Research Square

Preprints are preliminary reports that have not undergone peer review.
They should not be considered conclusive, used to inform clinical practice,
or referenced by the media as validated information.

Mechanizing Asynchronous Distributed Algorithms in Rocq: The Echo Algorithm as a Case Study

Illia Ilin

fate98765@gmail.com

V. N. Karazin Kharkiv National University

Grygoriy Zholtkevych

Lviv University

Research Article

Keywords: Distributed algorithms, Formal verification, Rocq, Labeled transition systems, Echo algorithm, Inductive invariants

Posted Date: October 6th, 2026

DOI: https://doi.org/10.21203/rs.3.rs-11040750/v1

License: This work is licensed under a Creative Commons Attribution 4.0 International License.
Read Full License

Additional Declarations: No competing interests reported.

Mechanizing Asynchronous Distributed Algorithms in Rocq: The Echo Algorithm as a Case Study

Illia Ilin[1*] and Grygoriy Zholtkevych[1,2]

[1*]V.N. Karazin Kharkiv National University, Kharkiv, Ukraine.
[2]Ivan Franko National University of Lviv, Lviv, Ukraine.
ORCID: 0009-0001-2774-251X (Illia Ilin)
ORCID: 0000-0002-7515-2143 (Grygoriy Zholtkevych)

*Corresponding author(s). E-mail(s): fate98765@gmail.com;
Contributing authors: grygoriy.zholtkevych@lnu.edu.ua;

Abstract

This paper studies a reusable way to mechanize asynchronous message-passing algorithms in Rocq, using the Echo algorithm as the first complete case study. A network state consists of a total map of local process states and a bag of in-flight packets. A labeled transition system represents nondeterministic message delivery, and reachability supports proofs by inductive invariants. Echo is easy to state locally: forward a token, adopt the sender of the first token as parent, and return an echo after all expected replies arrive. Its global proof must nevertheless cover every delivery order, including states with crossing tokens and partly completed return waves. We prove a decision-time correctness theorem: whenever the initiator has decided in a reachable state, every listed node has a finite parent chain to the initiator, and every parent edge belongs to the communication graph. The proof separates structural and parent-chain invariants from token/echo accounting. A central lemma shows that a decided initiator cannot coexist with a token whose destination is still idle. A graph-depth certificate and well-founded induction then carry this result across the network. The theorem is conditional on decision and assumes no delivery fairness. The reusable part of the development is deliberately small; algorithm states, handlers, and accounting invariants remain Echo-specific. The artifact contains 142 declarations closed by `Qed` and no admitted goals.

Keywords: Distributed algorithms, Formal verification, Rocq, Labeled transition systems, Echo algorithm, Inductive invariants

1 Introduction

Describing a distributed algorithm is often much easier than proving it correct. A process reads one message, updates a small local state, and sends a finite set of messages. An external scheduler chooses which message to deliver, so two runs of the same program on the same graph can pass through different intermediate states. This is one consequence of having no useful global clock in an asynchronous system [1, 2]. In Rocq, the consequence is concrete: every invariant must survive every handler branch for every packet that the scheduler may choose.

This paper examines a direct method for mechanizing algorithms in this setting. Global configurations combine a total map of process states with a bag of in-flight packets, while an LTS exposes each scheduler choice as a transition. Correctness is then reduced to invariants over reachable configurations. We use Echo as the first complete case study because its local handler is small but its proof still has to connect local counters, in-flight messages, and a global completion claim. We use *reusable method* in this limited sense; the current artifact is not a plug-in verification framework for arbitrary distributed algorithms. Whether the same organization transfers to other algorithms remains to be evaluated through additional case studies.

The Echo algorithm is a compact example of the underlying proof problem. An initiator sends a token wave through a connected undirected graph. A node adopts the sender of its first token as its parent, forwards tokens to its remaining neighbors, and eventually returns an echo. The initiator decides after receiving all expected echoes. The algorithm is well known [3, 4], but the usual informal proof leaves out states that matter to a proof assistant. Tokens can cross on an edge, a node can receive a second token after choosing a parent, and a token can remain in flight while echoes are already travelling towards the root.

Echo has also served as a formal-verification example. Kindler et al. [5] present a Petri-net-based verification method using the algorithm, while the TLA+ Examples repository contains a PlusCal/TLA+ specification and a TLC model [6]. Our result has a different scope: it is a proven Rocq theorem over an abstract node type and any finite listed graph satisfying explicit hypotheses. It is not bounded to the graph instances selected for model checking. We claim neither a new Echo algorithm nor a proof of eventual termination.

Our main result is conditional. If the initiator is in the decided phase in a reachable global state, then every listed node has a finite parent-pointer path to the initiator. Parent pointers lie on graph edges, the initiator has no parent, and non-initiators cannot decide. Since each node stores at most one parent, these properties yield a rooted spanning structure over the listed nodes.

We build on the model introduced by Barskyi, Ilin, and Zozulia [7]. That chapter proposed a common mathematical basis for simulation-based dynamic analysis and formal static analysis of distributed computations. It defined global configurations in terms of local process states and a communication store, but it contained neither a Rocq development nor a correctness proof for a distributed algorithm. Here we carry out that formal verification task. We retain its global state and message-delivery semantics, while fixing choices that the chapter left open: handler steps are atomic, the message bag is represented as a list, and reachability is defined explicitly.

The paper makes the following contributions.

- We define a small reusable LTS layer for asynchronous algorithms in Rocq. It separates algorithm-specific states and handlers from generic reachability, traces, and invariant induction, and it exposes scheduler nondeterminism without implementing a scheduler.
- We instantiate this method with an executable Echo specification over an abstract finite node type and give a parametric, kernel-checked proof of decision-time spanning-tree correctness.
- We identify the proof structure needed by this case study: structural and causal invariants are separated from token/echo accounting, then joined by a pending-chain argument. The theorem takes a connectivity certificate as a parameter and proves correctness upon decision without assuming eventual delivery.

Section 2 recalls the transition-system model and fixes the Echo semantics. Section 3 describes the Rocq representation. Sections 4 and 5 state the correctness criterion and explain the proof architecture. Section 6 presents the main results and their scope. The mechanization and its trusted base appear in Section 7. Sections 8 and 9 discuss related work and limitations.

2 Model and Algorithm

2.1 Labeled transition systems

A labeled transition system (LTS) is a tuple

$$\mathcal{T} = (\mathcal{S}, \mathcal{L}, I, \longrightarrow),$$

where $\mathcal{S}$ is a state type, $\mathcal{L}$ is a label type, $I \subseteq \mathcal{S}$ is the set of initial states, and $\longrightarrow \subseteq \mathcal{S} \times \mathcal{L} \times \mathcal{S}$ is a transition relation. We write $s \xrightarrow{\ell} s'$ for $(s, \ell, s') \in \longrightarrow$. This is the standard formulation [8]. We use it instead of the richer input/output (I/O) automata interface [9] because the proof needs neither action signatures nor composition theorems.

Reachability is the least predicate containing every initial state and closed under transitions:

$$\frac{s \in I}{\mathit{reachable}(s)} \qquad \frac{\mathit{reachable}(s) \quad s \xrightarrow{\ell} s'}{\mathit{reachable}(s')}.$$

This inductive definition is the only point at which the safety proofs refer to execution histories. A predicate $P : \mathcal{S} \to \mathsf{Prop}$ is an invariant if it holds in every reachable state. The standard induction rule reduces this claim to an initial-state proof and a one-step preservation proof.

The global state of a message-passing instance is a pair

$$G = (\sigma, B), \qquad \sigma : \mathcal{N} \to \mathit{LocalState}, \qquad B \in \mathit{list}(\mathit{Packet}).$$

The function σ is a total map from node identifiers to local states. We use the list B as a finite bag. Its order has no scheduling meaning, and the transition relation

may select any member. A packet records a source, destination, and body. Delivery removes one matching occurrence from B, runs the destination's handler, updates that destination in σ, and appends the packets produced by the handler.

The representation follows the configurations and communication system of the earlier model [7], with one change to the semantics. That presentation treats process actions, sends, and receives separately. In our Rocq instance, receipt of a packet, the resulting local update, and all sends made by the handler form one atomic LTS transition. A reachable state therefore cannot contain half of a handler action, such as an updated pending counter without the echo emitted by the same handler. Atomicity is an assumption of the model, not a property proved about an implementation.

2.2 The Echo protocol

Let $\mathcal{N}$ be a finite set of nodes, $r \in \mathcal{N}$ the initiator, and $adj : \mathcal{N} \times \mathcal{N} \to \mathsf{bool}$ an adjacency test for an undirected graph without self-loops. There are two message bodies:

$$\mathit{Token} \qquad \text{and} \qquad \mathit{Echo}.$$

The local state of a node is a quadruple

$$(\mathit{phase}, \mathit{parent}, \mathit{pending}, \mathit{children}),$$

where $\mathit{phase} \in \{\mathit{Idle}, \mathit{Active}, \mathit{Decided}\}$, *parent* is an optional node, *pending* is a natural number, and *children* is a list of nodes from which echoes have been received.

Initially, every node is idle, has no parent, has pending count zero, and has an empty children list. The packet bag is also empty. The initiator then performs the start action: it becomes active, sets its pending count to its degree, and sends a token to every neighbor.

The delivery handler has six phase/message combinations. Two perform the work of the algorithm.

1. When an idle node n receives a token from m, it records $\mathit{parent}(n) = m$ and becomes active. It sends tokens to all neighbors except m. Its pending count is the number of those neighbors. If this number is zero, n sends an echo to m immediately.
2. When an active node receives an echo, it decrements its pending count and records the sender as a child. If the old count was one, the node has received its last expected echo. A non-initiator sends an echo to its parent; the parentless initiator enters the decided phase.

An active node that receives another token keeps its parent and answers the sender with an echo. This reply is essential on non-tree edges. A decided node ignores tokens, while idle and decided nodes ignore echoes. These branches make the transition function total, and each must still be shown to preserve the invariants.

The model is asynchronous. It assumes neither first-in, first-out (FIFO) channels nor a bound on delay. It does not model crashes, packet loss, or packet creation by the network. A process may still receive what looks like a duplicate token: two neighbors can forward distinct tokens to the same node before either receives a reply.

3 Reusable LTS Infrastructure and Echo Formalization

The development targets Rocq 9.0 [10] and uses only the standard library. It separates a generic semantic layer from the Echo instance, but it does not yet package the boundary as a Rocq module type. Four files contain the main development: `LTS.v`, `EchoAlgorithm.v`, `EchoCorrectness.v`, and `Example.v`. A fifth file, `Network.v`, contains auxiliary generic network definitions but is not a dependency of the Echo proof.

3.1 The LTS layer

The generic LTS is a dependent record whose state and label fields determine the types of the remaining fields.

Listing 1 Core LTS and reachability definitions.

```
Record LTS : Type := mkLTS {
  lts_state : Type;
  lts_label : Type;
  lts_init  : lts_state -> Prop;
  lts_trans : lts_state -> lts_label -> lts_state -> Prop
}.

Inductive reachable (M : LTS) : lts_state M -> Prop :=
| reach_init : forall s,
    lts_init M s -> reachable M s
| reach_step : forall s lbl s',
    reachable M s ->
    lts_trans M s lbl s' ->
    reachable M s'.
```

The library also defines finite traces, trace validity, product composition, and an invariant induction rule. The Echo proof needs only reachability and invariant induction; its executions therefore use the general LTS definition.

3.2 Nodes, local states, and packets

The node type is abstract. The executable update and packet-removal functions compare identifiers, so they require decidable equality. The list `all_nodes` represents the finite network. The main theorem ranges over this list rather than requiring a finite-type class.

Listing 2 Local and global Echo state.

```
Inductive phase := Idle | Active | Decided.

Record proc_state := mkProc {
  ps_phase    : phase;
  ps_parent   : option node;
  ps_pending  : nat;
```

```
  ps_children : list node
}.

Record echo_packet := mkPkt {
  ep_src  : node;
  ep_dst  : node;
  ep_body : echo_msg
}.

Record echo_state := mkEchoState {
  es_procs : node -> proc_state;
  es_msgs  : list echo_packet
}.
```

A single-node update is defined as a point update:

$$update_proc(f, n, s)(m) = \begin{cases} s, & n = m, \\ f(m), & n \neq m. \end{cases} \tag{1}$$

The proof file keeps this definition locally opaque and works through two lemmas, `upd_self` and `upd_other`. Unfolding the equality test inside a large record expression would spread conditionals throughout the goal. These lemmas expose only the two useful cases: the updated node and an unchanged node.

The packet bag is a list. The function `remove_pkt` deletes the first packet whose source, destination, and body match the delivered packet. The lemma `remove_pkt_in` states that every remaining packet was already present. Packets have no unique identifier, so the model cannot distinguish equal packet records, although one delivery removes only one occurrence. The algorithm and its bag invariants do not require a stronger notion of packet identity.

3.3 Transition relation

There are two labels: `ELStart` and `ELDeliver pkt`. The global transition relation has one constructor for each label.

Listing 3 Global Echo transition relation.

```
Inductive echo_step : echo_state -> echo_label ->
                      echo_state -> Prop :=
| step_start : forall gs,
    ps_phase (es_procs gs initiator) = Idle ->
    echo_step gs ELStart (initiator_start gs)
| step_deliver : forall gs pkt gs',
    In pkt (es_msgs gs) ->
    gs' = handle_msg (ep_dst pkt)
            (mkEchoState (es_procs gs)
              (remove_pkt pkt (es_msgs gs))) pkt ->
    echo_step gs (ELDeliver pkt) gs'.
```

Table 1 Correspondence between the mathematical model and Rocq.

Mathematical object	Rocq object	Representation choice
State space $\mathcal{S}$	`echo_state`	Total process map and packet list
Initial predicate I	`echo_init`	All local states idle; empty bag
Action label ℓ	`echo_label`	Startup or delivery of a concrete packet
Transition relation	`echo_step`	Inductive relation with two constructors
Local transition	`handle_msg`	Total executable function
Communication store	`es_msgs`	List treated as a bag
Process-state update	`update_proc`	Pure point update on a total function
Execution prefix	`reachable`	Inductive closure from an initial state

The condition `In pkt (es_msgs gs)` is the scheduler interface. There is no packet-selection function; a transition may deliver any packet in the bag. A theorem about all `reachable` states therefore covers every finite delivery order allowed by the model.

The start constructor is enabled whenever the initiator is idle, so at first glance it seems to allow repeated starts. The invariant `init_idle_empty` rules this out in reachable states. An idle initiator has an empty bag; after the first start it is active, and no handler makes it idle again. Startup uniqueness follows from the transition system instead of being built into the rule.

3.4 Correspondence with the mathematical model

Table 1 matches the mathematical presentation to its Rocq objects. This correspondence stops at the formal specification; it does not connect that specification to network code running on separate machines.

4 Correctness Specification

We consider states in which the initiator has decided. Rather than treating *the spanning tree is complete* as a single claim, we separate the three properties it requires.

First, every parent pointer follows an edge:

$$\forall n, p. \quad parent_G(n) = \mathsf{Some}(p) \implies adj(n, p) = \mathsf{true}. \tag{2}$$

Second, the initiator has no parent, and no node points to itself. Third, every node reaches the initiator by repeatedly following parent pointers.

We encode the last property with a fuel-bounded function. For a global state G, node n, and $k \in \mathbb{N}$, `parent_path G n k` follows exactly k parent links when all required pointers exist. At zero it returns n. At a successor, it fails if the current node has no parent and otherwise recurses from that parent. A node reaches the initiator if the following equation holds for some fuel value:

Listing 4 Parent-chain correctness predicate.

```
Fixpoint parent_path (gs : EState) (n : node) (k : nat)
    : option node :=
  match k with
  | 0 => Some n
  | S k' =>
      match ps_parent (proc_of gs n) with
      | None => None
      | Some par => parent_path gs par k'
      end
  end.

Definition reaches_initiator (gs : EState) (n : node) : Prop :=
  exists k, parent_path gs n k = Some initiator.
```

This bounded function avoids a separate transitive-closure relation and makes it easier to show that parent paths survive a state update. The fuel need not equal graph depth; it only witnesses that the chain is finite.

Definition 4.1 (Decision-time spanning correctness) An Echo state G is decision-time correct if, whenever the initiator is decided, every node in `all_nodes` reaches the initiator, every parent pointer follows an adjacency edge and targets a node in `all_nodes`, the initiator has no parent, and no non-initiator is decided.

This is a safety property: it rules out an incorrect decision. It does not cover a run in which the scheduler leaves a packet undelivered forever and the initiator never decides. The conditional statement therefore needs no fairness assumption.

5 Proof Architecture

The proof is much longer than the algorithm because it must relate a local counter to packets and states across the graph. We use three layers: structural facts, parent-chain facts, and token/echo accounting needed to show that a decision excludes idle nodes. Figure 1 summarizes the dependencies.

5.1 Invariant induction and local updates

The generic rule `invariant_by_induction` has the following type:

$$\big(\forall s.\ I(s) \Rightarrow P(s)\big) \Rightarrow \big(\forall s, \ell, s'.\ P(s) \Rightarrow s \xrightarrow{\ell} s' \Rightarrow P(s')\big) \Rightarrow \forall s.\ \mathit{reachable}(s) \Rightarrow P(s).$$

Each application has a base case, a startup case, and a delivery case split by message body and receiver phase.

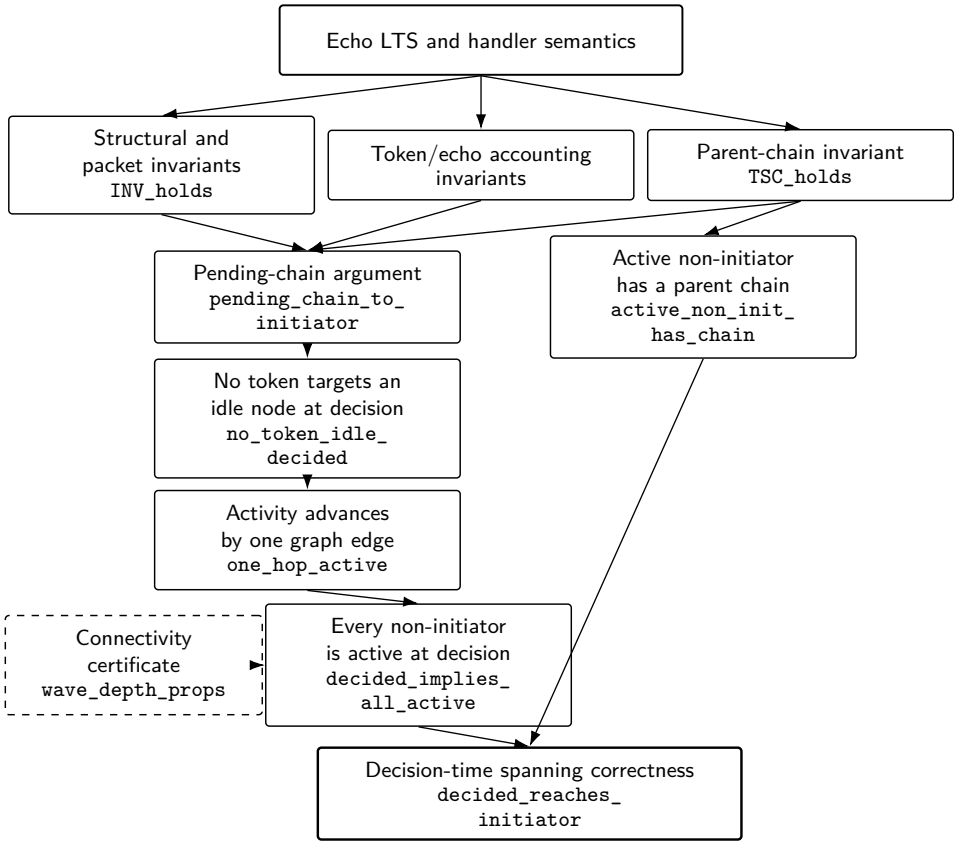


Fig. 1 Main dependencies in the decision-time correctness proof. Solid arrows show use in a later argument; the dashed box is the connectivity certificate supplied as a theorem parameter, and the heavier outline marks the final result. Local preservation lemmas are omitted.

Nearly every branch uses the point-update equations:

$$\begin{aligned} upd_self: &\quad update_proc(f,n,s)(n) = s, \\ upd_other: &\quad n \neq m \Rightarrow update_proc(f,n,s)(m) = f(m). \end{aligned}$$

For example, suppose an idle node `self` receives a token from `sender`. To preserve Eq. (2), at `self`, `upd_self` exposes the new parent `Some sender`; packet validity gives $adj(sender, self)$, and symmetry gives the required edge in the other direction. At any other node, `upd_other` reduces the goal to the pre-state invariant. The same self/other split appears throughout the preservation proof.

5.2 The structural invariant

The combined invariant `INV` has four clauses:

$$\begin{aligned} INV(G) \equiv\ & tree_invariant(G) \wedge \\ & children_are_neighbors(G) \wedge \\ & valid_packets(G) \wedge \\ & init_idle_empty(G). \end{aligned}$$

The tree clause itself combines parent adjacency, absence of self-parenting, and the initiator's lack of a parent. Packet validity states that every packet in the bag travels along an adjacency edge. The final clause states that an idle initiator implies an empty bag.

We prove these predicates together because their preservation arguments depend on one another. The only difficult branch is token delivery to an idle node. Before unfolding the new state, we establish five facts:

1. the incoming packet follows the edge from sender to receiver, by `valid_packets`;
2. the reverse edge exists, by symmetry;
3. sender and receiver differ because the graph has no self-loops;
4. the receiver is not the initiator, since an idle initiator would imply an empty bag while the incoming packet is in that bag;
5. every forwarded token follows an edge because its destination was selected from the receiver's neighbor list.

With these facts in place, the leaf and non-leaf branches reduce to point-update equations and list membership. The other handler cases either leave the process map unchanged or change fields that do not affect parent validity.

This gives:

$$\forall G. \quad reachable(G) \Rightarrow INV(G). \tag{3}$$

Alongside the structural layer, we prove two smaller lifecycle invariants. An active non-initiator always has a parent, and a non-initiator is never decided. The second uses the first: the handler's decision branch requires an active node with no parent, which is impossible for a non-initiator. Separately, `parent_in_all_nodes_holds` proves that every parent pointer targets a node in the finite list covered by the theorem.

5.3 Parent chains

Local parent validity does not imply that a parent chain reaches the initiator. A set of valid edges could still contain a cycle. The invariant `TSC` gives the missing parent-chain information:

$$\begin{aligned} TSC(G) \equiv\ & idle_no_parent(G) \wedge \\ & token_src_has_chain(G) \wedge \\ & active_non_init_has_chain(G). \end{aligned}$$

The key fact about tokens is this: a token can only be sent by the initiator at startup or by a node that has just adopted a parent. In both cases its source already has a chain to the initiator. When an idle receiver adopts that source as its parent, the source chain is extended by one edge. Once a parent is set it never changes, so later deliveries preserve the chain.

The preservation proof has to show that parent paths survive state updates. Two lemmas handle this step. `parent_path_upd_nil` preserves a successful path when the updated node previously had no parent; such a node cannot occur inside an old successful path. `parent_path_upd_agree` preserves paths when all parent fields agree between the two states. With these lemmas, the handler branches do not have to repeat an induction over the fuel of `parent_path`.

We use the following consequence later:

$$\begin{aligned} \mathit{reachable}(G) \land n \in \mathit{all_nodes} \land \mathit{phase}_G(n) = \mathit{Active} \land n \neq r \quad &\Longrightarrow \\ \mathit{reaches_initiator}(G, n). & \end{aligned} \tag{4}$$

5.4 Why decision excludes an idle token destination

The informal claim that *the wave has reached every node before the root decides* is not directly inductive. A token can still be in flight to an idle node after activity has spread elsewhere. We therefore prove:

$$\begin{aligned} \mathit{reachable}(G) \land \mathit{phase}_G(r) = \mathit{Decided} \land \mathit{Token}(m, n) \in B_G \quad &\Longrightarrow \\ \mathit{phase}_G(n) \neq \mathit{Idle}. & \end{aligned} \tag{5}$$

Suppose, for contradiction, that `Token(m,n)` remains in the bag and n is idle. Packet validity shows that m and n are neighbors. Since n is idle, it has not sent an echo and cannot occur in m's children list. The accounting invariant then shows $\mathit{pending}_G(m) \geq 1$.

If m is not the initiator, let $p = \mathit{parent}_G(m)$. A positive pending count at m means that m has not completed its own return step, so m cannot yet occur among p's completed children. The same accounting argument gives $\mathit{pending}_G(p) \geq 1$. We repeat this reasoning along the parent chain, which `TSC` proves is finite and ends at the initiator. The same inequality then holds at the root: $\mathit{pending}_G(r) \geq 1$.

The decision branch sets the initiator's pending count to zero, and the invariant `decided_pending_zero` preserves that equality. This contradicts $\mathit{pending}_G(r) \geq 1$. In Rocq, we carry the argument along the chain by induction on the fuel witnessing `reaches_initiator`; the local step is the lemma `pending_propagates`, and the full argument is `pending_chain_to_initiator`.

The accounting behind this argument is the largest part of the proof. It relates pending counters to forwarded-neighbor lists, children lists, and Echo packets. Because these clauses depend on one another in the last-echo branch, we group them in the combined invariant `token_echo_accounting`.

5.5 From one edge to the whole graph

Three further invariants support a one-edge propagation lemma:

- `token_src_not_idle`: the source of an in-flight token is not idle;
- `token_sent_or_notidle`: for an active node m and suitable neighbor n, either n is already non-idle or a token from m to n is in flight;
- `parent_is_active`: a recorded parent is active or decided.

Together with Eq. (5), these invariants prove `one_hop_active`. Suppose the initiator has decided, m is non-idle, and n is a neighbor farther from the root. If n is the parent of m, `parent_is_active` shows that n is not idle. Otherwise, `token_sent_or_notidle` says that n is already non-idle or that a token from m to n is in flight. Equation (5) rules out an idle n in the second case. Since a non-initiator cannot decide, n must be active.

To apply this local fact across the graph, the theorem takes a function $d : \mathcal{N} \to \mathbb{N}$ that satisfies

$$d(r) = 0 \quad \wedge \quad \forall n \in \mathcal{N}.\ n \neq r \Rightarrow \exists m \in \mathcal{N}.\ adj(n, m) \wedge d(m) < d(n). \tag{6}$$

This function is a connectivity certificate. On any finite connected graph, shortest-path distance from r provides one. Conversely, Eq. (6) rules out a disconnected component: repeatedly choosing a neighbor of smaller depth must end at the only possible node of depth zero, the initiator.

We then apply Rocq's well-founded induction to the inverse image of $<$ under d. It shows that every non-initiator is active when the initiator has decided. Equation (4) then gives the main theorem.

6 Mechanized Results

We state the final result in a form close to its Rocq type. Let *nodes* be a duplicate-free finite list containing r; let adjacency be symmetric and have no self-loops; and let d satisfy Eq. (6) for every node in the list.

Theorem 6.1 (Decision implies complete parent chains) *For every global state G, if G is reachable in the Echo LTS and the initiator r is decided in G, then*

$$\forall n \in nodes.\quad \exists k.\ parent_path(G, n, k) = \mathsf{Some}(r).$$

The corresponding Rocq theorem is `decided_reaches_initiator`. Its final proof is short because two earlier theorems do most of the work: `decided_implies_all_active` and `active_non_init_has_chain`. The initiator itself has a direct zero-length path; every other node is active by well-founded induction and therefore has a chain by `TSC`.

Theorem 6.2 (Structural safety) *In every reachable state, parent pointers follow communication edges, no node is its own parent, every parent target belongs to `all_nodes`, the*

Table 2 Rocq artifact summary.

File	`Qed`	Role
`LTS.v`	2	LTS, reachability, traces, invariant induction
`Network.v`	0	Auxiliary generic network definitions
`EchoAlgorithm.v`	0	Executable state and transition definitions
`EchoCorrectness.v`	127	Invariants and correctness theorems
`Example.v`	13	Concrete three-node execution
Total	142	No `Admitted` or `admit`

initiator has no parent, children entries are neighbors, and every packet in flight follows a communication edge.

This theorem combines `INV_holds` with `parent_in_all_nodes_holds`. Together, Theorems 6.1 and 6.2 justify the spanning-tree interpretation. A listed non-root node has a parent because a successful path from that node cannot have length zero; `ps_parent` stores at most one parent; each parent is listed and adjacent; and every parent chain reaches the root. A directed cycle of parent pointers would contradict the last fact.

Theorem 6.3 (Only the initiator decides) *In every reachable state, a node distinct from the initiator is not in the `Decided` phase.*

Theorems 6.1–6.3 cover all finite prefixes and every message order. They do not establish that every initial state leads to a decided state. Such a liveness theorem requires a fairness condition: a packet that remains in the bag must eventually be selected. No such condition occurs in the theorem type.

The graph-depth assumption is a proof parameter rather than runtime state. It expresses connectivity in a form suited to well-founded induction. The algorithm never reads d, so choosing a different certificate cannot change an execution.

7 Mechanization and Reproducibility

7.1 Size and proof status

Table 2 reports the source checked at commit `19cf5f879424`; the full identifier appears under Code availability.

The sources contain 142 occurrences of `Qed`. In Rocq 9.0, running `Print Assumptions decided_reaches_initiator` reports `Closed under the global context`, so the theorem introduces no global axioms. Its section parameters, including graph hypotheses and the depth certificate, remain explicit arguments in the final theorem type.

When the Rocq tools are on `PATH`, the clean build command is:

Table 3 Explicit parameters of the main theorem.

Parameter or hypothesis	Purpose
`node`	Abstract type of node identifiers
`node_eq`	Decidable equality used by executable updates and packet removal
`initiator`	Chosen root of the Echo wave
`all_nodes`	Finite node list covered by the conclusion
`adj`	Boolean test for communication edges
`adj_sym`	The modeled graph is undirected
`adj_irrefl`	The modeled graph has no self-loops
`initiator_in_nodes`	The root appears in the finite node list
`nodup_nodes`	The node list contains no duplicate identifiers
`wave_depth`	Natural-number connectivity certificate; not read by the algorithm
`wave_depth_props`	Root depth is zero; every non-root node has an adjacent node of smaller depth

Listing 5 Rebuilding the development.

```
make clean
make -j2
```

The portable `Makefile` also accepts optional `COQBIN` and `ROCQLIB` directory arguments for installations that are not on `PATH`. The checked commit can be rebuilt locally with Rocq 9.0 using the commands above.

7.2 Parameters and hypotheses

Table 3 separates computational parameters from graph hypotheses. The theorem makes no eventual-delivery assumption. Eventual decision would require such an assumption, but the checked conditional theorem neither declares nor uses one.

The trusted computing base consists of Rocq's kernel, the imported standard library modules, and the correspondence between the mathematical algorithm and its Gallina definition. Rocq can evaluate the handler, but we do not extract it or connect it to production networking code. The theorem therefore certifies the formal model, not an external implementation.

7.3 Concrete execution

`Example.v` runs the algorithm on the line graph $0 - 1 - 2$, with node 0 as initiator. It constructs five states: startup, delivery of $\mathit{Token}(0, 1)$, delivery of $\mathit{Token}(1, 2)$, delivery of $\mathit{Echo}(2, 1)$, and delivery of $\mathit{Echo}(1, 0)$. Computation proves that node 0 decides, the packet bag is empty, node 1 has parent 0, node 2 has parent 1, and the children lists match the line. This example is not evidence for the universal theorem. Its purpose is to make the executable semantics easy to inspect and to catch mismatches between the intended handler and its definition.

Table 4 Closest formal treatments of the Echo algorithm.

Work	Formalism	Scope reported by the artifact or paper
Kindler et al. [5]	Petri nets and temporal logic	Echo as the example for a Petri-net-based description and verification method
Merz [6]	PlusCal/TLA+ and TLC	Executable specification; type and ancestor invariants checked on one complete three-node model; a termination formula is stated in the source
This work	Rocq 9.0 and an LTS	Kernel-checked decision-time safety for every finite listed graph satisfying the stated hypotheses; no eventual-decision theorem

8 Related Work

Labeled transition systems and I/O automata are standard models of concurrency and distributed algorithms [2, 8, 9]. We use the smaller LTS interface because this is a closed-system invariant proof. Interfaces and compositional reasoning fall outside this model.

The closest work is specific to Echo. Kindler, Reisig, Völzer, and Walter [5] use the algorithm to present a verification method based on Petri nets and temporal logic. The official TLA+ example by Merz [6] specifies Echo in PlusCal/TLA+. Its source defines termination and parent/ancestor properties; the bundled TLC configuration checks `TypeOK` and `AncestorProperties` on a complete three-node graph. Table 4 makes the comparison explicit. We do not claim the first formal model of Echo. Our contribution is a Rocq proof in which graph size and node type remain parameters, together with the invariants that connect local pending counters to the global decision condition. The result proves size-parametric safety, not eventual termination.

Several mechanized frameworks address executable distributed systems. Verdi uses Coq network semantics and verified system transformers to transfer correctness results from idealized networks to richer fault models. Its case studies include Raft and replicated services [11]. We did not use Verdi because this work begins with a different model [7]: a single explicit global configuration and message-delivery transition are used for both simulator traces and deductive analysis. Verdi's implementation and fault-model machinery supports questions beyond our decision-time invariant and would shift attention away from that link. This is a difference in scope, not a limitation of Verdi.

Disel combines protocols, dependent types, and a Hoare-style logic to verify implementations and their composition in Coq [12]. Aneris provides a higher-order separation logic for modular node-local reasoning about distributed programs [13]. IronFleet combines state-machine refinement and program verification to prove safety and liveness properties of practical implementations [14]. These systems address implementation refinement, composition, faults, or program logic, none of which we cover here. Our narrower aim is to study a direct LTS-based method for invariant proofs of asynchronous algorithms, with Echo as the present case study.

The Randomised Distributed Algorithms (RDA) library models randomized distributed algorithms in Coq with both set and distributional semantics [15]. Echo has no randomness; its nondeterminism comes from delivery order. We can therefore use an inductive transition relation without a probabilistic semantics.

Mechanized work on message-passing theory also appears in the Journal of Automated Reasoning. Tirore, Bengtson, and Carbone prove soundness and completeness of a computable projection for global types, with all results mechanized in Coq [16]. They study a type-level projection algorithm rather than a distributed graph traversal. Their work also shows why it is useful to say which definitions are executable and how recursion is represented in the mechanization. Automation such as CoqHammer can reduce local proof-search effort [17]. We rely on explicit invariants and standard tactics because the difficult step here is finding the right intermediate properties, not closing local proof goals.

Finally, our starting point is the mathematical model of Barskyi, Ilin, and Zozulia [7], which supports both dynamic and static analysis. We do not repeat its survey of cloud and simulation concerns. We use the model, fix the atomic-step semantics, and provide the machine-checked correctness argument left as future work in that chapter.

9 Discussion and Limitations

9.1 What the proof says about Echo

The theorem rules out premature decision under any finite message reordering. It says more than a check of selected schedules, but less than a total correctness result. A scheduler can stop with a packet still in the bag. Such a run never reaches a decided state and does not contradict Theorem 6.1. Proving liveness would require infinite runs, a weak fairness assumption, and a proof that every continuously enabled delivery eventually occurs. Fair delivery alone would still not handle a one-node graph. The startup handler leaves an isolated initiator active with pending count zero and emits no packet; the execution then has no next step and never decides. A total-correctness development must change the startup rule for this case or assume that the initiator has a neighbor.

The pending-chain argument is easy to miss in a textbook description. In the code, a decision depends only on the initiator’s counter. The proof must relate that counter to an idle node that may be several edges away. The parent chain provides a path back to the root, while the accounting invariant turns each missing echo along that path into a positive counter. Parent adjacency and graph connectivity alone are therefore not enough.

9.2 Generality

The LTS record, reachability relation, trace definitions, and invariant rule do not depend on Echo. Point-update and bag-removal lemmas are also common when an asynchronous protocol is represented by functional maps and packet lists. The Echo-specific accounting invariants cannot be reused unchanged. Their organization nevertheless suggests a pattern for other wave algorithms: separate parent-chain facts

from numeric facts about unfinished work, then connect the two in the completion proof.

The resulting method has three parts: define the global configuration and nondeterministic delivery relation, prove local preservation lemmas for each handler branch, and combine structural, causal, and accounting invariants to derive the algorithm's global safety property. Only the first part is currently captured by a generic Rocq record. Local state, packet types, handlers, and the main correctness theorem are defined directly for Echo. The development should therefore be read as evidence from one complete instantiation, not as a validated framework in which a new algorithm can already be plugged into a module interface. A second case study would be needed to test which proof components transfer without redesign.

The node-identifier type and graph size remain parameters. The proof applies to any finite undirected graph satisfying the listed assumptions. Each concrete graph must currently supply a depth certificate. A more complete library could define paths and connectedness once, derive shortest-path depth for finite graphs, and prove `wave_depth_props` automatically.

9.3 Model boundary

The proof has four main limits.

1. The topology is fixed, finite, undirected, and free of self-loops. Node joins, failures, and edge changes are not transitions of the LTS.
2. The network does not lose, duplicate, or create packets. Delivery can be delayed and reordered without bound. Duplicate-looking tokens arise from the algorithm's graph traversal, not from network duplication.
3. Handler execution and its outgoing sends are atomic. A concrete implementation would require a refinement argument covering serialization, crashes between local update and send, and the behavior of the transport.
4. The formal development and the simulation model share mathematical structure, but no proved refinement currently connects simulator traces to Rocq traces. Such a refinement is needed before dynamic experiments and static theorems can be said to concern the same executable artifact.

These limits point to several next steps: fair-run liveness, reusable finite-graph connectivity results, a second algorithm case study with an explicit module boundary, and a trace-preserving link to the simulator.

10 Conclusion

We presented a direct method for mechanizing asynchronous distributed algorithms in Rocq and evaluated it through a complete Echo case study. The generic LTS layer provides reachability and invariant induction; the algorithm instance supplies its states, messages, handler, and proof-specific invariants. For Echo, if the initiator has decided, the proof gives a finite parent path from every listed node to the root. This holds for every reachable state and every finite delivery order allowed by the model. The key point, often left implicit in textbook accounts, is that an outstanding token to an idle

node forces a positive pending count along the parent chain. That contradicts decision at the initiator. Structural invariants and graph-depth induction complete the argument. The present evidence comes from one algorithm; turning the method into a reusable library requires another instantiation and an explicit module boundary.

This result supplies the static-verification component proposed in the Springer chapter by Barskyi, Ilin, and Zozulia [7]. Future work is to connect fair infinite LTS executions, simulator traces, and a distributed implementation through formal refinement results.

Statements and Declarations

Funding. No funding was received for conducting this study or preparing this manuscript.

Competing interests. The authors have no relevant financial or non-financial interests to disclose.

Author contributions. Illia Ilin developed the Rocq formalization, performed the mechanized proof, and prepared the manuscript. Grygoriy Zholtkevych contributed to the formal model and proof design, supervised the work, and reviewed the manuscript. Both authors read and approved the final manuscript.

Data availability. No research datasets were generated or analyzed. The software artifact supporting the formal results is identified under Code availability.

Code availability. The Rocq sources, build files, and worked example are available at https://github.com/Ilyin-Ilya/echo-algorithm-coq. The results reported in this article correspond to merge commit `19cf5f879424b19d16c571491161b2aa08553001`.

Use of generative AI. Anthropic Claude Code was used to assist with Rocq proof development, proof refactoring, and artifact documentation. OpenAI Codex was used to assist with manuscript drafting, language revision, organization, checks against journal guidelines, and preparation of the TikZ source for the proof-dependency diagram. The authors determined and approved the formal model, theorem statements, technical arguments, interpretation, and conclusions. They reviewed and revised all AI-assisted material, rebuilt the complete Rocq development, and checked the technical claims and citations against the Rocq sources and cited literature. AI-generated output was not treated as evidence. The authors take full responsibility for the accuracy, originality, and integrity of the manuscript, formalization, and diagram.

References

[1] Lamport, L.: Time, clocks, and the ordering of events in a distributed system. Communications of the ACM **21**(7), 558–565 (1978) https://doi.org/10.1145/359545.359563

[2] Lynch, N.A.: Distributed Algorithms. Morgan Kaufmann, San Francisco (1996)

[3] Chang, E.J.H.: Echo algorithms: Depth parallel operations on general graphs. IEEE Transactions on Software Engineering **SE-8**(4), 391–401 (1982) https://doi.org/10.1109/TSE.1982.235573

[4] Tel, G.: Introduction to Distributed Algorithms, 2nd edn. Cambridge University Press, Cambridge (2000)

[5] Kindler, E., Reisig, W., Völzer, H., Walter, R.: Petri net based verification of distributed algorithms: An example. Formal Aspects of Computing **9**(4), 409–424 (1997) https://doi.org/10.1007/BF01211299

[6] Merz, S.: The Echo Algorithm. TLA+ Examples repository. PlusCal/TLA+ specification and TLC model; accessed 15 July 2026 (2020). https://github.com/tlaplus/Examples/tree/master/specifications/echo

[7] Barskyi, O., Ilin, I., Zozulia, A.: Modeling technique of distributed computations. In: Ermolayev, V., Potapov, I., Ignatenko, O., Hornung, R., Hlybovets, A., Yakovyna, V., Prytula, Y., Spivakovsky, O. (eds.) Information and Communication Technologies in Education, Research, and Industrial Applications. Communications in Computer and Information Science, vol. 2359, pp. 258–272. Springer, Cham (2025). https://doi.org/10.1007/978-3-031-81372-6_19

[8] Milner, R.: A Calculus of Communicating Systems. Lecture Notes in Computer Science, vol. 92. Springer, Berlin (1980). https://doi.org/10.1007/3-540-10235-3

[9] Lynch, N.A., Tuttle, M.R.: An introduction to input/output automata. CWI Quarterly **2**(3), 219–246 (1989)

[10] The Rocq Development Team: The Rocq Prover Reference Manual, Version 9.0. https://rocq-prover.org/doc/V9.0.0/refman/. Accessed 15 July 2026 (2025)

[11] Wilcox, J.R., Woos, D., Panchekha, P., Tatlock, Z., Wang, X., Ernst, M.D., Anderson, T.: Verdi: A framework for implementing and formally verifying distributed systems. In: Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation, pp. 357–368. Association for Computing Machinery, New York, NY, USA (2015). https://doi.org/10.1145/2737924.2737958

[12] Sergey, I., Wilcox, J.R., Tatlock, Z.: Programming and proving with distributed protocols. Proceedings of the ACM on Programming Languages **2**(POPL), 1–30 (2018) https://doi.org/10.1145/3158116

[13] Krogh-Jespersen, M., Timany, A., Ohlenbusch, M.E., Gregersen, S.O., Birkedal, L.: Aneris: A mechanised logic for modular reasoning about distributed systems. In: Programming Languages and Systems. Lecture Notes in Computer Science, vol. 12075, pp. 336–365. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-44914-8_13

[14] Hawblitzel, C., Howell, J., Kapritsos, M., Lorch, J.R., Parno, B., Roberts, M.L., Setty, S.T.V., Zill, B.: Ironfleet: Proving practical distributed systems correct. In: Proceedings of the 25th Symposium on Operating Systems Principles, pp. 1–17. Association for Computing Machinery, New York, NY, USA (2015). https://doi.org/10.1145/2815400.2815428

[15] Fontaine, A., Zemmari, A.: RDA: A Coq library to reason about randomised distributed algorithms in the message passing model. Scientific Annals of Computer Science **26**(2), 157–186 (2016) https://doi.org/10.7561/SACS.2016.2.157

[16] Tirore, D., Bengtson, J., Carbone, M.: A sound and complete projection for global types. Journal of Automated Reasoning **69**(14) (2025) https://doi.org/10.1007/s10817-025-09726-9

[17] Czajka, Ł., Kaliszyk, C.: Hammer for Coq: Automation for dependent type theory. Journal of Automated Reasoning **61**, 423–453 (2018) https://doi.org/10.1007/s10817-018-9458-4