Using Lean in Pattern-Oriented Diagnostics

Concept by Dmitry Vostokov
DumpAnalysis.org

Using Lean in Pattern-Oriented Diagnostics

Dmitry Vostokov | DumpAnalysis.org
Developed with AI assistance (GPT-6 Astra and Claude Fable 5.1)

Version 8 | 30 October 2026

Throughout this article, “we” and “our” refer to Dmitry Vostokov as the authorial plural. The acknowledgment of AI assistance does not attribute the cited prior work to the AI systems.

Introduction

When we recognize an analysis pattern in a memory dump or a software trace, we also make a statement about the collected software execution artifacts. For example, several waiting relationships form a Wait Chain, a collection of ownership and waiting relationships suggests Deadlock, or the absence of an expected message directs further analysis. Such statements have different strengths. Some describe recorded data. Others reconstruct software behavior or explain a reported problem.

In this article, we explore Lean as a language for making selected diagnostic statements precise and checking their derivations. The starting point is our pattern-oriented approach to software diagnostics and the associated memory dump and trace analysis pattern languages [1, 2]. We propose adding formal companions to selected patterns: definitions, explicit assumptions, examples, and proofs that can be checked again by another diagnostician.

Lean is both a programming language and an interactive theorem prover [3]. It can express a diagnostic model, represent small collections of evidence, and check proofs of statements about them. However, a proof about a representation does not by itself establish that the representation faithfully describes the captured system. We therefore keep the evidence boundary visible throughout the examples.

The practical part consists of a short introduction and three reproducible experiments: a Wait Chain certificate, a restricted Deadlock model with a timeout counterexample, and a Missing Message inference with an explicit coverage condition. The accompanying project uses Lean 4.19.0 and its bundled standard library. It requires no Mathlib download, debugger, production dump, or external service. All positive examples were checked, and four deliberately incorrect attempts were rejected. The examples are small constructed cases for learning the method; they do not constitute a production diagnostic engine.

1 From pattern recognition to a checked statement

Pattern-oriented analysis gives names to recurrent structures and behaviors. A name allows us to recognize a familiar configuration, compare it with other cases, and communicate an analysis. Formalization adds questions about the configuration itself. Which objects participate? Which relationships are required? Which observations support them? Which interpretation follows, and under what conditions?

Consider three threads. If thread 11 waits for thread 22, and thread 22 waits for thread 33, there is a dependency path from thread 11 to thread 33. This is a structural conclusion about the represented relationships. Whether thread 33 is stuck, still working, or waiting for an external response is a separate question. A path alone does not settle it.

Similarly, a cycle in a wait graph is not automatically a proof of permanent lack of progress. The semantics of the waits matter. A timeout, cancellation, or external release may permit an escape. A formal model is useful here because it forces us to say which transitions are possible. It also makes omissions easier to discuss: if cancellation is absent from the model, a proof cannot tell us whether cancellation exists in the actual implementation.

We distinguish four objects in a formal companion to a diagnostic pattern.

Object	Diagnostic purpose
Evidence representation	Records the observations being analyzed, with an identified artifact and capture scope
Pattern predicate	States the relationships required for a particular pattern occurrence
Interpretation theorem	States what follows from the pattern and the specified behavioral assumptions
Case certificate	Supplies the witnesses or proof for one particular case

The interpretation theorem can be reused. The case certificate belongs to a particular evidence set. A change in a dump, trace, parser, model, or assumption may require a new certificate. This is compatible with iterative analysis: new evidence can change the applicable premises without invalidating a correctly proved conditional theorem.

2 A short introduction to Lean

Lean represents propositions as types and proofs as terms of those types. A theorem declaration gives the statement and its proof. Tactics help construct proof terms, and the kernel checks the resulting terms. A tactic is therefore part of proof construction; its apparent success is not a substitute for checking the resulting theorem and its dependencies [3, 4].

For this article, only a small part of Lean is necessary. A definition introduces data or a function. A structure groups fields. An inductive declaration introduces a data type or a relation generated by named constructors. For example, a path relation can be generated by a single edge and by extending a shorter path. The same language can describe both a diagnostic object and the ways of establishing a statement about it [4, 5].

Lean notation	Meaning in the examples
`Nat`, `Bool`, `List`	Natural numbers, Boolean values, and finite lists
`Prop`	The universe of propositions
`def`, `theorem`	A definition and a proved statement
`structure`, `inductive`	A record type and a type or relation defined by constructors
`h : P`	A parameter named `h` that supplies a proof of proposition `P`
`P → Q`, `P ∧ Q`, `¬ P`	Implication, conjunction, and negation
`x ∈ xs`, `x ∉ xs`	List membership and its negation
`by`, `exact`, `intro`	Start a tactic proof, supply a proof, and introduce an assumption
`constructor`, `cases`	Build a constructor-based result and analyze possible constructors
`rfl`, `decide`, `unfold`	Use definitional equality, a decision procedure evaluated by reduction, and a definition's body
`#eval`, `#print axioms`	Evaluate an expression and inspect a theorem's axiom dependencies

The distinction between `Bool` and `Prop` is useful. A Boolean result is data produced by a computation. A proposition is a statement for which a proof may be supplied. In the examples, `#eval` displays simple computations, while named theorems retain checked statements. The `decide` tactic handles the small finite membership and equality questions used here. It is not a universal procedure for discovering diagnostic explanations [4, 5].

Lean also distinguishes a missing proof from a proof of negation. If a proof attempt fails, the proposition may be false, the chosen tactic may be insufficient, or the required assumptions may be missing. We do not translate every failed proof into a negative diagnosis. Where an example has a counterexample, we construct it explicitly.

3 Preparing the reproducible project

Install Lean using the official instructions, which recommend Visual Studio Code and the Lean 4 extension [6]. Complete the extension setup and open a new terminal. The extension's setup guide installs Elan, which manages toolchains; each toolchain includes Lake for working with projects. The companion is also available in our LeanPOD repository: https://github.com/DumpAnalysis/LeanPOD. Open the cloned or downloaded repository, or the archive's lean-pattern-diagnostics directory, in VS Code. Run all commands below from the project directory.

The project deliberately pins Lean 4.19.0 [7]. This is a reproducibility choice, not a claim that it is the latest release. The lean-toolchain file contains the following single line:

```
leanprover/lean4:v4.19.0
```

Install that toolchain, check the selected version, and build the examples:

```
elan toolchain install leanprover/lean4:v4.19.0
lake env lean --version
lake build
```

The version command should identify Lean 4.19.0. A successful build ends with Build completed successfully. Some informational output includes the values produced by #eval and the axiom reports. Subsequent builds may reuse completed work. To replay a particular file's output, use lake env lean Intro.lean, replacing the filename when necessary.

The archive contains four example source files, the umbrella module Diagnostics.lean, project configuration, four negative files, a Python verification script, and the captured verification output. It also includes README.txt, VALIDATION.txt, SHA256SUMS.txt, and the compatibility shim source.

The Python script is optional. The Lean commands work without Python. To run all checks together, use python3 verify.py on macOS or Linux, or py -3 verify.py on Windows with Python installed. The expected final line is All 10 checks passed. The negative files are intentionally excluded from the normal build. Run lake build successfully before trying the negative files: they import the compiled positive modules.

All listings below are editable text. Their complete files are also in the archive and repository. When copying from Word, preserve indentation and save the source as UTF-8. The supplied files avoid copying problems with mathematical symbols. In VS Code, the Lean extension also supports entering symbols with backslash abbreviations, such as \to for the implication arrow.

We retain import Std consistently in the listings, although these examples need only the implicitly imported Init prelude and also check with that line removed.

A first checked observation

Save the following as Intro.lean, or open the supplied file.

```
import Std

namespace Intro

def waiters : List Nat := [11, 22, 33]

#eval waiters.length

theorem three_waiters : waiters.length = 3 := by
  decide
```

```
theorem member_of_evidence (xs : List Nat) (n : Nat)
    (h : n ∈ xs) : n ∈ xs := by
  exact h

#print axioms three_waiters

end Intro
```

Running `lake env lean Intro.lean` prints `3` and reports that `Intro.three_waiters` does not depend on any axioms. Here, `decide` proves a concrete statement about the length of a list. The second theorem illustrates a premise: `exact h` returns the supplied membership proof. It establishes no new membership fact independently of that premise. This small distinction becomes important when diagnostic assumptions are introduced later.

After a successful build, run `lake env lean negative/FalseCount.lean`. That file asks Lean to prove that the same list has length four using `decide`. Lean reports that the proposition is false and returns a nonzero exit status. A failed check is expected for this file.

4 Example 1 Checking a Wait Chain

We start with a minimal representation. An edge is a pair of thread identifiers. Its direction is from a waiting thread to the thread on which it depends. Resource identities, wait modes, process identities, and capture timestamps are omitted in this first experiment. In a real case, those details would be needed to justify the edge extraction and to disambiguate identifiers.

The following part of `WaitChain.lean` defines reachability by one or more edges. The constructor `one` accepts one recorded edge. The constructor `more` places an edge before an already established path. There is no constructor for a path of zero edges; consequently, reachability from a thread to itself requires an actual closed walk in this representation.

```
import Std

namespace WaitChain

abbrev Edges := List (Nat × Nat)

inductive Reaches (es : Edges) : Nat → Nat → Prop where
  | one {a b} : (a, b) ∈ es → Reaches es a b
  | more {a b c} : (a, b) ∈ es →
      Reaches es b c → Reaches es a c

def TwoStep (es : Edges) (a b c : Nat) : Prop :=
  (a, b) ∈ es ∧ (b, c) ∈ es

theorem twoStep_reaches (es : Edges) (a b c : Nat)
    (h : TwoStep es a b c) : Reaches es a c := by
  exact Reaches.more h.1 (Reaches.one h.2)
```

`TwoStep` names a particularly small certificate: two edges sharing an intermediate thread. The theorem `twoStep_reaches` converts that certificate into a reachability proof. The projections `h.1` and `h.2` select the two parts of the conjunction. This is a reusable rule, independent of the specific thread identifiers.

Append the rest of the file:

```
def captured : Edges := [(11, 22), (22, 33)]

#eval captured.contains (11, 22)
#eval captured.contains (22, 44)

theorem chain_11_33 : Reaches captured 11 33 := by
  exact twoStep_reaches captured 11 22 33 (by
    unfold TwoStep
    decide)

theorem wrong_witness : ¬ TwoStep captured 11 22 44 := by
  unfold TwoStep
  decide
```

```
def cyclic : Edges := [(11, 22), (22, 33), (33, 11)]

theorem cycle_11 : Reaches cyclic 11 11 := by
  exact Reaches.more (b := 22) (by decide)
    (Reaches.more (b := 33) (by decide)
      (Reaches.one (by decide)))

#print axioms chain_11_33
#print axioms wrong_witness
#print axioms cycle_11

end WaitChain
```

The two evaluations print `true` and `false`. The captured data contains the edge from 11 to 22, but no edge from 22 to 44. The theorem `chain_11_33` establishes the two-edge path. The theorem `wrong_witness` proves that the proposed two-edge certificate through 22 to 44 is invalid. It is specifically a statement about that certificate; its wording does not claim a general theorem about every possible path to 44.

The second evidence set adds the edge from 33 to 11. The theorem `cycle_11` constructs a closed path with three edges. The explicit intermediate identifiers in the proof tell Lean which witnesses to use. Lean checks each membership claim against the supplied list.

Run `lake env lean WaitChain.lean`. The three axiom reports contain `propext` and `Quot.sound`, standard axioms of Lean's logic, and no unfinished-proof axiom. These dependencies arise from library proofs used by the membership decision procedure, including instances for the element type. They do not mean that the diagnostic edges were introduced as axioms. In these examples, the edges are concrete data and their memberships are checked [8].

For a deliberate failure, run `lake env lean negative/FalseChain.lean`. It attempts to certify the nonexistent edge to 44 and is rejected. As a further experiment, make a copy of the positive file and remove the edge from 33 to 11 from `cyclic`. The existing cycle certificate can no longer be checked. This is a useful property of a case certificate: it is attached to the evidence it uses.

In the removed-edge experiment, Lean reports an error and returns exit status 1. Its error recovery still allows the later `#print axioms cycle_11` command to run, now exposing `sorryAx`. That output describes an unfinished proof; it does not make the failed check successful.

The result is a certified structural relationship in a small representation. There is still no theorem saying that these threads are deadlocked. To make that claim, we need to specify what the participating threads can do.

5 Example 2 Deadlock and an explicit escape

For the next experiment, we use two threads and two locks. Thread T1 owns A and requests B. Thread T2 owns B and requests A. A pending thread can finish only when its requested lock is free. A finish transition abstracts acquisition, completion, and release into one step. An optional timeout allows T1 to abort and release A. The timeout parameter means that this abort transition is available; the model does not measure elapsed time.

This model is deliberately restricted. There are no third-party releases, recursive acquisitions, additional lock instances, or external cancellation mechanisms. The two ownership flags describe the locks held by their designated threads. They are not a general lock manager. These restrictions are part of the interpretation of the model. `State` admits all 16 Boolean combinations; the transition constructors do not enforce this ownership interpretation. The proofs below concern the named states. The state also does not distinguish completed from abandoned work: after `abort1`, T1's operation is no longer pending.

The first part of Deadlock.lean describes the state and all allowed transitions:

```
import Std

namespace Deadlock

structure State where
  aHeld : Bool
  bHeld : Bool
  pending1 : Bool
  pending2 : Bool
  deriving Repr, DecidableEq

-- T1 owns A and requests B. T2 owns B and requests A.
-- A finish step acquires the free requested lock,
-- completes the operation, and releases both locks.
inductive Step (timeouts : Bool) : State → State → Prop where
  | finish1 {s} : s.pending1 = true → s.bHeld = false →
      Step timeouts s {s with aHeld := false, pending1 := false}
  | finish2 {s} : s.pending2 = true → s.aHeld = false →
      Step timeouts s {s with bHeld := false, pending2 := false}
  | abort1 {s} : timeouts = true → s.pending1 = true →
      Step timeouts s {s with aHeld := false, pending1 := false}
```

The transition relation is generated only by these constructors. We define a stuck state as one with unfinished work and no allowed next step. Requiring unfinished work excludes normal termination from this predicate.

```
def Stuck (timeouts : Bool) (s : State) : Prop :=
  (s.pending1 = true ∨ s.pending2 = true) ∧
  ∀ next, ¬ Step timeouts s next

def locked : State := ⟨true, true, true, true⟩

theorem locked_stuck : Stuck false locked := by
  constructor
  · exact Or.inl rfl
  · intro next h
    cases h with
    | finish1 hp hb => cases hb
    | finish2 hp ha => cases ha
    | abort1 ht hp => cases ht
```

The proof first establishes that an operation is pending. It then considers an arbitrary supposed next step. Neither finish constructor can apply because the requested lock is held. The abort constructor cannot apply because timeouts are disabled. Each case therefore contains an impossible Boolean equality. The proof eliminates all three cases.

For the named state locked with timeouts disabled, we use Stuck as the formal reading of Deadlock: each pending thread holds the lock requested by the other, and no transition provides an escape. This interpretation concerns the demonstrated configuration; Stuck is not a characterization of circular wait for every record representable by State.

For example, the record ⟨false, true, true, false⟩ is stuck with timeouts disabled: T1 is pending and holds no lock, B is held, and T2 has no pending work. None of the three transitions is enabled, yet there is no circular wait.

Now keep the same locked state but enable the timeout transition:

```
def escaped : State := ⟨false, true, false, true⟩

theorem timeout_escape : Step true locked escaped := by
  exact Step.abort1 rfl rfl

theorem timeout_not_stuck : ¬ Stuck true locked := by
  intro h
  exact h.2 escaped timeout_escape
```

```
def finished : State := ⟨false, false, false, false⟩

theorem second_finishes : Step true escaped finished := by
  exact Step.finish2 rfl rfl

#print axioms locked_stuck
#print axioms timeout_escape
#print axioms timeout_not_stuck
#print axioms second_finishes

end Deadlock
```

timeout_escape establishes an allowed abort by T1. timeout_not_stuck uses that transition to refute the stuck-state predicate in the extended model. second_finishes then gives T2 a completion step after A has been released. These are proofs of possible transitions. They do not prove that a scheduler will eventually select them. A guarantee of eventual progress would require a temporal model and appropriate scheduling assumptions.

Run lake env lean Deadlock.lean. All four displayed theorems report that they do not depend on any axioms. Then run lake env lean negative/FalseDeadlock.lean. It tries to reuse locked_stuck as a proof for the timeout-enabled model. Lean rejects the attempt because the theorem types do not match.

The diagnostic point is that the same instantaneous ownership configuration supports different conclusions under different transition models. An available timeout invalidates this particular deadlock conclusion even before the timeout occurs. Conversely, if the real system has a possible escape omitted from the model, the no-transition proof applies only to the restricted model.

The graph from the previous example and the state machine here are separate teaching models. No automatic translation between them has been proved. A production implementation would need a theorem connecting a resource-aware wait graph to its execution semantics, together with a justification for reconstructing that graph from debugger observations.

6 Example 3 Missing Message and capture coverage

Missing Message can direct an investigation by identifying an expected message that is absent from a trace [2]. We want to make explicit the additional condition needed to move from a statement about the trace to a statement about execution.

Suppose a request starts within an observation window, but its completion message is absent. The request may still be pending, or it may have completed without a retained message. Logging may have been disabled, a buffer may have overflowed, a filter may have removed the message, or the capture may have ended before the relevant message became observable. A negative statement about execution needs a coverage argument.

In this example, actual is an ideal event list for the window, and observed is the captured event list for the same window. Request identifiers are assumed to distinguish the relevant operations in that window. A finish event in the ideal list means that the operation completed there. We use list membership only; we do not yet formalize event ordering, latency, or repeated instances with the same identifier.

The first part of MissingMessage.lean defines the event type and a request-specific coverage condition:

```
import Std

namespace MissingMessage

inductive Event where
  | start (request : Nat)
  | finish (request : Nat)
  deriving Repr, DecidableEq

-- Both traces describe the same fixed observation window.
def CompleteFor (actual observed : List Event) (r : Nat) : Prop :=
  Event.finish r ∈ actual → Event.finish r ∈ observed
```

```
theorem absence_with_coverage (actual observed : List Event)
    (r : Nat) (coverage : CompleteFor actual observed r)
    (absent : Event.finish r ∉ observed) :
    Event.finish r ∉ actual := by
  intro occurred
  exact absent (coverage occurred)
```

CompleteFor says that if the relevant finish event occurred in the ideal history, the corresponding finish event is present in the captured history. It is weaker than requiring every event to be captured. The theorem then proves that, under this condition, absence from the captured list implies absence from the ideal list. The proof assumes occurrence and derives a contradiction with the recorded absence.

Append the concrete observation and its conditional interpretation:

```
def captured : List Event := [.start 7, .start 8, .finish 8]

theorem missing_7 : Event.finish 7 ∉ captured := by
  decide

theorem conditional_case (actual : List Event)
    (coverage : CompleteFor actual captured 7) :
    Event.finish 7 ∉ actual := by
  exact absence_with_coverage actual captured 7 coverage missing_7
```

The absence from captured is proved directly. The statement about actual remains conditional on coverage. Lean does not obtain this coverage premise by inspecting the observed list. In practice, its justification would have to account for instrumentation, the relevant event semantics, capture boundaries, and retention. A model of an instrumented program and its capture mechanism could make part of that justification formal. Merely writing the premise as a custom axiom would leave the empirical question unanswered.

The following counterexample shows why the premise cannot simply be omitted:

```
def actualLossy : List Event := [.start 7, .finish 7]
def observedLossy : List Event := [.start 7]

theorem lossy_counterexample :
    Event.finish 7 ∉ observedLossy ∧
    Event.finish 7 ∈ actualLossy := by
  decide

theorem lossy_not_complete :
    ¬ CompleteFor actualLossy observedLossy 7 := by
  intro coverage
  have recorded := coverage (by decide)
  have absent : Event.finish 7 ∉ observedLossy := by decide
  exact absent recorded

#print axioms absence_with_coverage
#print axioms conditional_case
#print axioms lossy_counterexample
#print axioms lossy_not_complete


end MissingMessage
```

The ideal history contains a completion, while the captured history does not. Lean proves both facts together and proves that the coverage condition fails for these lists. This is stronger than reporting an unsuccessful attempt to prove non-occurrence: it supplies a concrete case in which the desired unconditional inference would be wrong.

Run lake env lean MissingMessage.lean. The general absence theorem has no axiom dependencies. The displayed instantiated and counterexample theorems depend only on propext. The difference from the Wait Chain reports comes from the library instances used for the respective element types. Run lake env lean negative/FalseAbsence.lean to see Lean reject a claim that the finish event is absent from actualLossy.

The conclusion remains bounded by the observation window. Even adequate coverage does not establish that the request never completes later. It also does not explain why it failed to complete within the window. That explanation needs further evidence and possibly other patterns. A conditional absence result is therefore one element of a diagnostic narrative.

7 Reproducing and interpreting the checks

The verification script runs ten checks: the toolchain version, the project build, each of the four positive files, and each of the four deliberately rejected files. It checks expected output as well as exit status. In particular, an unavailable import or a missing executable does not count as the expected rejection of a false diagnostic statement. The captured command output is included as `verification-output.txt`.

File	Expected result
`Intro.lean`	Prints 3 and checks the list-length theorem
`WaitChain.lean`	Prints true and false; checks a path, a rejected witness, and a cycle
`Deadlock.lean`	Checks the restricted deadlock, timeout escape, and subsequent completion
`MissingMessage.lean`	Checks conditional absence and the explicit lossy-capture counterexample
`negative/FalseCount.lean`	Rejected because the proposed count is false
`negative/FalseChain.lean`	Rejected because a required edge is missing
`negative/FalseDeadlock.lean`	Rejected because the no-timeout theorem has the wrong type for the new model
`negative/FalseAbsence.lean`	Rejected because the ideal history contains the event

A diagnostic theorem should be reviewed together with its definition and hypotheses. In these examples, the axiom reports contain only the stated standard logical dependencies. There is no `sorry`, `admit`, custom diagnostic axiom, or `native_decide` in the positive source. `#eval` is used only for displaying example computations. The proof results do not rely on treating that printed output as a proof.

The command `#print axioms` is useful, but it is not a complete semantic review. An implication with an impossible premise may be valid and diagnostically useless. A predicate can also have a misleading name. We therefore inspect what the statement actually says and whether its premises correspond to the investigation. Lean's validation guidance makes the same distinction between proof validity and the intended meaning of a statement [8].

The original checks were performed on 15 October 2026 on Linux with the official Lean 4.19.0 binary, commit `6caaee842e94`. The execution environment needed a small compatibility shim for locating the process's own executable. It redirected only that path lookup; the Lean binary, proof kernel, standard library, and example sources were unmodified. The archive records this detail and includes the shim source for transparency. A normal Lean installation should not need it.

8 Connecting Lean to diagnostic tools

The examples can be developed into a workflow in which a debugger extracts observations, a search procedure proposes pattern instances, and a Lean component checks selected claims. Prolog is a natural candidate for the relationship-search component. An LLM can assist with candidate formalizations, proof construction, and explanation. These are proposed roles; the supplied project implements the small formal examples, not an integrated WinDbg or Prolog extension.

For a first integration, we would export a bounded, explicit evidence set. A wait record should retain the waiting thread, resource, owner, wait mode, capture identity, and extraction location. A trace record should retain its

event identity, correlation information, original message location, and window. A source digest helps identify the artifact, but does not prove that the extraction is correct. Identifiers must also be interpreted in their process and capture context.

The search component could return a candidate chain or cycle. Lean would validate the witness against the supplied evidence and apply an appropriate interpretation theorem. This separates the cost of finding a candidate from the responsibility for checking it. It follows the established idea of certifying algorithms: an output can be accompanied by a witness whose checking is simpler than repeating or trusting the producing algorithm [9].

We would keep the following distinctions in the resulting report.

Report entry	Information retained
Observed facts	Source locations, capture scope, and extraction details
Checked pattern	Definition version and the validated witness
Conditional interpretation	The theorem and all outstanding premises
Refutation	A checked counterexample or a proof of negation
Unresolved claim	Missing evidence, missing premises, or an unfinished proof attempt

Pattern cooperation can then be represented by theorems that compose compatible results. Compatibility includes agreement about object identity, time interval, and model. A cycle reconstructed from one capture and a timeout configuration read from another cannot be combined silently. A theorem about composition should expose the required correspondence.

For LLM-assisted analysis, we would also keep the diagnostic specification under review. An LLM could produce a valid proof after weakening the intended statement or strengthening its premises. A checked term would then support a different claim. The diagnostician remains responsible for the meaning of the formalization and the evidence-to-model correspondence. Formal checking makes selected inferential steps inspectable; it does not turn every generated explanation into an established cause.

9 Related work and novelty

The use of logic in diagnosis has a substantial history. Reiter's theory of diagnosis from first principles relates a system description, observations, and assumptions about abnormal components. It also permits competing diagnoses [10]. We therefore do not claim that formal diagnostic reasoning begins with Lean or with this proposal. The present examples are also not an implementation of Reiter's diagnosis algorithm.

Proof-carrying code associates code with a formal justification of a specified safety property [11]. Certifying algorithms attach witnesses to particular outputs [9]. Lean provides an existing language, proof infrastructure, and checking kernel [3]. These are prior foundations for the proposed workflow. The elementary reachability construction, transition analysis, and absence argument in this article are standard forms of reasoning.

The contribution we propose is a particular organization of these ideas around diagnostic analysis patterns and software execution artifacts. A pattern instance can be accompanied by a formal statement, its evidence witness, the assumptions needed for interpretation, and a checkable derivation. This can extend a pattern catalogue with formal companions while retaining the vocabulary used by practicing diagnosticians.

Three aspects are worth developing further. First, the formal companion should distinguish an observed structure from its behavioral interpretation. Second, a negative conclusion should carry the coverage conditions that make it applicable to execution. Third, pattern cooperation should preserve the scope and provenance of the evidence being combined. The examples demonstrate small parts of these requirements; a complete evidence and report schema remains future work.

We use the expression proof-carrying diagnostic report descriptively for this proposed application of existing proof-carrying and certification ideas. We do not claim priority for the general expression or for the use of proof assistants in diagnostics. The references establish relevant foundations and context, but do not constitute an exhaustive literature or patent search. The defensible novelty claim at this stage is an application and methodological synthesis, supported by a small reproducible Lean demonstration, rather than a new logic or a new theorem-proving technique.

10 Further development

The next useful step is to formalize one real pattern variant with its acquisition assumptions. For Wait Chain, this could mean one synchronization mechanism, one debugger extraction format, and one well-defined capture mode. It would then be possible to examine both the recognizer and the evidence transformation, including malformed records and inconsistent observations.

For trace analysis, a natural extension is to distinguish several reasons why a message is absent. A missing event, a dropped event, an event outside the window, and a message removed by filtering should not collapse into one Boolean value. This would connect formal checking more closely to the incomplete histories encountered in practice. An explicit uncertainty status can coexist with proofs of narrower statements about the available trace.

We also see an educational use. A diagnostic pattern can be taught together with a small positive case, a counterexample to an overgeneralization, and the premise that separates them. The learner can change the evidence or the model and ask which statements remain justified. In this way, the formal companion becomes another way of exploring a pattern's applicability and limits.

References

[1] Vostokov, D. Theoretical Software Diagnostics: Collected Articles. Fourth Edition. OpenTask, 2024. ISBN 978-1-912636-91-4. https://dumpanalysis.org/theoretical-software-diagnostics-book

[2] Vostokov, D. Trace, Log, Text, Narrative, Data: An Analysis Pattern Reference for Information Mining, Diagnostics, Anomaly Detection. Fifth Edition. OpenTask, 2023. ISBN 978-1-912636-58-7. See especially Missing Message. https://dumpanalysis.org/trace-log-analysis-pattern-reference

[3] de Moura, L., and Ullrich, S. The Lean 4 Theorem Prover and Programming Language. Automated Deduction — CADE 28, 2021, pp. 625–635. https://doi.org/10.1007/978-3-030-79876-5_37. Author manuscript: https://lean-lang.org/papers/lean4.pdf

[4] Avigad, J., de Moura, L., Kong, S., and Ullrich, S., with community contributions. Theorem Proving in Lean 4. Online text, especially Propositions and Proofs, Inductive Types, and Axioms and Computation. https://lean-lang.org/theorem_proving_in_lean4/

[5] Theorem Proving in Lean 4. Tactics. https://lean-lang.org/theorem_proving_in_lean4/Tactics/

[6] Lean. Install Lean. Official installation instructions. https://lean-lang.org/install/ Toolchain contents: https://lean-lang.org/doc/reference/latest/Build-Tools-and-Distribution/

[7] Lean project. Lean 4.19.0 release. Toolchain used for the accompanying examples. https://github.com/leanprover/lean4/releases/tag/v4.19.0

[8] Lean. The Lean Language Reference. Validating a Lean Proof. https://lean-lang.org/doc/reference/latest/ValidatingProofs/

[9] McConnell, R. M., Mehlhorn, K., Näher, S., and Schweitzer, P. Certifying algorithms. Computer Science Review 5(2), 2011, pp. 119–161. https://doi.org/10.1016/j.cosrev.2010.09.009. Institutional record: https://researchportalplus.anu.edu.au/en/publications/certifying-algorithms/

[10] Reiter, R. A theory of diagnosis from first principles. Artificial Intelligence 32(1), 1987, pp. 57–95. https://doi.org/10.1016/0004-3702(87)90062-2. Paper: https://cse.sc.edu/~mgv/csce580f11/gradPres/reiter-diagnosis.pdf

[11] Necula, G. C. Proof-Carrying Code. Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, 1997, pp. 106–119. https://doi.org/10.1145/263699.263712. Author's publication record and paper links: https://people.eecs.berkeley.edu/~necula/papers.html

Online references consulted on 15 October 2026; installation details and references [6] and [11] updated on 28 October 2026; repository link added on 30 October 2026. The online Lean documentation may evolve independently of the pinned example toolchain.