider. Defining liveness. *Information Processing Letters*, 21(4):181–185, 1985.

[3] A. W. Appel. *Compiling with Continuations*. Cambridge University Press, 1992.

[4] T. Chajed, J. Tassarotti, M. F. Kaashoek, and N. Zeldovich. Verifying concurrent, crash-safe systems with Perennial. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles (SOSP)*, 2019. doi: 10.1145/3341301.3359632.

[5] K. Claessen and J. Hughes. QuickCheck: A lightweight tool for random testing of Haskell programs. In *Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming (ICFP)*, pages 268–279, 2000.

[6] L. de Moura and S. Ullrich. The Lean 4 theorem prover and programming language. In *Automated Deduction – CADE 28*, volume 12699 of *Lecture Notes in Computer Science*, pages 625–635. Springer, 2021. doi: 10.1007/978-3-030-79876-5_37.

[7] M. Felleisen. On the expressive power of programming languages. *Science of Computer Programming*, 17(1–3): 35–75, 1991. doi: 10.1016/0167-6423(91)90036-W.

[8] H. Garcia-Molina and K. Salem. Sagas. In *Proceedings of the 1987 ACM SIGMOD International Conference on Management of Data*, pages 249–259, 1987. doi: 10.1145/38713.38742.

[9] D. Gorla. Towards a unified approach to encodability and separation results for process calculi. *Information and Computation*, 208(9):1031–1053, 2010. doi: 10.1016/j.ic.2010.05.002.

[10] C. Hawblitzel, J. Howell, M. Kapritsos, J. R. Lorch, B. Parno, M. L. Roberts, S. Setty, and B. Zill. IronFleet: Proving practical distributed systems correct. In *Proceedings of the 25th ACM Symposium on Operating Systems Principles (SOSP)*, 2015.

[11] C. A. R. Hoare. Monitors: An operating system structuring concept. *Communications of the ACM*, 17(10): 549–557, 1974. doi: 10.1145/355620.361161.

[12] A. Jangda, D. Pinckney, Y. Brun, and A. Guha. Formal foundations of serverless computing. *Proceedings of the ACM on Programming Languages*, 3(OOPSLA):149:1–149:26, 2019. doi: 10.1145/3360575.

[13] K. Kallas, H. Zhang, R. Alur, S. Angel, and V. Liu. Executing microservice applications on serverless, correctly. *Proceedings of the ACM on Programming Languages*, 7(POPL):367–395, 2023. doi: 10.1145/3571206.

[14] B. Kiepuszewski, A. H. M. ter Hofstede, and W. M. P. van der Aalst. Fundamentals of control flow in workflows. *Acta Informatica*, 39(3):143–209, 2003. doi: 10.1007/s00236-002-0105-4.

[15] G. Klein, K. Elphinstone, G. Heiser, J. Andronick, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, T. Sewell, H. Tuch, and S. Winwood. seL4: Formal verification of an OS kernel. In *Proceedings of the 22nd ACM Symposium on Operating Systems Principles (SOSP)*, pages 207–220, 2009.

[16] B. W. Lampson and D. D. Redell. Experience with processes and monitors in Mesa. *Communications of the ACM*, 23(2):105–117, 1980.

[17] X. Leroy. Formal verification of a realistic compiler. *Communications of the ACM*, 52(7):107–115, 2009. doi: 10.1145/1538788.1538814.

[18] N. Lynch and F. Vaandrager. Forward and backward simulations, part I: Untimed systems. *Information and Computation*, 121(2):214–233, 1995. doi: 10.1006/inco.1995.1134.

[19] R. Milner. *Communication and Concurrency*. Prentice Hall, 1989.

[20] C. Palamidessi. Comparing the expressive power of the synchronous and asynchronous π-calculi. *Mathematical Structures in Computer Science*, 13(5):685–719, 2003.

[21] K. Peters. Comparing process calculi using encodings. In *Proceedings of the Combined Workshop on Expressiveness in Concurrency and Structural Operational Semantics (EXPRESS/SOS 2019)*, volume 300 of *Electronic Proceedings in Theoretical Computer Science*, pages 19–38, 2019. doi: 10.4204/EPTCS.300.2.

[22] G. Ramalingam and K. Vaswani. Fault tolerance via idempotence. In *Proceedings of the 40th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 249–262, 2013. doi: 10.1145/2429069.2429100.

[23] N. Russell, A. H. M. ter Hofstede, W. M. P. van der Aalst, and N. Mulyar. Workflow control-flow patterns: A revised view. BPM Center Report BPM-06-22, BPMcenter.org, 2006.

[24] W. M. P. van der Aalst and A. H. M. ter Hofstede. YAWL: Yet another workflow language. *Information Systems*, 30(4):245–275, 2005. doi: 10.1016/j.is.2004.02.002.

[25] W. M. P. van der Aalst, A. H. M. ter Hofstede, B. Kiepuszewski, and A. P. Barros. Workflow patterns. *Distributed and Parallel Databases*, 14:5–51, 2003.

[26] R. J. van Glabbeek. The linear time – branching time spectrum I: The semantics of concrete, sequential processes. In J. A. Bergstra, A. Ponse, and S. A. Smolka, editors, *Handbook of Process Algebra*, pages 3–99. Elsevier, 2001.

[27] J. R. Wilcox, D. Woos, P. Panchekha, Z. Tatlock, X. Wang, M. D. Ernst, and T. Anderson. Verdi: A framework for implementing and formally verifying distributed systems. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 357–368, 2015.

[28] P. Wohed, N. Russell, A. H. M. ter Hofstede, B. Andersson, and W. M. P. van der Aalst. Patterns-based evaluation of open source BPM systems: The cases of jBPM, OpenWFE, and Enhydra Shark. *Information and Software Technology*, 51(8):1187–1216, 2009. doi: 10.1016/j.infsof.2009.02.002.

[29] D. Woos, J. R. Wilcox, S. Anton, Z. Tatlock, M. D. Ernst, and T. Anderson. Planning for change in a formal verification of the Raft consensus protocol. In *Proceedings of the 5th ACM SIGPLAN Conference on Certified Programs and Proofs (CPP)*, pages 154–165, 2016. doi: 10.1145/2854065.2854081.

[30] H. Zhang, A. Cardoza, P. B. Chen, S. Angel, and V. Liu. Fault-tolerant and transactional stateful serverless workflows. In *Proceedings of the 14th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 1187–1204, 2020.

	Assumption	In Lean
D1	The fragment of the first theorem: one Flow, the admissible environment extended by StartFlow retries; no RPCs, SubFlows, StopFlow, or Attribute locks.	The Dex model, extended.
D2	Every Step method is SYNC: an attempt reaches a Worker only after its start commits. An ASYNC start lost in a crash runs the same attempt again with a new snapshot.	`IsSyncOnly`, true by construction: DexF has only the SYNC start.
H1	No attempt exhausts its retries. Exhaustion closes the Flow as Failed, with WORKER_API_FAIL or INTERNAL in the code, a close that a failure-free run reaches only by deciding ForceFail.	Part of the trace sets; the model closes with the value `"WORKER_API_FAIL"`.
H2	No failure policy (ProceedOnFailure, ProceedToConfiguredStep).	`HasNoFailurePolicy`, true by construction.
H3	User code ignores the attempt-varying context: attempt number, first attempt time, last error, last heartbeat value. Hard-delete Attributes, which are re-read on every attempt, are outside the fragment.	A hypothesis: `IgnoresAttemptContext`.
H4	No non-retryable failure.	A case of H1.
D6	On one node, the wall clock does not step back between the timer service's due check and the stamping of its command.	The premise of the timer rule.
D7	The input word is the *outcome word*: requests in the order of their final outcomes. A re-sent publication is a second request.	The admissible outcome words.
HP	Every StartFlow retry is ordered in the log before the purge of the run.	True by construction: the model keeps a closed run.
TID	Every logged command of a started attempt (result, returned attempt, heartbeat, failure after start) carries a non-empty task ID.	True by construction: task IDs are minted numbers.
F0	Every request of the outcome word gets its final outcome, in order.	`RequestsArrive`, part of `IsFairMaximalTrace`.
F1–F6	The fairness of Table 6, restated for DexF: clocks grow without bound (F1); a timer of either kind that stays armed and due is fired (F2); an invocation that stays startable while the server is up starts (F3); every real task, in Θ or Z, eventually returns or is dropped (F4); every returned task eventually leaves by a Complete rule or a stale result (F5); F6 is trivial in the fragment.	`DexF.IsFair`.
FF, FR, FI	For liveness: finitely many crashes, lost attempts, and returned attempts; the server comes back after every crash; every interrupt eventually leaves I, by its commit or by a crash that empties I, so after the last recovery each one commits.	Part of `DexF.IsFair`.
OD1	A timer command fires a non-empty prefix of the due timers in the order (fire time, encoded timer ID), as in the first theorem.	The timer rule of the Dex model for wait timers. Lean does not compute timer ID bytes: at equal fire times a wait timer comes before a retry timer, and retry timers keep their list order.

Table 12: What the failure theorem assumes. H1 to H4 exclude four features that change observable behavior beyond repeated effects.

Package	Content	Statements
WP-F1	A normal step maps to its Dex counterpart.	`normalStep_simulated`
WP-F2	A failure step is a stutter under α.	`failureStep_isStutter`
WP-F3	First occurrences map to the virtual task, repeats to stutters; the ghost h and the trace agree.	`workerStep_simulated`, `effectHistory_of_exec`
WP-F4	The invariants are preserved.	`invariant_step`
WP-F5	Safety, both inclusions.	`safety_failure_forward`, `freeFiniteTrace_ isFailureFiniteTrace`, …
WP-F6	The progress rule and liveness.	`isProgress_eq_dexProgress`, `liveness_failure`
WP-F7	The end-to-end corollary.	`endToEnd_safety`, `endToEnd_liveness`

Table 13: The work packages of the failure proof.

Part of `Failure/`	Files	Lines	Content
definitions	6	2,268	configurations, rules, repetition, α, invariants, trace sets
`Statements/`	2	391	the leaf and main statements
`Proofs/`	49	18,940	WP-F1 to WP-F4 and the leaf part of WP-F6
`Composite/`	17	8,305	WP-F5 to WP-F7
`Examples/`	9	3,449	kernel-checked runs, witnesses, counterexamples, review probes, the fuzz
tooling	5	575	decidable equality, checkers, aggregators
Total	88	33,928	

Table 14: The failure development at commit `dd449bd57`, counted with `wc -l`. `FailureFuzz.lean`, the fuzz entry point, adds 59 lines outside `Failure/`.