VERIBENCH-INFINITY: DETECTING VERIFICATION HALLUCINATIONS, VALID PROOFS OF THE WRONG PROGRAM

Anonymous authors
Paper under double-blind review

ABSTRACT

Formal verification aims to prove that a program satisfies its specification for every valid input, using an interactive theorem prover such as Lean 4. Yet AI agents can produce valid proofs about mistranslated programs or weakened requirements, creating false confidence in software that still contains bugs. We introduce **VeriBench-Infinity**, a benchmark and evaluator that detects these *verification hallucinations* by checking whether AI-generated program translations and specifications preserve the source program's behavior and intended requirements. For each Python program, we independently construct a reference implementation and specification in Lean. We cross-check the reference against Python through execution tests and independent review. Rather than accepting LLM judgment or agreement on finite tests, a separate prover must **formally establish** that the agent and reference agree on observable behavior and requirements for **every valid input** represented by the reference. These comparisons accommodate different data representations and produce machine-checked proofs or concrete counterexamples; incomplete proof searches remain unresolved. We evaluate three agents on 48 program instances pairing correct and buggy versions of competition problems and repository functions. Our analysis exposes translation and specification errors that survive LLM review, while separate case studies identify bugs in CPython's standard library and `intervaltree`. VeriBench-Infinity shifts evaluation from rewarding valid proofs in isolation toward measuring whether AI-generated correctness claims justify trust in the software developers actually use.

1 INTRODUCTION

Formal verification aims to prove that a program's output satisfies certain specifications describing its intended behavior, by modelling the program in an interactive theorem prover such as Lean, and proving statements about this model. Such a proof would show that the specifications are satisfied over an unbounded input domain (Loughridge et al., 2024; Ye et al., 2025; Thakur et al., 2025), unlike test-based checks, which cover only finitely many inputs (Gas Station Manager, 2024). However, any compiling proof only applies to the model of the program in Lean, unless further steps are taken to ensure that the agent faithfully models the source. For example, an agent may simplify the algorithm, omit an exception, or inadvertently repair a bug and still produce a valid Lean proof. We call this a *verification hallucination*: the proof is correct, but it verifies the wrong program or contract.

A simple off-by-one error illustrates the risk. Suppose a Python function is intended to return $\max(x, 0)$ but returns $x + 1$ when $x \geq 0$. If an agent silently translates that branch as x while modeling the program in Lean, Lean can prove that its translation returns $\max(x, 0)$, although Python returns 1 on input 0 and the Lean model returns 0. Such a proof would compile, but applies to a repaired program, not the source. (Appendix A discusses real-library examples of the same failure mode.) A benchmark that measures program verification ability thus has to measure the agent's ability to faithfully model the source, not just prove that the specifications hold for its own model.

The main obstacle is the language boundary: Lean can prove facts about Lean, but cannot directly establish that executable Python and a Lean translation agree, since there is no canonical verified

Reviewers: please read the Reviewer Guidelines (iclr.cc/Conferences/2027/ReviewerGuidelines) and the AI Policy for Reviewers (iclr.cc/Conferences/2027/AIPolicyForReviewers). If you used AI to expand, edit, or polish your review, please provide the input text to the LLM. Better still, consider skipping the LLM and submitting your original text: we, and the authors, are much more interested in your unedited thoughts than in what an LLM has to say. AI-assisted or not, you are putting your name and reputation behind your review: LLM-generated falsehoods, hallucinations or misrepresentations are subject to disciplinary action, which may include desk-rejecting all papers you have authored.

Python to Lean translation. We can solve this problem by independently curating a reference Lean translation for each source. To ensure that these references are faithful and accurate, these references undergo strict audit reviews as well as differential tests against their original Python source. Now, when we evaluate an agent that produces a Lean translation and specification, a separate prover checks the agreement of the agent and reference implementations over the entire valid input set, in the spirit of translation validation and verified lifting (Pnueli et al., 1998; Bhatia et al., 2024). The role of this prover is analogous to a differential tester that is able to check that both implementations match over an infinite set of inputs, rather than a generated finite set—"infinite differential testing", in a sense.

The caveat is that such a prover can only verify external behavior, not if the internal steps of the translation actually carry out the same computation (such as replacing a complex, optimized helper function with a naive brute force). Thus, in addition to a proof judge, we use a semantic LLM judge to test for syntactic faithfulness, which reads the two functions and evaluates how well the internal steps of each function match. We define *implementation fidelity* as both preserving the internal steps of the function and completely matching the external behavior.

Because a canonical Python to Lean translation does not exist, the agent may choose to represent the inputs and outputs of a Python program using a different Lean type as the reference. For example, the agent may translate a Python output of a list of integers to a Lean input of `List Int`, while the reference uses `Array Int`. As such, a prover would not be able to prove that the functions behaved identically, since the two types are not even comparable. We resolve this through the use of LLM-generated relations that connect agent and reference inputs and outputs even when their representations differ. We then check for equivalent behavior and contracts: related valid inputs must produce related observable outputs (Benton, 2004), and the two contracts must permit the same behaviors. The relations themselves are audited and subjected to coverage obligations.

Using these proof-based and semantic evaluators, we evaluate agents on 48 instances derived from 24 tasks, each with a correct and intentionally incorrect source variant. Twelve are full International Olympiad in Informatics (IOI) problems: their precise input/output requirements and nontrivial algorithms stress faithful translation over large input domains. The other twelve are repository functions, including tasks adapted from Vero (Ye et al., 2026); they add dependency context, state, exceptions, and library behavior less visible in self-contained contest tasks. The resulting evaluation separates source translation, contract formalization, formal comparison, and proof or counterexample construction, exposing failures that a single proof-completion score hides.

Our contributions are:

- **Verification hallucinations as an evaluation target.** We identify and name the failure mode in which an agent produces a kernel-checked Lean proof about a mistranslated program or a weakened specification. We evaluate fidelity to the source and its requirements, not only proof completion.
- **Semantic review and infinite differential testing.** We combine semantic review of source meaning and internal computation with eight Lean proof obligations for observable behavior and specification equivalence. LLM-generated relations connect independently chosen agent and reference representations. A separate prover checks the obligations over the entire reference-valid domain, producing kernel-checked proofs or counterexamples; incomplete searches remain unresolved.
- **A frozen three-stage protocol.** Translation, specification, and proof are produced in fresh contexts, with completed artifacts hash-frozen before the next stage. This prevents later proof work from silently simplifying the implementation or weakening its specification and makes failures attributable to individual stages.
- **VeriBench-Infinity.** We introduce 24 tasks—12 full IOI problems and 12 repository functions—each paired with correct and bug-preserving variants, yielding 48 instances. Each instance includes an independently curated, differentially tested Lean reference, with targeted bug witnesses.
- **Empirical findings.** Across three evaluated agents, semantic review catches structural changes that pass formal comparison, and checked counterexamples expose behavioral mismatches that semantic review accepts. Reference-based semantic review catches more of these mismatches than Python-only review, but the Astra–Terra comparison shows that it remains model-dependent. Separate case studies identify bugs in CPython's `textwrap` and the `intervaltree` library.

2 Related Work

Table 1 displays the presence or absence of four features of autoformalization benchmarks for the closest related works.

We have not encountered a formal behavioral equivalence checker in prior autoformalization benchmarks. VeriBench, the closest prior work, uses an LLM judge to assess whether agent theorems cover reference theorems; it does not explicitly compare agent and reference implementations or establish full specification equivalence (Miranda et al., 2026). BEq compares generated and reference formal mathematical statements, while ShadowBench checks generated theorems against intended "shadow" statements in both directions (Liu et al., 2025; Han et al., 2026). These are analogues of our specification-fidelity evaluations, not our separate implementation-fidelity evaluations. VeriEquivBench proves that generated Dafny code satisfies its own specification and that the specification fully constrains the code's output; it checks intent separately through LLM judgment (Zeng et al., 2026).

FormalizeWithTest validates specifications on finite examples instead of proving that they hold on the entire input domain. (Gas Station Manager, 2024). CLEVER and VERINA evaluate whether an agent can create a program that satisfies supplied contracts directly in the proving language, rather than preservation of a source implementation (Thakur et al., 2025; Ye et al., 2025). Vero scales this idea to repository functions (Ye et al., 2026). LLMLift and translation validation prove program equivalence in different settings. Because their translation pipelines constrain the representations, they do not need the same checks for independently chosen representations (Bhatia et al., 2024; Pnueli et al., 1998). They receive partial implementation-fidelity credit because they formally check external behavior, but do not evaluate preservation of the source's internal algorithmic structure.

Table 1: Features of closely related evaluations. Green cells indicate supported features, red cells indicate absent or out-of-scope features, and yellow cells indicate partial or constrained support. Full-domain proof covers whether the program's specification is proven to hold on all valid inputs. Implementation fidelity means preservation of the supplied source program's internal algorithmic structure and external behavior; specification fidelity means agreement between the agent and reference specifications.

	Implementation fidelity	Specification fidelity	Full-domain proof	Independent representations
BEq (Liu et al., 2025)	×	✓	✓	×
ShadowBench (Han et al., 2026)	×	✓	✓	◐
VeriEquivBench (Zeng et al., 2026)	×	◐	✓	×
FormalizeWithTest (Gas Station Manager, 2024)	×	◐	×	×
VeriBench (Miranda et al., 2026)	×	◐	✓	×
CLEVER (Thakur et al., 2025)	×	✓	✓	×
VERINA (Ye et al., 2025)	×	◐	×	×
Vero (Ye et al., 2026)	◐	×	✓	×
LLMLift (Bhatia et al., 2024)	◐	×	✓	◐
Translation validation (Pnueli et al., 1998)	◐	×	✓	◐
VeriBench-Infinity	✓	✓	✓	✓

3 VeriBench-Infinity

We cover the components of the VeriBench-Infinity benchmark and evaluators. Primarily, these include: the **reference (gold) packages** that contain the Python source, descriptions, context, curated Lean model, and Lean-verified proofs or counterexamples that the Lean model satisfies the contract; the **agent running protocol** that dictates the steps in which the agent returns its model and proofs; and the **semantic and proof judges** that measure whether the agent successfully model the Python source in Lean.

3.1 GOLD REFERENCE PACKAGE

The primary parts of the a gold package are a fixed Python function, as well as the *intent*: a natural-language description of the specification that the function should satisfy. Half of the Python functions do satisfy the contract; the other half have deliberate bugs introduced, chosen based on observed difficulties for agents to preserve the bugs in their Lean models without inadvertently repairing them. Some functions call other functions in the repository during runtime, so these other functions along with the rest of the repository are given to the agent as context. In addition, some stateful functions require an "API boundary" to determine exactly what constitutes the observable state; see Appendix B.

The reference package also supplies an independently curated Lean implementation and contract. These were generated with GPT-6 Astra, and passed strict semantic review by independent Claude Fable 5.1 and Astra judges to ensure fidelity to the Python source. In addition to this, a Python–Lean adapter was created for every translation, so that the implementations could be checked against each other on a finite set of inputs. For buggy instances, we check that the same input reproduces the defect in Python and gold.

The gold package finally contains the Lean-verified proof that the Lean model satisfies the contract, or a counterexample to show that it doesn't. These were generated with GPT-6 Astra.

Representing Python behavior. Some functions do not translate cleanly to Lean. Stateful functions cannot be represented directly because Lean functions are stateless, while potentially non-terminating functions need an encoding compatible with Lean's termination checker. We expose observable pre- and post-state at the declared API boundary, and use either an ordinary total function or a fuelled execution returning a completed result or `outOfFuel`. Appendix B defines both representations and their stability requirements.

3.2 FROZEN THREE-STAGE PROTOCOL

We introduce a three-stage protocol in which an agent generates the Lean translation, contract, and proof respectively, freezing outputs produced in each state before starting the next. CLEVER similarly separates certification of a generated Lean specification from certification of an implementation and its correctness proof (Thakur et al., 2025). The three-stage protocol was used due to observed difficulties for the agent to provide a faithful Lean translation when asked to produce all artifacts in a single stage: in an early one-stage trial based on the IOI 2025 problem Triple Peaks, the GPT-5.6 Sol agent produced a faithful optimized $O(N\sqrt{N})$ translation, but proof difficulty led the agent to degrade the algorithm to $O(N^3)$; even though the external behavior was preserved, the Lean translation no longer represented the actual internal behavior of the Python function. The agent run with this harness would likely exhibit semantic failures on many tasks, so the three-stage harness was used to prevent such reward hacking and represent each agent's true capabilities better.

We call the three agent stages C1 (source translation), C2 (contract formalization), and C3 (proof or refutation), and call an artifact *frozen* once its bytes are immutable. Each stage receives only its designated inputs in a fresh context; completed artifacts are hash-recorded and cannot be revised later.

C1: Source translation. The agent receives the Python source, dependencies, and natural language intent. It must construct Lean input and output types that describe the corresponding Python input and output types, as well as a natural language description of how the Lean types correspond with the Python types. It also translates the Python source to a Lean function using the defined Lean types as input/output. We denote A_I, A_O for its input/output types and $f_A : A_I \to A_O$ for its implementation.

C2: Contract formalization. The agent receives the natural-language intent of the Python function, API boundary, source context, and C1 public interface—but not the C1 implementation. It must formalize the natural-language intent by constructing *preconditions* (denoted with $P_A : A_I \to \mathrm{Prop}$) which describe which inputs are valid, and *postconditions* (denoted with $Q_A : A_I \to A_O \to \mathrm{Prop}$) which describe what function outputs are valid for given inputs. For example, if the natural intent states that a function takes an integer between 1 and 5 and adds one

to it, the precondition would check that the input is indeed between 1 and 5, and the postcondition would check that the output is one plus the input.

C3: Correctness proof or refutation. The C1 and C2 outputs are semantically and formally judged on implementation and specification fidelity (judges described in the next section). The C3 call runs only after both the semantic and formal judgments pass. The agent receives its frozen outputs from the previous two calls: they now contain a Lean function, along with a postcondition that a function output should satisfy given an input. Its job is to now prove or disprove that the Lean implementation indeed satisfies the contract; symbolically: $\forall a : A_I,\ P_A(a) \Rightarrow Q_A(a, f_A(a))$. (For fuelled implementations, F1 separately proves that the fuelled implementation is valid; Appendix C gives the obligations.)

Figure 1 shows the semantic and formal judges alongside the three agent stages.

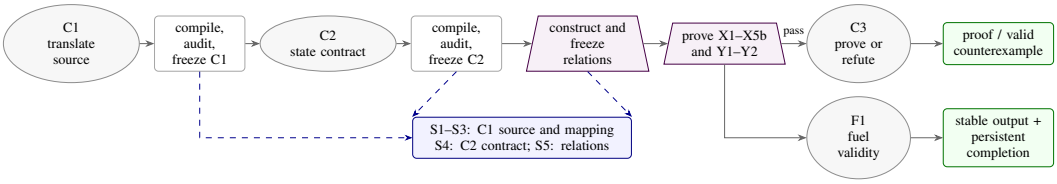

Figure 1: Judging protocol. Ellipses denote evaluated-agent calls, trapezoids evaluator-model calls, and rectangles pipeline operations or outcomes.

3.3 Semantic and formal judges

We describe the semantic and formal judges used to evaluate the C1 and C2 calls (since C3 is judged completely mechanically in Lean, testing if the agent produces a compiling proof or counterexample with no additional axioms). Semantic judges are LLM judges that are given the agent artifact along with a corresponding gold artifact, and are asked to evaluate whether the agent artifact is faithful to the gold artifact in some way. Proof judges are given a formal target in Lean that describes this faithfulness in the form of a concrete proposition, and the result is decided by whether the judge proves the proposition or finds a counterexample. Proof judges are allowed to use the gold package proofs or counterexamples as helpers. Astra is used for all proof judges; we use GPT-6 Astra for the main semantic judge evaluations. We also run an independent GPT-5.6 Terra replay (S1′–S5′) to determine the impact of model strength on semantic judge outcomes.

Below we describe the case where neither the agent nor gold translation uses fuelled mode; we call this *direct-total mode*. This case suffices to explain the main agent–reference comparison.

As stated before, the agent and gold translations may use different representations, requiring the use of a *relation* between the agent types and output types. (A relation is a set of corresponding pairs (Weisstein, 2026); here, it is a Lean proposition saying when agent and reference values represent the same Python input or output.) Denote A_I, A_O for the agent input and output types, G_I, G_O for the gold types, f_A, f_G for their implementations, and P_A, Q_A, P_G, Q_G for their preconditions and postconditions. After C1 is completed, an LLM (Astra) uses the intent, natural language explanations of the Lean inputs and outputs, and source code to construct the input and output relations: $R_I : A_I \to G_I \to \text{Prop}$; $R_O : A_I \to G_I \to A_O \to G_O \to \text{Prop}$. $R_I(a, g)$ asserts that a and g represent the same source input, while $R_O(a, g, o_A, o_G)$ asserts that the outputs represent the same observable result for those inputs. The relations need not be functions or include every equivalent pair: their semantic soundness is reviewed separately from the formal coverage checks.

Our benchmark contains eight proof obligations: X1 through X5a/b check C1's observable behavior and representation consistency and coverage; Y1 and Y2 check C2's specifications. The semantic judges provide complementary reviews: S1 and S2 compare the internal behavior of C1 with Python and gold, respectively (alongside X1); S3 grades the fidelity of C1's types and mapping (alongside X2–X5b); S4 reviews whether the specifications emitted by C2 match the intent and gold (alongside Y1–Y2); and S5 reviews the relations' source-level meaning.

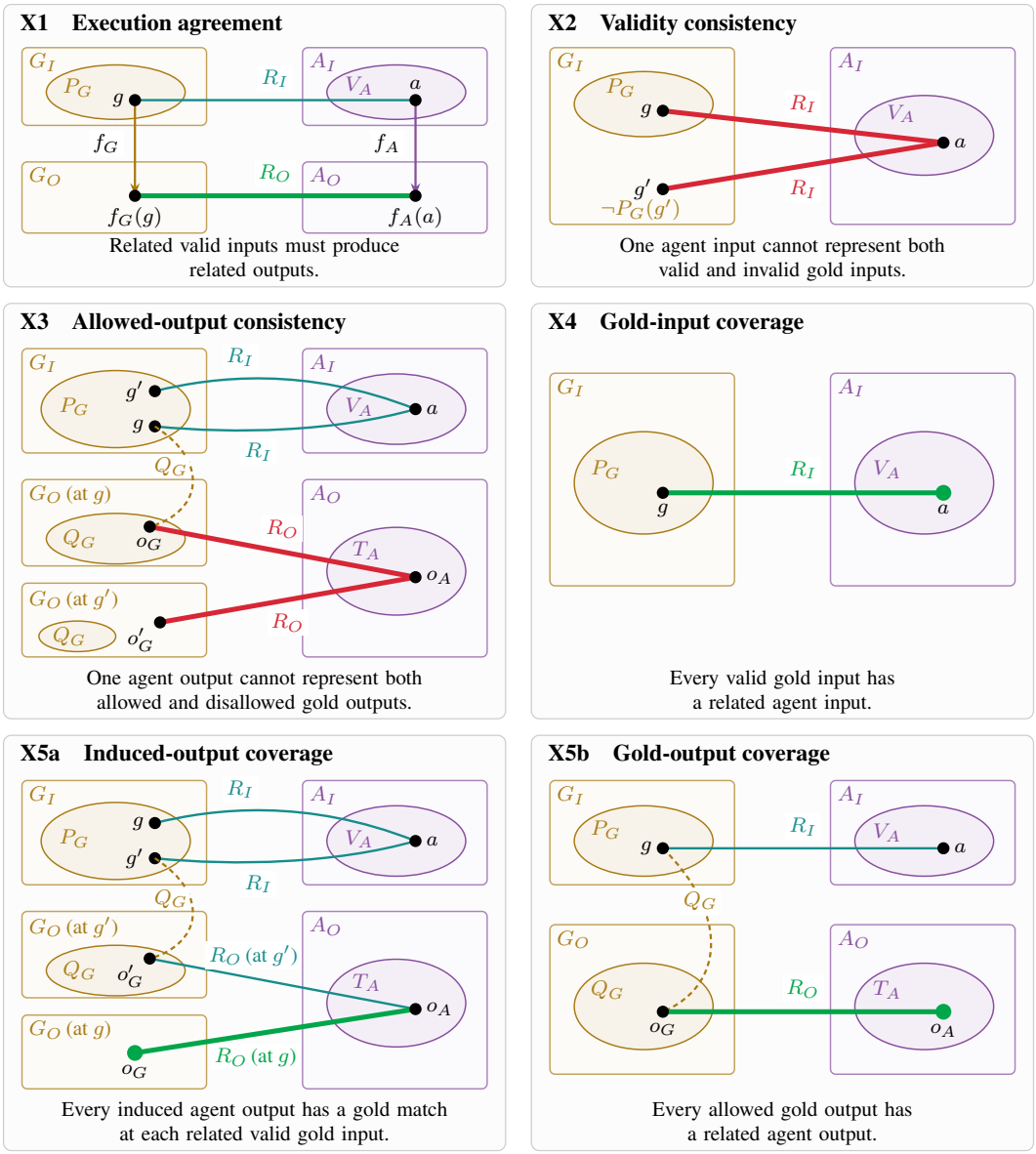

Figure 2: The six X obligations for direct-total implementations, two per row. Ochre and purple denote gold and agent types; shaded ovals select valid inputs or allowed outputs. Dashed ochre links show Q_G from input to output. Teal links are premises; thick green links and dots are required; red forks are forbidden together. The separate gold-output boxes in X3 and X5a distinguish the contexts g and g'. In X5a, only the output at g' has a Q_G premise; the output at g is the required relation witness.

Figure 2 illustrates the six X obligations. Here, V_A denotes agent inputs related to valid gold inputs, and T_A denotes agent outputs related to allowed gold outputs at corresponding valid inputs; as such, we expect these to match the formal pre/postconditions P_A, Q_A emitted by C2, which the formal judges Y1 and Y2 check. Appendix D gives the formal definitions, proof obligations, and C3 validation rules, with a proof sketch showing why the X and Y obligations suffice to establish equivalent specifications and observable behavior on valid inputs under the chosen relations.

4 Experimental Setup

We evaluate three agents with their respective harnesses: GPT-6 Astra with Codex, GPT-5.6 Sol with Codex, and Opus 5.1 with Claude Code, on every $(\text{task}, \text{variant})$ pair (where $\text{variant} \in \{\text{correct}, \text{incorrect}\}$). This yields 144 independent agent rows. All models are accessed through provider APIs.

C1, C2, and C3 each have a 64,000-output-token limit; in C3, a model may request to preserve its current progress on the proof and continue from a fresh context, but the total output tokens over all continuations may not exceed 64,000. Isolation details are in Appendix E.

5 Results

We analyze the 144 agent rows in Tables 4 and 5, separating semantic review, formal comparison, and final verification.

5.1 Where agents fail

Table 2 separates failures to produce an artifact, fidelity rejections, and failures to complete the final proof or counterexample. Opus exhausts the C1 translation budget on 26 of 48 instances; Astra and Sol produce C1 translations and specifications for all 48. Among Sol's translations, semantic implementation review rejects 22, and formal implementation comparison refutes 19.

Table 2: Failures by check family and final outcomes for correct (C) and incorrect (I) sources. Above the double line, fractions use evaluated submissions as denominators. Below it, mutually exclusive counts sum to 24 per column; C3 outcomes count only after joint fidelity acceptance. Success requires all checks plus a correctness proof (C) or counterexample (I).

Failure or outcome	Astra		Sol		Opus	
	C	I	C	I	C	I
C1 token-limit exhaustion	0/24	0/24	0/24	0/24	11/24	15/24
Implementation review rejects (S1–S3)	1/24	1/24	8/24	14/24	1/13	1/9
Implementation comparison refuted (X)	1/24	1/24	7/24	12/24	1/13	0/9
C2 token-limit exhaustion	0/24	0/24	0/24	0/24	0/13	0/9
Specification/relation review rejects (S4–S5)	1/24	1/24	4/24	5/24	1/13	1/9
Specification comparison refuted (Y)	1/24	1/24	4/24	5/24	0/13	1/9
C3 token-limit exhaustion	17/23	1/22	16/16	1/8	11/11	0/7
Final outcomes						
Call 1 token limit	0	0	0	0	11	15
Semantic rejection only	0	0	1	2	1	1
Proof rejection only	0	0	0	0	0	0
Both rejections	1	2	7	14	1	1
Call 3 token limit	17	1	16	1	11	0
Success	6	21	0	7	0	7

After C1 and C2 are accepted, C3 exhausts its budget on 44 of 50 correct-source submissions, compared with 2 of 37 incorrect-source submissions. Only Astra completes proofs for correct sources

(6 of 24 instances). On incorrect sources, Astra, Sol, and Opus produce counterexamples satisfying all acceptance requirements on 21, 7, and 7 of 24 instances, respectively. Even after fidelity checks pass, agents often exhaust their budget trying to prove correct programs.

5.2 Semantic and formal checks are complementary

Our results support combining semantic review with formal comparison, rather than treating either as a substitute for the other. Semantic review assesses source-level meaning and internal algorithmic structure, while formal comparison checks observable behavior and specifications under the reviewed reference and relations. These are separate conditions requiring different types of judges: semantic judgment can never provide a formal proof for external behavior agreement, and external behavior proof judges can never detect mismatches in internal implementation.

We emphasize the importance of having a proof judge whose completed verdict does not depend on model strength: a weak model can only leave a proof obligation unfulfilled, and emitting a false acceptance or false rejection for the Lean proposition is impossible (unless a kernel bug in Lean is exploited). Recall that the S1 semantic judge judges implementation fidelity by comparing the agent Lean against the Python source; the S2 semantic judge instead compares against the gold reference Lean. While human review was done for every disagreement between the semantic and formal judges, and agreed with the rejecting side on all instances, the S1 Astra semantic judge, comparing Python against agent Lean, emitted 3 false acceptances in our benchmark. Against the human-reviewed Astra S2 verdicts, Terra makes 29 errors in S1′ (23 false acceptances, 6 false rejections) versus 3 in S2′ (2 false acceptances, 1 false rejection) across the same 118 submissions. S2′ reaches 97.46% accuracy (115/118), but semantic judgment still depends on model strength.

Table 3 compares the Python-facing and gold-facing reviews on the same X1-refuted translations. The gold-facing check catches more mismatches for both judges, but its reliability still depends on the judge model. Terra accepts Sol's incorrect-source Split translation even in S2′, whereas Astra rejects it and X1 supplies a checked counterexample. Comparing agent Lean implementation against the gold reference instead of the Python source improves semantic review; it does not turn that review into a proof.

Table 3: Semantic acceptances among the same 21 translations with X1 counterexamples, drawn from 118 rows with both judges' S1/S2 results and a terminal X1 verdict. These counts measure missed behavioral mismatches on this subset, not overall semantic-judge error rates.

Judge	Python-facing S1/S1′	Gold-facing S2/S2′
Astra primary	3/21	0/21
Terra replay	19/21	1/21

5.3 Cost and practicality

Formal judging requires more inference and Lean checking than semantic classification and can require several proof contexts. We preserve proof progress across continuations; checkpointing and reusable gold packages make benchmark evaluation practical, but this is not a low-cost replacement for ordinary testing. The 118 rows with recorded proof costs in Table 4 total $4,123.30. Per-row costs have a mean of $34.94 and median of $26.35, ranging from $4.59 to $118.29. These figures exclude agent calls and earlier failed comparison attempts.

The 24-task sample is also too small to establish performance on all domains, libraries, or classes of defects.

6 Limitations

Trust boundary. Lean proves correspondence to the gold model, not directly to a formal Python semantics. Differential tests, four semantic audits, targeted bug witnesses, and human review reduce this risk but cannot eliminate it. Moreover, the versioned Python model does not cover arbitrary reflection, native extensions, external I/O, concurrency, or environment interaction; such features

require task-specific abstractions. A faulty gold translation, incomplete abstraction, or ambiguous intent can invalidate downstream conclusions.

Astra across roles. Astra authored the gold translations and proofs, constructs the relations, and serves as proof judge and primary semantic judge while also being an evaluated agent. All proof-judge verdicts are kernel-checked and therefore model-independent; the residual risk is that gold conventions favor Astra's representations and that semantic blind spots correlate. We mitigate this with Fable audits of every gold package, Python–Lean differential tests, targeted bug witnesses, human adjudication of disagreements, and the independent Terra replay.

The observation boundary matters. X1–X5b and Y1–Y2 certify the behavior encoded by R_I, R_O, and the contracts. They do not prove equality of internal traces, data structures, time complexity, memory use, or unspecified runtime behavior. Implementation-fidelity judges address some of these concerns semantically, but no kernel-checked algorithm-fidelity theorem is claimed unless the property is explicitly formalized.

7 Conclusion

VeriBench-Infinity makes verification hallucinations an explicit evaluation target: a valid proof can certify a mistranslated program or a weakened specification. We address the Python–Lean boundary by curating a separate Lean reference, checked against Python through differential tests, targeted bug witnesses, and independent review. This enables formal comparison of the agent and reference implementations as well as their specifications, rather than treating proof completion or statement equivalence alone as evidence of fidelity to the source.

The pipeline separates translation, specification, and proof into fresh contexts and freezes each completed artifact. Later proof work therefore cannot silently simplify the implementation or weaken its specification, and failures can be attributed to individual stages. Reviewed relations connect independently chosen Lean representations, while eight proof obligations check execution agreement, representation consistency and coverage, and specification equivalence over the full reference-valid domain. Semantic judges review source meaning, internal algorithmic structure, and the meaning of the relations. Only submissions accepted by both semantic and formal judges proceed to correctness proofs or bug counterexamples; frozen artifacts and validation records make the outcomes auditable.

The benchmark pairs correct and buggy versions of 12 full IOI problems and 12 repository functions, producing 48 instances and 144 submissions from three agents. These pairs distinguish preserving a source bug from inadvertently repairing it during translation. The results expose failures before and after fidelity checking: Opus exhausts its translation budget on 26 of 48 instances, and formal implementation comparison refutes 19 of Sol's 48 translations. After fidelity acceptance, agents complete correctness proofs for only 6 of 50 correct-source submissions, but find counterexamples for 35 of 37 incorrect-source submissions. Faithful formalization and successful proof construction remain distinct challenges.

Gold references also improve semantic review. Across 118 submissions, comparing agent Lean with gold Lean rather than Python reduces Terra's total errors from 29 to 3, counting both false acceptances and false rejections against the human-reviewed Astra S2 verdicts. Gold-based review remains model-dependent; formal verdicts instead carry kernel-checked evidence about the stated Lean propositions. Conversely, semantic review can reject internal implementation changes that preserve observable outputs. Separate case studies uncover nontermination in CPython's `textwrap` and balancing defects in `intervaltree`, demonstrating why verification must account for state, exceptions, and termination behavior.

The remaining trust boundary is the connection from Python and intended requirements to the reference and relations; checking downstream proofs does not establish that connection. Future work should extend the evaluation to whole repositories by composing guarantees across function calls, shared state, and dependencies. Other priorities are larger and more diverse benchmarks, lower-cost reference curation and proof search, and formal checks of internal behavior such as execution traces and resource use. The goal is not merely more verified theorems, but stronger evidence about the software those theorems claim to verify.

REPRODUCIBILITY STATEMENT

We provide an anonymized, representative artifact ZIP in the supplementary material. It includes six reference packages, frozen artifacts for all six agent/variant runs of one benchmark task, the X/Y Lean obligation templates, and validation code with reproduction instructions. The included artifacts are accompanied by versioned configurations and evidence receipts. Appendix E describes reference curation, the evaluation protocol, and the recorded dependencies, prompts, budgets, and validation checks.

AI USE STATEMENT

Generative models were used for evaluated source translation, contract formalization, proof and counterexample search, relation construction, semantic judgment, gold-package development and auditing, experiment orchestration, and writing assistance. Evaluated agents and model-based judges were accessed through their providers' APIs rather than executed from local model weights. The authors selected the research questions and evaluation criteria, reviewed AI-assisted artifacts, and remain responsible for all claims. Kernel-checked claims, model judgments, and human curation are labeled separately.

REFERENCES

Nick Benton. Simple relational correctness proofs for static analyses and program transformations. In *Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pp. 14–25, 2004. doi: 10.1145/964001.964003. URL https://doi.org/10.1145/964001.964003.

Sahil Bhatia, Jie Qiu, Niranjan Hasabnis, Sanjit A. Seshia, and Alvin Cheung. Verified code transpilation with LLMs. In *Advances in Neural Information Processing Systems*, volume 37, 2024. doi: 10.52202/079017-1310. URL https://proceedings.neurips.cc/paper_files/paper/2024/hash/48bb60a0c0aebb4142bf314bd1a5c6a0-Abstract-Conference.html.

Gas Station Manager. FormalizeWithTest: Automated verification for autoformalization of coding problems. https://github.com/GasStationManager/FormalizeWithTest, 2024. Software artifact.

Hojae Han, Jongyoon Kim, Sanghyeok Park, Dongwook Cheon, Yeachan Park, Myeong Jae Jeon, Sunjong Choe, Soonho Kong, Wonseok Hur, Seung-won Hwang, and Donghoon Hyeon. SHADOWBENCH: Toward reliable automatic evaluation of semantic alignment in autoformalization. *arXiv preprint arXiv:2608.29270*, 2026. URL https://arxiv.org/abs/2608.29270.

Qi Liu, Xinhao Zheng, Xudong Lu, Qinxiang Cao, and Junchi Yan. Rethinking and improving autoformalization: Towards a faithful metric and a dependency retrieval-based approach. In *International Conference on Learning Representations*, 2025. URL https://openreview.net/forum?id=hUb2At2DsQ.

Chloe Loughridge, Qinyi Sun, Seth Ahrenbach, Federico Cassano, Chuyue Sun, Ying Sheng, Anish Mudide, Md Rakib Hossain Misu, Nada Amin, and Max Tegmark. DafnyBench: A benchmark for formal software verification. *arXiv preprint arXiv:2406.08467*, 2024. URL https://arxiv.org/abs/2406.08467.

Brando Miranda, Srivatsava Daruru, Ethan S. Hersch, Zhanke Zhou, Allen Nie, Daneshvar Amrollahi, Leni Aniva, Iddah Mlauzi, Kirill Acharya, Elyas Obbad, Dilara Soylu, Weston Kirk, Zixiao Jolene Wang, Kai Fronsdal, Ying Li, Donald Poindexter, Rakshit Kaushik, Shurui Liu, Yegor Denisov-Blanch, Steven Dillmann, Simon Obstbaum, Santiago Cuellar, John Sarracino, Rylan Schaeffer, Mo Tiwari, Donghyun Lee, Bo Han, and Sanmi Koyejo. VeriBench: An end-to-end formal verification benchmark for AI coding agents in Lean 4. In *3rd AI for Math Workshop: Toward Self-Evolving Scientific Agents*, 2026. URL https://openreview.net/forum?id=lkL0qnUv3p.

Amir Pnueli, Michael Siegel, and Eli Singerman. Translation validation. In *Tools and Algorithms for the Construction and Analysis of Systems*, pp. 151–166, 1998. doi: 10.1007/BFb0054170. URL https://doi.org/10.1007/BFb0054170.

Amitayush Thakur, Jasper Lee, George Tsoukalas, Meghana Sistla, Matthew Zhao, Stefan Zetzsche, Greg Durrett, Yisong Yue, and Swarat Chaudhuri. CLEVER: A curated benchmark for formally verified code generation. In *Advances in Neural Information Processing Systems, Datasets and Benchmarks Track*, 2025. URL https://arxiv.org/abs/2505.13938.

Eric W. Weisstein. Relation. https://mathworld.wolfram.com/Relation.html, 2026. MathWorld—A Wolfram Resource; accessed September 25, 2026.

Zhe Ye, Zhengxu Yan, Jingxuan He, Timothe Kasriel, Kaiyu Yang, and Dawn Song. VERINA: Benchmarking verifiable code generation. *arXiv preprint arXiv:2505.23135*, 2025. URL https://arxiv.org/abs/2505.23135.

Zhe Ye, Hantao Lou, Yuechun Sun, Peiyang Song, Zhengxu Yan, Timothe Kasriel, Qingyang Zhang, Kaiyu Yang, Soonho Kong, Jingxuan He, and Dawn Song. Vero: Can AI agents build formally verified software repositories? *arXiv preprint arXiv:2608.13522*, 2026. URL https://arxiv.org/abs/2608.13522.

Lingfei Zeng, Fengdi Che, Xuhan Huang, Fei Ye, Xu Xu, Binhang Yuan, and Jie Fu. VeriEquivBench: An equivalence score for ground-truth-free evaluation of formally verifiable code. In *International Conference on Learning Representations*, 2026. URL https://arxiv.org/abs/2510.06296.

A Implementation-Fidelity Case Study

We study two Python functions from Vero's dataset to demonstrate the importance of implementation fidelity. Vero is a software-synthesis benchmark: it uses manually curated Lean translations to test proof completion or code synthesis, rather than evaluating an agent's ability to preserve a supplied Python implementation. Without semantic or formal bounds from those Lean artifacts to Python, the formal results do not by themselves guarantee the behavior of the original source. When we applied a stricter fidelity review, we found bugs in two widely used sources: CPython's standard library and an established third-party package.

CPython `textwrap`. This function inserts line breaks into a string of words to wrap it to a target width. An infinite loop can be triggered when the tab size overflows the text width. There is no requirement that the tab size be smaller than the width, and the Python source explicitly attempts to handle this case, yet fails. Hence the source does not terminate on every valid input. We store it as the incorrect variant and repair the loop in the correct variant.

`intervaltree`. This function stores intervals in an AVL tree and removes the part of each interval overlapping a range. We found two balancing bugs.

- *Bulk construction.* Passing all intervals directly to the constructor can produce a tree outside the library's AVL invariant. On our witness, the returned interval list is still correct, but the retained tree fails the library's own verification check. We repair this bug by starting with an empty tree and adding intervals one at a time, guaranteeing a balanced and valid exit state.
- *Deletion rebalancing.* During `chop`, intervals drawn from Python sets are deleted in the hash set's iteration order, which Python does not guarantee. Some orders cause a deletion to shrink a subtree by more than one level. The source assumes that one rotation is sufficient, leaving the tree imbalanced so that a later deletion can raise `KeyError`. We pin a CPython version to make this order deterministic. Exposing the behavior required a constructed hash-collision input, found only after an implementation-equivalence prover repeatedly failed to connect iteration order to the balance guarantee.

Our incorrect variant repairs only bulk construction and preserves the iteration-order bug. The correct variant also repairs the rotation logic.

B Representing State and Nontermination

Stateful functions. The agent must represent both how the current state affects execution and how execution changes that state. It defines a Lean type for the relevant state, takes that state as part of the input, and returns the modified state as part of the output.

Because the implementation is fixed before contract formalization, C1 must already know which state is observable. The curator therefore declares an API boundary: return values, relevant exceptions, changes retained after the call, and later API operations that may use that state. This prevents a formalization from discarding information needed by later observations. For example, a cache operation may return nothing while changing the cache; its Lean output must represent the updated cache state. Private storage details need not become public outputs unless the boundary makes them observable. The declaration fixes the scope of observation, not the Lean data types, and does not authorize changing the source algorithm or claim correctness for every later operation.

Potentially nonterminating functions. A direct translation may be impossible when Python does not terminate on every input, because Lean rejects definitions whose termination cannot be established. This includes programs that diverge only outside the contract and buggy programs that diverge on a valid input; the translation must preserve either behavior.

In direct-total mode, the agent defines an ordinary Lean function, so every represented input has a result. In fuelled mode, the function also takes a natural-number bound and returns either a completed output or `outOfFuel`. A terminating source execution must complete once enough fuel is supplied, whereas a diverging source execution never completes.

F1 establishes that the fuel model is stable on all public inputs: completed outputs agree, and completion persists at every larger bound. C3 separately proves eventual completion when the contract requires termination. Conversely, showing that a valid input never completes refutes that correctness claim; one out-of-fuel evaluation does not. Different translations may require different fuel bounds.

C Fuelled Evaluation Obligations

In direct-total mode, each translation is an ordinary Lean function that returns an output for every represented input without a fuel parameter. Fuelled mode instead represents possible nontermination and requires the additional proof obligations below.

Write $F_A(n, a)$ and $F_G(m, g)$ for agent and gold executions with fuel n and m. The agent's F1 certificate requires, for all public inputs and without assuming P_A,

$$\begin{aligned} &F_A(n,a) = \text{finished}(o) \wedge F_A(m,a) = \text{finished}(o') \\ &\qquad \Rightarrow o = o', \end{aligned} \tag{F1a}$$

$$\begin{aligned} &F_A(n,a) = \text{finished}(o) \wedge m \geq n \\ &\qquad \Rightarrow F_A(m,a) = \text{finished}(o). \end{aligned} \tag{F1b}$$

The gold package proves analogous properties on its valid domain. In the gold-centric comparison, X1 assumes $P_G(g) \wedge R_I(a, g)$, not the C2 predicate $P_A(a)$. When both sides are fuelled, X1 is replaced by

$$\begin{aligned} &P_G(g) \wedge R_I(a,g) \wedge F_A(n,a) = \text{finished}(o_A) \\ &\qquad \Rightarrow \exists m, o_G :\ F_G(m,g) = \text{finished}(o_G) \wedge R_O(a,g,o_A,o_G) \end{aligned} \tag{X1a}$$

$$\begin{aligned} &P_G(g) \wedge R_I(a,g) \wedge F_G(m,g) = \text{finished}(o_G) \\ &\qquad \Rightarrow \exists n, o_A :\ F_A(n,a) = \text{finished}(o_A) \wedge R_O(a,g,o_A,o_G). \end{aligned} \tag{X1b}$$

If only one side is fuelled, the other completion equation becomes its ordinary function equation. Fuel bounds need not match. These simulations allow mutual divergence but reject a related valid input on which only one side terminates.

For the C3 target, define $E_A(a, o) := \exists n :\ F_A(n, a) = \text{finished}(o)$. When the contract requires termination, C3 must establish

$$\forall a,\ P_A(a) \Rightarrow \big((\exists o :\ E_A(a,o)) \wedge (\forall o,\ E_A(a,o) \Rightarrow Q_A(a,o))\big). \tag{C3-F}$$

D Semantic and Formal Judge Details

This appendix expands Section 3.3 for direct-total implementations; Appendix C gives the fuelled variants.

D.1 Proof obligations and C3 validation

Gold-induced domain. To compare independently chosen types, we factor the gold contract through the frozen relations. Define the induced agent input domain and allowed-output set by

$$\begin{aligned} V_A(a) &:= \exists g :\ P_G(g) \wedge R_I(a,g), \\ T_A(a,o_A) &:= \exists g, o_G :\ P_G(g) \wedge R_I(a,g) \wedge Q_G(g,o_G) \wedge R_O(a,g,o_A,o_G). \end{aligned}$$

These definitions use gold and the frozen relations, not C2's P_A, Q_A.

Implementation behavior: S1/S2 and X1. S1 reviews whether the C1 Lean implementation preserves the Python source's observable behavior and algorithmic structure; S2 compares that implementation with gold Lean. X1 is their formal counterpart for observable behavior: in the direct-total case, every gold-valid related input pair must produce related outputs,

$$P_G(g) \wedge R_I(a,g) \Rightarrow R_O(a,g,f_A(a),f_G(g)). \tag{X1}$$

X1 compares the two Lean executions; it does not run Python or prove that their internal algorithms match. For fuelled implementations, F1 checks stability and bidirectional simulations replace direct-total X1; Appendix C gives the obligations.

Representation coverage: S3 and X2–X5b. S3 reviews whether C1's public types and Python–Lean mapping cover the source's observable inputs, outputs, and retained state. The X2–X5b obligations formally check consistency and coverage of the *declared* relations. X2 prevents one agent input from representing both a gold-valid and an invalid input. X3 requires the gold contract to agree across multiple gold outputs related to the same agent output. X4 covers every valid gold input, while X5a and X5b ensure that neither the induced contract nor gold permits an output hidden by R_O:

$$R_I(a,g) \wedge R_I(a,g') \Rightarrow \big(P_G(g) \leftrightarrow P_G(g')\big) \tag{X2}$$

$$P_G(g) \wedge P_G(g') \wedge R_I(a,g) \wedge R_I(a,g') \wedge R_O(a,g,o_A,o_G) \wedge R_O(a,g',o_A,o'_G) \Rightarrow \big(Q_G(g,o_G) \leftrightarrow Q_G(g',o'_G)\big) \tag{X3}$$

$$P_G(g) \Rightarrow \exists a:\ R_I(a,g) \tag{X4}$$

$$P_G(g) \wedge R_I(a,g) \wedge P_G(g') \wedge R_I(a,g') \wedge Q_G(g',o'_G) \wedge R_O(a,g',o_A,o'_G) \Rightarrow \exists o_G:\ R_O(a,g,o_A,o_G) \tag{X5a}$$

$$P_G(g) \wedge R_I(a,g) \wedge Q_G(g,o_G) \Rightarrow \exists o_A:\ R_O(a,g,o_A,o_G). \tag{X5b}$$

In X5a, g', o'_G make explicit the witness in $T_A(a, o_A)$; the required output relation must also exist at every related valid g. All free variables are universally quantified. These proofs establish compatibility of C1's chosen types with gold on the valid domain, but cannot establish that the declared mappings faithfully describe Python; that is S3's semantic task.

Specification fidelity: S4 and Y1–Y2. S4 reviews C2's pre/postconditions against the intended requirements and gold, including missing or added requirements. Y1 and Y2 are the formal counterparts: they compare C2's input domain and allowed outputs with the gold-induced sets,

$$P_A(a) \leftrightarrow V_A(a) \tag{Y1}$$

$$V_A(a) \Rightarrow \big(Q_A(a,o_A) \leftrightarrow T_A(a,o_A)\big). \tag{Y2}$$

Given gold and semantically faithful relations, the contract C2 must express is fixed up to logical equivalence: P_A must equal V_A, and Q_A must equal T_A on that domain. Y2 covers even outputs neither implementation produces. A C2 token-limit exhaustion need not suppress X work, but leaves Y unavailable.

Relation meaning: S5. S5 reviews whether R_I and R_O express the intended input and output correspondence. It has no proof-only counterpart: every X and Y obligation assumes these relations, so proving those obligations cannot establish the relations' semantic fidelity to Python. Semantic judgments, formal judgments, and evaluated-agent outcomes are recorded separately.

Proof validation and C3 outcomes. A proof judge must prove or refute its X or Y target against the exact frozen bytes; the reference proof or counterexample alone does not establish the agent–reference comparison. A proof is accepted only after a clean rebuild, kernel checking, an axiom audit, and a hash match. A kernel-checked X or Y counterexample is a formal comparison failure. C3 runs only when C1 and C2 are mechanically valid, the primary semantic judgments S1–S5 pass, and all applicable X and Y obligations pass. It proves or refutes the frozen correctness target, which in direct-total mode is

$$\forall a,\ P_A(a) \Rightarrow Q_A(a, f_A(a)). \tag{C3}$$

A kernel-checked proof is a successful C3 result on a correct source; a kernel-checked valid-input counterexample is successful bug finding on an incorrect source. This differs from an X1 counterexample, which means the agent and gold implementations disagree. Token-limit exhaustion, provider failure, and incomplete proof search are reported separately. Final acceptance also requires the applicable F1 certificate. Appendix C gives the fuelled target.

D.2 Why the obligations suffice

For the direct-total case, a simple target is that the agent satisfies its specification exactly when gold satisfies its own. Write $\mathrm{Correct}_A$ for $\forall a,\ P_A(a) \Rightarrow Q_A(a, f_A(a))$, and $\mathrm{Correct}_G$ for the corresponding gold claim. Assuming X1–X5b and Y1–Y2, we want

$$\mathrm{Correct}_A \iff \mathrm{Correct}_G.$$

We also want to preserve all permitted outputs, not just the outputs that the two implementations happen to return. The following proof sketch establishes both properties.

Valid inputs agree and are covered. For any $R_I(a, g)$, X2 and Y1 give $P_A(a) \leftrightarrow P_G(g)$. If $P_G(g)$ holds, this pair witnesses $V_A(a)$, so Y1 gives $P_A(a)$. Conversely, if $P_A(a)$ holds, Y1 supplies some valid g' related to a; X2 then makes g valid as well. Y1 therefore gives a valid gold match for every valid agent input, and X4 gives a valid agent match for every valid gold input.

Related outputs agree on the specification. Fix $P_G(g)$, $R_I(a, g)$, and $R_O(a, g, o_A, o_G)$. We claim $Q_A(a, o_A) \leftrightarrow Q_G(g, o_G)$. If $Q_G(g, o_G)$ holds, the given pair witnesses $T_A(a, o_A)$, and Y2 gives $Q_A(a, o_A)$. Conversely, if $Q_A(a, o_A)$ holds, Y2 supplies a witness g', o'_G for $T_A(a, o_A)$. Both gold outputs are related to the same o_A at valid inputs related to a, so X3 transfers $Q_G(g', o'_G)$ to $Q_G(g, o_G)$.

No permitted output is lost. Keep the valid input pair a, g fixed. For any permitted agent output o_A, Y2 supplies its T_A witness, and X5a supplies an output o_G related to o_A at this particular g. The preceding argument then gives $Q_G(g, o_G)$. Conversely, X5b supplies a related agent output for every gold-permitted o_G, and the same agreement argument gives $Q_A(a, o_A)$. Thus every permitted output has a permitted counterpart at each related valid input pair.

Correctness transfers in both directions. X1 relates the actual outputs $f_A(a)$ and $f_G(g)$ at every related valid input pair. Applying the specification agreement above gives

$$Q_A(a, f_A(a)) \iff Q_G(g, f_G(g)).$$

If gold is correct, take any valid agent input, choose its valid gold match using Y1, and transfer the gold postcondition through this equivalence. If the agent is correct, take any valid gold input, choose a valid agent match using X4 and Y1, and transfer the agent postcondition in the other direction. This proves the target; likewise, a valid-input counterexample on either side has a related counterexample on the other.

Why additional matching edges are unnecessary. The argument uses the supplied correspondences and their coverage, never the assumption that every equivalent pair is included. Consequently, it does not require complete bipartite connected components or any other rule that fills in missing edges. This is a sufficiency argument, not a claim that every obligation is necessary for the basic correctness equivalence: X5a/X5b additionally prevent permitted outputs from being omitted even when those outputs are never returned. The conclusion concerns the two Lean models; interpreting it as source-level fidelity still relies on the curated reference and the semantic soundness of the relations.

D.3 Judge dependencies and the X/Y split

The primary semantic judgments S1–S5 gate C3, but the formal judges run independently of semantic outcomes. S5 reviews the evaluator's frozen R_I, R_O, shared by both X and Y, rather than C1 or C2 alone. X1–X5b use frozen C1, gold, and R_I, R_O, without inspecting the actual C2 predicates; Y1/Y2 additionally read frozen C2. Primed S1′–S5′ judgments independently replay the semantic checks on identical bytes and remain diagnostic rather than gating C3. A retained relation module can nevertheless import a C2 helper merely to elaborate; such an X job cannot be launched before that helper is frozen without changing the adapter. This is an operational dependency, not a C2 premise in the X theorem.

E Reproducibility and Artifact Release

Paired variants and targeted witnesses. For each task, the two variants share an intended contract and public API boundary but differ in source behavior. The correct variant is validated against the intent. The incorrect variant contains one deliberately retained or introduced behavior-affecting defect selected to be plausible in ordinary code and nontrivial to identify from surface syntax alone. Mutations include boundary-condition errors, state-update mistakes, closure capture, incorrect cache

behavior, data-structure invariant violations, and failures that depend on iteration order or nontermination.

The pair is accepted into the benchmark only after a targeted witness exercises the defect. The incorrect Python and incorrect gold Lean translation must agree on that witness, while the intended correct behavior differs. Random testing or a generic "bug found" label is insufficient. The correct-source package is tested on the same relevant boundary and receives its own proof of correctness; the incorrect-source package receives a certified contract counterexample.

Gold curation and release gates. Gold implementations are developed independently of evaluated submissions. Every released package must compile from a clean environment, pass a kernel and axiom audit, record frozen SHA-256 hashes, and include executable Python–Lean differential tests. Each final gold translation is reviewed on its final bytes by four independent Python-facing semantic audits: two Astra reviews and two Fable reviews. Stateful packages record their API boundary and observable state; fuelled packages include gold-side instances of Equations F1a–F1b, restricted to the gold precondition. Correct and incorrect packages are separate artifacts even when they share intent text. This prevents a repaired gold translation from being silently reused for a buggy source.

Isolation. Each evaluated-agent call receives a fresh, deny-by-default workspace with read-only, hash-checked inputs and access only to the Lean toolchain. Gold artifacts, reference proofs, other runs, user configuration, and shared temporary storage are inaccessible to the evaluated agent. Proof judges have separate workspaces containing their designated frozen inputs and the reference-side proof or counterexample. Codex calls have networking disabled; Opus permits only its model API. A machine-readable audit checks the model, target, dependencies, and visibility policy before every new or resumed call.

Release records and validation. The release records the benchmark revision, source provenance, Python and Lean versions, dependency snapshots, adapter code, targeted witnesses, prompts, model configurations, budgets, intermediate artifacts, judgments, proof-checker commands, root theorems, axiom audits, and SHA-256 hashes. A machine-readable manifest follows continuations and protocol migrations. Complete histories for gold authoring and evaluated runs are retained, including failed or superseded attempts.

Each official result is reconstructible from an immutable evidence receipt. The validation script checks lineage, authorized inputs, model identity, artifact hashes, compilation, kernel acceptance, axioms, and terminal status. Request identifiers separate gold authoring, agent calls, semantic judgments, and proof judgments in cost reconstruction. Host limits are execution controls, not Python semantics, unless the contract makes them observable.

F Official Evaluation Matrix

F.1 How to read our results

C3 requires both primary semantic acceptance (S1–S5) and successful formal comparison. The primed semantic replays are diagnostic; the matrix's X/Y columns are kernel-checked comparisons. Under the X/Y split, final success requires accepted C1 and C2 artifacts, primary semantic acceptance, every applicable X and Y obligation, any required F1 certificate, and a C3 proof for a correct source or C3 counterexample for an incorrect source. A retained C3 verdict on a row rejected by semantic or formal comparison does not count as final success. Em dashes are inapplicable, and missing or incomplete evidence is never counted as a pass. Costs cover only the reported retained usage.

F.2 Merged semantic and gold-centric evaluation

Table codes. C and I denote the correct and incorrect source variants. FIN means stage completed (C1/C2); AC, judge acceptance; SR, semantic rejection; P, machine-checked proof; CEX, machine-checked counterexample; ATL, agent token-limit endpoint; and BYE, certified F1 exemption. C3 records its completed result as P or CEX. An em dash marks an inapplicable or gated stage, or unavailable cost. A green check and red cross mark final success and failure. A dark-green outline

around SR marks a human-reviewed false negative (an incorrect rejection); a red outline around AC marks a human-reviewed false positive (an incorrect acceptance).

Table 4: Combined semantic and gold-centric evaluation. In the original 1,104-cell X/Y scope, 875 obligations are proved, 37 refuted, and 192 inapplicable; none remain unresolved. intervaltree contributes 32 additional X/Y verdicts outside that scope. Each submission uses two lines matching the upper and lower headers: implementation checks above, specification checks and final outcomes below.

Var.	Agent	C1 C2	S1 S4	S2 S5	S3 Y1	X1 Y2	X2 F1	X3 C3	X4 Proof $	X5a Final	X5b
IOI 2023 Beech Tree											
C	Astra	FIN	AC	AC	AC	P	P	P	P	P	P
		FIN	AC	AC	P	P	BYE	ATL	18.42	×	
C	Sol	FIN	AC	AC	AC	P	P	P	P	P	P
		FIN	AC	AC	P	P	BYE	ATL	19.09	×	
C	Opus	ATL	—	—	—	—	—	—	—	—	—
		—	—	—	—	—	—	—	—	×	
I	Astra	FIN	AC	AC	AC	P	P	P	P	P	P
		FIN	AC	AC	P	P	AC	CEX	28.38	✓	
I	Sol	FIN	AC	AC	AC	P	P	P	P	P	P
		FIN	AC	AC	P	P	AC	CEX	45.46	✓	
I	Opus	ATL	—	—	—	—	—	—	—	—	—
		—	—	—	—	—	—	—	—	×	
Vero Cachetools LRU											
C	Astra	FIN	AC	AC	AC	P	P	P	P	P	P
		FIN	AC	AC	P	P	BYE	P	9.37	✓	
C	Sol	FIN	AC	AC	AC	P	P	P	P	P	P
		FIN	AC	AC	P	P	BYE	ATL	5.56	×	
C	Opus	FIN	AC	AC	AC	P	P	P	P	P	P
		FIN	AC	AC	P	P	BYE	ATL	10.17	×	
I	Astra	FIN	AC	AC	AC	P	P	P	P	P	P
		FIN	AC	AC	P	P	BYE	CEX	6.89	✓	
I	Sol	FIN	AC	AC	AC	P	P	P	P	P	P
		FIN	AC	AC	P	P	BYE	CEX	9.20	✓	
I	Opus	FIN	AC	AC	AC	P	P	P	P	P	P
		FIN	SR	AC	P	CEX	—	—	26.26	×	
IOI 2022 Catfish											
C	Astra	FIN	AC	AC	AC	P	P	P	P	P	P
		FIN	AC	AC	P	P	BYE	ATL	16.90	×	
C	Sol	FIN	AC	AC	AC	P	P	P	P	P	P
		FIN	AC	AC	P	P	BYE	ATL	13.10	×	
C	Opus	FIN	AC	AC	AC	P	P	P	P	P	P
		FIN	AC	AC	P	P	BYE	ATL	17.95	×	
I	Astra	FIN	AC	AC	AC	P	P	P	P	P	P
		FIN	AC	AC	P	P	BYE	CEX	18.29	✓	
I	Sol	FIN	AC	AC	AC	P	P	P	P	P	P
		FIN	AC	AC	P	P	BYE	CEX	20.63	✓	
I	Opus	FIN	AC	AC	AC	P	P	P	P	P	P
		FIN	AC	AC	P	P	BYE	CEX	20.24	✓	

Continued on next page

Table 4 — continued

Var.	Agent	C1	S1	S2	S3	X1	X2	X3	X4	X5a	X5b
		C2	S4	S5	Y1	Y2	F1	C3	Proof $	Final	
Vero `difflib` blocks											
C	Astra	FIN	AC	AC	AC	P	P	P	P	P	P
		FIN	AC	AC	P	P	BYE	P	12.47	✓	
C	Sol	FIN	AC	AC	AC	P	P	P	P	P	P
		FIN	AC	AC	P	P	BYE	ATL	17.23	×	
C	Opus	FIN	AC	AC	AC	P	P	P	P	P	P
		FIN	AC	AC	P	P	BYE	ATL	30.54	×	
I	Astra	FIN	AC	AC	AC	P	P	P	P	P	P
		FIN	AC	AC	P	P	BYE	CEX	14.79	✓	
I	Sol	FIN	AC	AC	AC	P	P	P	P	P	P
		FIN	AC	AC	P	P	BYE	CEX	22.13	✓	
I	Opus	FIN	AC	AC	AC	P	P	P	P	P	P
		FIN	AC	AC	P	P	BYE	CEX	27.96	✓	
IOI 2021 Candies											
C	Astra	FIN	AC	AC	AC	P	P	P	P	P	P
		FIN	AC	AC	P	P	BYE	P	6.23	✓	
C	Sol	FIN	AC	AC	AC	P	P	P	P	P	P
		FIN	AC	AC	P	P	BYE	ATL	7.05	×	
C	Opus	FIN	AC	AC	AC	P	P	P	P	P	P
		FIN	AC	AC	P	P	BYE	ATL	6.15	×	
I	Astra	FIN	AC	AC	AC	P	P	P	P	P	P
		FIN	AC	AC	P	P	BYE	CEX	6.15	✓	
I	Sol	FIN	AC	AC	AC	P	P	P	P	P	P
		FIN	AC	AC	P	P	BYE	CEX	7.86	✓	
I	Opus	FIN	AC	AC	AC	P	P	P	P	P	P
		FIN	AC	AC	P	P	BYE	CEX	5.71	✓	
Vero NetworkX CC											
C	Astra	FIN	AC	AC	AC	P	P	P	P	P	P
		FIN	AC	AC	P	P	BYE	P	11.37	✓	
C	Sol	FIN	AC	AC	AC	P	P	P	P	P	P
		FIN	AC	AC	P	P	BYE	ATL	11.45	×	
C	Opus	FIN	AC	AC	AC	P	P	P	P	P	P
		FIN	AC	AC	P	P	BYE	ATL	26.18	×	
I	Astra	FIN	AC	AC	AC	P	P	P	P	P	P
		FIN	AC	AC	P	P	AC	CEX	11.57	✓	
I	Sol	FIN	SR	SR	AC	P	P	P	P	P	P
		FIN	AC	AC	P	P	—	—	16.03	×	
I	Opus	FIN	AC	AC	AC	P	P	P	P	P	P
		FIN	AC	AC	P	P	AC	CEX	30.03	✓	

Continued on next page

Table 4 — continued

Var.	Agent	C1	S1	S2	S3	X1	X2	X3	X4	X5a	X5b
		C2	S4	S5	Y1	Y2	F1	C3	Proof $	Final	
IOI 2020 Packing Biscuits											
C	Astra	FIN	AC	AC	AC	P	P	P	P	P	P
		FIN	AC	AC	P	P	BYE	P	6.67	✓	
C	Sol	FIN	AC	AC	AC	P	P	P	P	P	P
		FIN	AC	AC	P	P	BYE	ATL	4.80	×	
C	Opus	FIN	AC	AC	AC	P	P	P	P	P	P
		FIN	AC	AC	P	P	BYE	ATL	13.24	×	
I	Astra	FIN	AC	AC	AC	P	P	P	P	P	P
		FIN	AC	AC	P	P	BYE	CEX	8.30	✓	
I	Sol	FIN	AC	AC	AC	P	P	P	P	P	P
		FIN	AC	AC	P	P	BYE	CEX	10.08	✓	
I	Opus	FIN	AC	AC	AC	P	P	P	P	P	P
		FIN	AC	AC	P	P	BYE	CEX	15.32	✓	
Vero SortedContainers											
C	Astra	FIN	AC	AC	AC	P	P	P	P	P	P
		FIN	AC	AC	P	P	BYE	P	5.20	✓	
C	Sol	FIN	AC	AC	AC	P	P	P	P	P	P
		FIN	AC	AC	P	P	BYE	ATL	11.32	×	
C	Opus	FIN	AC	AC	AC	P	P	P	P	P	P
		FIN	AC	AC	P	P	BYE	ATL	18.98	×	
I	Astra	FIN	AC	AC	AC	P	P	P	P	P	P
		FIN	AC	AC	P	P	BYE	CEX	4.59	✓	
I	Sol	FIN	AC	SR	AC	CEX	P	P	P	P	P
		FIN	AC	AC	P	P	—	—	6.07	×	
I	Opus	ATL	—	—	—	—	—	—	—	—	—
		—	—	—	—	—	—	—	—	×	
IOI 2019 Split											
C	Astra	FIN	AC	AC	AC	P	P	P	P	P	P
		FIN	AC	AC	P	P	BYE	ATL	94.67	×	
C	Sol	FIN	AC	AC	AC	P	P	P	P	P	P
		FIN	AC	AC	P	P	BYE	ATL	92.47	×	
C	Opus	FIN	AC	AC	AC	P	P	P	P	P	P
		FIN	AC	AC	P	P	BYE	ATL	73.76	×	
I	Astra	FIN	AC	AC	AC	P	P	P	P	P	P
		FIN	AC	AC	P	P	BYE	CEX	78.20	✓	
I	Sol	FIN	SR	SR	AC	CEX	P	P	P	P	P
		FIN	AC	AC	P	P	—	—	13.67	×	
I	Opus	ATL	—	—	—	—	—	—	—	—	—
		—	—	—	—	—	—	—	—	×	

Continued on next page

Table 4 — continued

Var.	Agent	C1	S1	S2	S3	X1	X2	X3	X4	X5a	X5b
		C2	S4	S5	Y1	Y2	F1	C3	Proof $	Final	
IOI 2020 Supertrees											
C	Astra	FIN	AC	AC	AC	P	P	P	P	P	P
		FIN	AC	AC	P	P	BYE	ATL	15.56	×	
C	Sol	FIN	AC	AC	AC	P	P	P	P	P	P
		FIN	AC	AC	P	P	BYE	ATL	7.91	×	
C	Opus	ATL	—	—	—	—	—	—	—	—	—
		—	—	—	—	—	—	—	—	×	
I	Astra	FIN	AC	AC	AC	P	P	P	P	P	P
		FIN	AC	AC	P	P	BYE	CEX	21.73	✓	
I	Sol	FIN	SR	SR	AC	CEX	P	P	P	P	P
		FIN	AC	AC	P	P	—	—	16.62	×	
I	Opus	ATL	—	—	—	—	—	—	—	—	—
		—	—	—	—	—	—	—	—	×	
IOI 2022 Radio Towers											
C	Astra	FIN	AC	AC	AC	P	P	P	P	P	P
		FIN	AC	AC	P	P	BYE	ATL	56.45	×	
C	Sol	FIN	AC	AC	AC	P	P	P	P	P	P
		FIN	AC	AC	P	P	BYE	ATL	37.20	×	
C	Opus	ATL	—	—	—	—	—	—	—	—	—
		—	—	—	—	—	—	—	—	×	
I	Astra	FIN	AC	AC	AC	P	P	P	P	P	P
		FIN	AC	AC	P	P	BYE	CEX	45.99	✓	
I	Sol	FIN	AC	SR	AC	CEX	P	P	P	P	P
		FIN	AC	AC	P	P	—	—	24.95	×	
I	Opus	ATL	—	—	—	—	—	—	—	—	—
		—	—	—	—	—	—	—	—	×	
IOI 2023 Closing Time											
C	Astra	FIN	AC	AC	AC	P	P	P	P	P	P
		FIN	AC	AC	P	P	BYE	ATL	47.50	×	
C	Sol	FIN	AC	AC	AC	P	P	P	P	P	P
		FIN	AC	AC	P	P	BYE	ATL	80.40	×	
C	Opus	ATL	—	—	—	—	—	—	—	—	—
		—	—	—	—	—	—	—	—	×	
I	Astra	FIN	AC	AC	AC	P	P	P	P	P	P
		FIN	AC	AC	P	P	BYE	ATL	70.90	×	
I	Sol	FIN	AC	AC	AC	P	P	P	P	P	P
		FIN	AC	AC	P	P	BYE	ATL	58.19	×	
I	Opus	ATL	—	—	—	—	—	—	—	—	—
		—	—	—	—	—	—	—	—	×	

Continued on next page

Table 4 — continued

Var.	Agent	C1	S1	S2	S3	X1	X2	X3	X4	X5a	X5b
		C2	S4	S5	Y1	Y2	F1	C3	Proof $	Final	
IOI 2023 Soccer Stadium											
C	Astra	FIN	AC	AC	AC	P	P	P	P	P	P
		FIN	AC	AC	P	P	BYE	ATL	76.65	×	
C	Sol	FIN	AC	AC	AC	P	P	P	P	P	P
		FIN	AC	AC	P	P	BYE	ATL	69.05	×	
C	Opus	ATL	—	—	—	—	—	—	—	—	—
		—	—	—	—	—	—	—	—	×	
I	Astra	FIN	AC	AC	AC	P	P	P	P	P	P
		FIN	AC	AC	P	P	BYE	CEX	59.58	✓	
I	Sol	FIN	SR	SR	AC	CEX	P	P	P	P	P
		FIN	AC	AC	P	P	—	—	47.76	×	
I	Opus	ATL	—	—	—	—	—	—	—	—	—
		—	—	—	—	—	—	—	—	×	
IOI 2024 Hieroglyphs											
C	Astra	FIN	AC	AC	AC	P	P	P	P	P	P
		FIN	AC	AC	P	P	BYE	ATL	82.77	×	
C	Sol	FIN	AC	AC	AC	P	P	P	P	P	P
		FIN	AC	AC	P	P	BYE	ATL	49.52	×	
C	Opus	FIN	AC	AC	AC	P	P	P	P	P	P
		FIN	AC	AC	P	P	BYE	ATL	54.14	×	
I	Astra	FIN	AC	AC	AC	P	P	P	P	P	P
		FIN	AC	AC	P	P	BYE	CEX	37.64	✓	
I	Sol	FIN	AC	AC	AC	P	P	P	P	P	P
		FIN	AC	AC	P	P	BYE	CEX	65.16	✓	
I	Opus	FIN	AC	AC	AC	P	P	P	P	P	P
		FIN	AC	AC	P	P	BYE	CEX	33.08	✓	
IOI 2025 Festival											
C	Astra	FIN	AC	AC	AC	P	P	P	P	P	P
		FIN	AC	AC	P	P	BYE	ATL	45.04	×	
C	Sol	FIN	AC	AC	AC	P	P	P	P	P	P
		FIN	AC	AC	P	P	BYE	ATL	38.05	×	
C	Opus	FIN	AC	AC	AC	P	P	P	P	P	P
		FIN	AC	AC	P	P	BYE	ATL	26.30	×	
I	Astra	FIN	AC	AC	AC	P	P	P	P	P	P
		FIN	AC	AC	P	P	BYE	CEX	30.29	✓	
I	Sol	FIN	SR	SR	AC	CEX	P	P	P	P	P
		FIN	AC	AC	P	P	—	—	14.70	×	
I	Opus	ATL	—	—	—	—	—	—	—	—	—
		—	—	—	—	—	—	—	—	×	

Continued on next page

Table 4 — continued

Var.	Agent	C1	S1	S2	S3	X1	X2	X3	X4	X5a	X5b
		C2	S4	S5	Y1	Y2	F1	C3	Proof $	Final	
IOI 2025 Obstacles											
C	Astra	FIN	AC	AC	AC	P	P	P	P	P	P
		FIN	AC	AC	P	P	BYE	ATL	29.44	×	
C	Sol	FIN	AC	AC	AC	P	P	P	P	P	P
		FIN	AC	AC	P	P	BYE	ATL	26.19	×	
C	Opus	ATL	—	—	—	—	—	—	—	—	—
		—	—	—	—	—	—	—	—	×	
I	Astra	FIN	AC	AC	AC	P	P	P	P	P	P
		FIN	AC	AC	P	P	BYE	CEX	50.98	✓	
I	Sol	FIN	SR	SR	AC	CEX	P	P	P	P	P
		FIN	AC	AC	P	P	—	—	27.33	×	
I	Opus	ATL	—	—	—	—	—	—	—	—	—
		—	—	—	—	—	—	—	—	×	
Command Profile											
C	Astra	FIN	AC	AC	AC	P	P	P	P	P	P
		FIN	AC	AC	P	P	BYE	ATL	73.01	×	
C	Sol	FIN	SR	SR	AC	CEX	P	CEX	P	P	P
		FIN	SR	SR	P	CEX	—	—	15.44	×	
C	Opus	ATL	—	—	—	—	—	—	—	—	—
		—	—	—	—	—	—	—	—	×	
I	Astra	FIN	AC	AC	AC	P	P	P	P	P	P
		FIN	AC	AC	P	P	BYE	CEX	83.30	✓	
I	Sol	FIN	SR	SR	AC	CEX	P	CEX	P	P	P
		FIN	SR	SR	P	CEX	—	—	24.80	×	
I	Opus	ATL	—	—	—	—	—	—	—	—	—
		—	—	—	—	—	—	—	—	×	
Path Itinerary											
C	Astra	FIN	AC	AC	AC	P	P	P	P	P	P
		FIN	AC	AC	P	P	AC	ATL	54.23	×	
C	Sol	FIN	SR	SR	AC	CEX	P	P	P	P	P
		FIN	SR	AC	P	CEX	—	—	42.24	×	
C	Opus	ATL	—	—	—	—	—	—	—	—	—
		—	—	—	—	—	—	—	—	×	
I	Astra	FIN	AC	AC	AC	P	P	P	P	P	P
		FIN	AC	AC	P	P	BYE	CEX	106.85	✓	
I	Sol	FIN	AC	AC	AC	P	P	P	P	P	P
		FIN	SR	AC	P	CEX	—	—	78.31	×	
I	Opus	FIN	AC	AC	AC	P	P	P	P	P	P
		FIN	AC	AC	P	P	BYE	CEX	57.81	✓	

Continued on next page

Table 4 — continued

Var.	Agent	C1	S1	S2	S3	X1	X2	X3	X4	X5a	X5b
		C2	S4	S5	Y1	Y2	F1	C3	Proof $	Final	
Schema Values											
C	Astra	FIN	AC	AC	AC	P	P	P	P	P	P
		FIN	AC	AC	P	P	BYE	ATL	26.84	×	
C	Sol	FIN	SR	SR	AC	CEX	P	P	P	P	P
		FIN	AC	AC	P	P	—	—	20.90	×	
C	Opus	FIN	SR	SR	AC	CEX	P	P	P	P	P
		FIN	AC	AC	P	P	—	—	18.92	×	
I	Astra	FIN	AC	AC	AC	P	P	P	P	P	P
		FIN	SR	AC	P	CEX	—	—	28.11	×	
I	Sol	FIN	SR	SR	AC	P	P	P	P	P	P
		FIN	AC	AC	P	P	BYE	CEX	40.27	×	
I	Opus	FIN	SR	SR	AC	P	P	P	P	P	P
		FIN	AC	AC	P	P	BYE	CEX	38.47	×	
Manifest Tokens											
C	Astra	FIN	AC	AC	AC	P	P	P	P	P	P
		FIN	AC	AC	P	P	AC	ATL	39.16	×	
C	Sol	FIN	SR	SR	AC	CEX	P	P	P	P	P
		FIN	AC	AC	P	P	—	—	22.19	×	
C	Opus	FIN	AC	AC	AC	P	P	P	P	P	P
		FIN	SR	AC	P	P	—	—	92.17	×	
I	Astra	FIN	AC	AC	AC	P	P	P	P	P	P
		FIN	AC	AC	P	P	AC	CEX	28.29	✓	
I	Sol	FIN	AC	AC	AC	P	P	P	P	P	P
		FIN	SR	AC	P	CEX	—	—	59.76	×	
I	Opus	ATL	—	—	—	—	—	—	—	—	—
		—	—	—	—	—	—	—	—	×	
Markup Segments											
C	Astra	FIN	SR	SR	AC	CEX	P	P	P	P	P
		FIN	SR	AC	P	CEX	—	—	93.48	×	
C	Sol	FIN	SR	SR	AC	CEX	P	P	P	P	P
		FIN	SR	SR	CEX	CEX	—	—	43.76	×	
C	Opus	ATL	—	—	—	—	—	—	—	—	—
		—	—	—	—	—	—	—	—	×	
I	Astra	FIN	AC	AC	AC	P	P	P	P	P	P
		FIN	AC	AC	P	P	BYE	CEX	118.29	✓	
I	Sol	FIN	SR	SR	AC	CEX	P	P	P	P	P
		FIN	SR	SR	CEX	CEX	—	—	53.27	×	
I	Opus	ATL	—	—	—	—	—	—	—	—	—
		—	—	—	—	—	—	—	—	×	

Continued on next page

Table 4 — continued

Var.	Agent	C1 C2	S1 S4	S2 S5	S3 Y1	X1 Y2	X2 F1	X3 C3	X4 Proof $	X5a Final	X5b
Prefix Records											
C	Astra	FIN	AC	AC	AC	P	P	P	P	P	P
		FIN	AC	AC	P	P	BYE	ATL	48.13	×	
C	Sol	FIN	SR	SR	AC	CEX	P	P	P	P	P
		FIN	AC	AC	P	P	—	—	11.27	×	
C	Opus	FIN	AC	AC	AC	P	P	P	P	P	P
		FIN	AC	AC	P	P	BYE	ATL	23.52	×	
I	Astra	FIN	AC	AC	AC	P	P	P	P	P	P
		FIN	AC	AC	P	P	BYE	CEX	26.39	✓	
I	Sol	FIN	SR	SR	AC	CEX	P	P	P	P	P
		FIN	AC	AC	P	P	—	—	22.13	×	
I	Opus	ATL	—	—	—	—	—	—	—	—	—
		—	—	—	—	—	—	—	—	×	
Vero Textwrap											
C	Astra	FIN	AC	AC	AC	P	P	P	P	P	P
		FIN	AC	AC	P	P	AC	ATL	71.88	×	
C	Sol	FIN	SR	SR	SR	CEX	P	P	CEX	P	P
		FIN	SR	SR	P	CEX	—	—	26.82	×	
C	Opus	ATL	—	—	—	—	—	—	—	—	—
		—	—	—	—	—	—	—	—	×	
I	Astra	FIN	AC	AC	AC	P	P	P	P	P	P
		FIN	AC	AC	P	P	AC	CEX	74.31	✓	
I	Sol	FIN	SR	SR	SR	P	P	P	CEX	P	P
		FIN	SR	SR	P	CEX	—	—	60.95	×	
I	Opus	ATL	—	—	—	—	—	—	—	—	—
		—	—	—	—	—	—	—	—	×	
Vero intervaltree											
C	Astra	FIN	AC	AC	AC	P	P	P	P	P	P
		FIN	AC	AC	P	P	AC	ATL	77.90	×	
C	Sol	FIN	SR	SR	AC	P	P	P	P	P	P
		FIN	AC	AC	P	P	AC	ATL	13.26	×	
C	Opus	ATL	—	—	—	—	—	—	—	—	—
		—	—	—	—	—	—	—	—	×	
I	Astra	FIN	AC	SR	AC	CEX	P	P	P	P	P
		FIN	AC	AC	P	P	—	—	35.58	×	
I	Sol	FIN	SR	SR	AC	CEX	P	P	P	P	P
		FIN	AC	AC	P	P	—	—	10.07	×	
I	Opus	ATL	—	—	—	—	—	—	—	—	—
		—	—	—	—	—	—	—	—	×	

F.3 Astra and Terra semantic judgments

Original Astra S1–S5 judgments are repeated from the main matrix beside Terra's primed semantic replay judgments on frozen evaluated-agent artifacts. The Terra columns are not Terra agent submissions. Cell fills and human-reviewed false-negative/false-positive outlines follow the main-matrix convention.

Table 5: Side-by-side Astra and Terra semantic judgments, kept separate from the main S/X–Y matrix. NR means no recorded judgment.

		Astra					Terra replay				
Var.	Agent	S1	S2	S3	S4	S5	S1′	S2′	S3′	S4′	S5′
IOI 2023 Beech Tree											
C	Astra	AC	AC	AC	AC	AC	AC	AC	AC	AC	AC
C	Sol	AC	AC	AC	AC	AC	AC	AC	AC	AC	AC
C	Opus	—	—	—	—	—	—	—	—	—	—
I	Astra	AC	AC	AC	AC	AC	AC	AC	AC	AC	AC
I	Sol	AC	AC	AC	AC	AC	SR	AC	AC	AC	AC
I	Opus	—	—	—	—	—	—	—	—	—	—
Vero Cachetools LRU											
C	Astra	AC	AC	AC	AC	AC	AC	AC	AC	AC	AC
C	Sol	AC	AC	AC	AC	AC	AC	AC	AC	AC	AC
C	Opus	AC	AC	AC	AC	AC	AC	AC	AC	AC	AC
I	Astra	AC	AC	AC	AC	AC	AC	AC	AC	AC	AC
I	Sol	AC	AC	AC	AC	AC	AC	AC	AC	AC	AC
I	Opus	AC	AC	AC	SR	AC	AC	AC	AC	SR	AC
IOI 2022 Catfish											
C	Astra	AC	AC	AC	AC	AC	AC	AC	AC	AC	AC
C	Sol	AC	AC	AC	AC	AC	AC	AC	AC	AC	AC
C	Opus	AC	AC	AC	AC	AC	AC	AC	AC	AC	AC
I	Astra	AC	AC	AC	AC	AC	AC	AC	AC	AC	AC
I	Sol	AC	AC	AC	AC	AC	SR	AC	AC	AC	AC
I	Opus	AC	AC	AC	AC	AC	AC	AC	AC	AC	AC
Vero `difflib` blocks											
C	Astra	AC	AC	AC	AC	AC	AC	AC	AC	SR	AC
C	Sol	AC	AC	AC	AC	AC	AC	AC	AC	AC	AC
C	Opus	AC	AC	AC	AC	AC	AC	AC	AC	AC	AC
I	Astra	AC	AC	AC	AC	AC	AC	AC	AC	AC	AC
I	Sol	AC	AC	AC	AC	AC	AC	AC	AC	AC	AC
I	Opus	AC	AC	AC	AC	AC	AC	AC	AC	AC	AC
IOI 2021 Candies											
C	Astra	AC	AC	AC	AC	AC	AC	AC	AC	AC	AC
C	Sol	AC	AC	AC	AC	AC	AC	AC	AC	AC	AC
C	Opus	AC	AC	AC	AC	AC	AC	AC	AC	AC	AC
I	Astra	AC	AC	AC	AC	AC	AC	AC	AC	AC	AC
I	Sol	AC	AC	AC	AC	AC	AC	AC	AC	AC	AC
I	Opus	AC	AC	AC	AC	AC	AC	AC	AC	AC	AC
Vero NetworkX CC											
C	Astra	AC	AC	AC	AC	AC	AC	AC	AC	AC	AC
C	Sol	AC	AC	AC	AC	AC	AC	AC	AC	AC	AC
C	Opus	AC	AC	AC	AC	AC	AC	AC	AC	AC	AC
I	Astra	AC	AC	AC	AC	AC	AC	AC	AC	AC	AC
I	Sol	SR	SR	AC	AC	AC	AC	AC	AC	AC	AC
I	Opus	AC	AC	AC	AC	AC	AC	AC	AC	AC	AC
IOI 2020 Packing Biscuits											
C	Astra	AC	AC	AC	AC	AC	AC	AC	AC	AC	AC
C	Sol	AC	AC	AC	AC	AC	AC	AC	AC	AC	AC
C	Opus	AC	AC	AC	AC	AC	AC	AC	AC	AC	AC
I	Astra	AC	AC	AC	AC	AC	AC	AC	AC	AC	AC
I	Sol	AC	AC	AC	AC	AC	AC	AC	AC	AC	AC
I	Opus	AC	AC	AC	AC	AC	AC	AC	AC	AC	AC

Continued on next page

Table 5 — continued

		Astra					Terra replay				
Var.	Agent	S1	S2	S3	S4	S5	S1′	S2′	S3′	S4′	S5′
Vero SortedContainers											
C	Astra	AC	AC	AC	AC	AC	AC	AC	AC	AC	AC
C	Sol	AC	AC	AC	AC	AC	AC	AC	AC	AC	AC
C	Opus	AC	AC	AC	AC	AC	AC	AC	AC	AC	AC
I	Astra	AC	AC	AC	AC	AC	AC	AC	AC	AC	AC
I	Sol	AC	SR	AC	AC	AC	AC	SR	AC	AC	AC
I	Opus	—	—	—	—	—	—	—	—	—	—
IOI 2019 Split											
C	Astra	AC	AC	AC	AC	AC	AC	AC	AC	AC	AC
C	Sol	AC	AC	AC	AC	AC	AC	AC	AC	AC	AC
C	Opus	AC	AC	AC	AC	AC	AC	AC	AC	AC	AC
I	Astra	AC	AC	AC	AC	AC	AC	AC	AC	AC	AC
I	Sol	SR	SR	AC	AC	AC	AC	AC	AC	AC	AC
I	Opus	—	—	—	—	—	—	—	—	—	—
IOI 2020 Supertrees											
C	Astra	AC	AC	AC	AC	AC	AC	AC	AC	AC	AC
C	Sol	AC	AC	AC	AC	AC	AC	AC	AC	AC	AC
C	Opus	—	—	—	—	—	—	—	—	—	—
I	Astra	AC	AC	AC	AC	AC	AC	AC	AC	AC	AC
I	Sol	SR	SR	AC	AC	AC	AC	SR	AC	AC	AC
I	Opus	—	—	—	—	—	—	—	—	—	—
IOI 2022 Radio Towers											
C	Astra	AC	AC	AC	AC	AC	AC	AC	AC	AC	AC
C	Sol	AC	AC	AC	AC	AC	AC	AC	AC	AC	AC
C	Opus	—	—	—	—	—	—	—	—	—	—
I	Astra	AC	AC	AC	AC	AC	SR	AC	AC	AC	AC
I	Sol	AC	SR	AC	AC	AC	AC	SR	AC	AC	AC
I	Opus	—	—	—	—	—	—	—	—	—	—
IOI 2023 Closing Time											
C	Astra	AC	AC	AC	AC	AC	AC	AC	AC	AC	AC
C	Sol	AC	AC	AC	AC	AC	AC	AC	AC	AC	AC
C	Opus	—	—	—	—	—	—	—	—	—	—
I	Astra	AC	AC	AC	AC	AC	AC	AC	AC	AC	AC
I	Sol	AC	AC	AC	AC	AC	AC	AC	AC	AC	AC
I	Opus	—	—	—	—	—	—	—	—	—	—
IOI 2023 Soccer Stadium											
C	Astra	AC	AC	AC	AC	AC	AC	AC	AC	AC	AC
C	Sol	AC	AC	AC	AC	AC	AC	AC	AC	AC	AC
C	Opus	—	—	—	—	—	—	—	—	—	—
I	Astra	AC	AC	AC	AC	AC	AC	AC	AC	AC	AC
I	Sol	SR	SR	AC	AC	AC	AC	SR	AC	AC	AC
I	Opus	—	—	—	—	—	—	—	—	—	—
IOI 2024 Hieroglyphs											
C	Astra	AC	AC	AC	AC	AC	AC	AC	AC	AC	AC
C	Sol	AC	AC	AC	AC	AC	AC	AC	AC	AC	AC
C	Opus	AC	AC	AC	AC	AC	AC	AC	AC	AC	AC
I	Astra	AC	AC	AC	AC	AC	AC	AC	AC	AC	AC
I	Sol	AC	AC	AC	AC	AC	AC	AC	AC	AC	AC
I	Opus	AC	AC	AC	AC	AC	AC	AC	AC	AC	AC

Continued on next page

Table 5 — continued

		Astra					Terra replay				
Var.	Agent	S1	S2	S3	S4	S5	S1′	S2′	S3′	S4′	S5′
IOI 2025 Festival											
C	Astra	AC	AC	AC	AC	AC	AC	AC	AC	AC	AC
C	Sol	AC	AC	AC	AC	AC	AC	AC	AC	AC	AC
C	Opus	AC	AC	AC	AC	AC	AC	AC	AC	AC	AC
I	Astra	AC	AC	AC	AC	AC	SR	AC	AC	AC	AC
I	Sol	SR	SR	AC	AC	AC	AC	SR	AC	AC	AC
I	Opus	—	—	—	—	—	—	—	—	—	—
IOI 2025 Obstacles											
C	Astra	AC	AC	AC	AC	AC	AC	AC	AC	AC	AC
C	Sol	AC	AC	AC	AC	AC	AC	AC	AC	AC	AC
C	Opus	—	—	—	—	—	—	—	—	—	—
I	Astra	AC	AC	AC	AC	AC	AC	AC	AC	AC	AC
I	Sol	SR	SR	AC	AC	AC	AC	SR	AC	AC	AC
I	Opus	—	—	—	—	—	—	—	—	—	—
Command Profile											
C	Astra	AC	AC	AC	AC	AC	SR	AC	AC	AC	AC
C	Sol	SR	SR	AC	SR	SR	AC	SR	AC	SR	SR
C	Opus	—	—	—	—	—	—	—	—	—	—
I	Astra	AC	AC	AC	AC	AC	AC	AC	AC	AC	AC
I	Sol	SR	SR	AC	SR	SR	AC	SR	AC	SR	SR
I	Opus	—	—	—	—	—	—	—	—	—	—
Path Itinerary											
C	Astra	AC	AC	AC	AC	AC	AC	SR	AC	AC	AC
C	Sol	SR	SR	AC	SR	AC	AC	SR	AC	SR	AC
C	Opus	—	—	—	—	—	—	—	—	—	—
I	Astra	AC	AC	AC	AC	AC	SR	AC	AC	AC	AC
I	Sol	AC	AC	AC	SR	AC	AC	AC	AC	AC	AC
I	Opus	AC	AC	AC	AC	AC	AC	AC	AC	AC	AC
Schema Values											
C	Astra	AC	AC	AC	AC	AC	AC	AC	AC	AC	AC
C	Sol	SR	SR	AC	AC	AC	AC	SR	AC	AC	AC
C	Opus	SR	SR	AC	AC	AC	AC	SR	AC	AC	AC
I	Astra	AC	AC	AC	SR	AC	AC	AC	AC	SR	AC
I	Sol	SR	SR	AC	AC	AC	AC	SR	AC	AC	AC
I	Opus	SR	SR	AC	AC	AC	AC	SR	AC	AC	AC
Manifest Tokens											
C	Astra	AC	AC	AC	AC	AC	AC	AC	AC	AC	AC
C	Sol	SR	SR	AC	AC	AC	AC	SR	AC	AC	AC
C	Opus	AC	AC	AC	SR	AC	AC	AC	AC	SR	AC
I	Astra	AC	AC	AC	AC	AC	AC	AC	AC	AC	AC
I	Sol	AC	AC	AC	SR	AC	AC	AC	AC	SR	AC
I	Opus	—	—	—	—	—	—	—	—	—	—
Markup Segments											
C	Astra	SR	SR	AC	SR	AC	AC	SR	AC	AC	AC
C	Sol	SR	SR	AC	SR	SR	AC	SR	AC	SR	AC
C	Opus	—	—	—	—	—	—	—	—	—	—
I	Astra	AC	AC	AC	AC	AC	AC	AC	AC	AC	AC
I	Sol	SR	SR	AC	SR	SR	AC	SR	AC	SR	AC
I	Opus	—	—	—	—	—	—	—	—	—	—

Continued on next page

Table 5 — continued

		Astra					Terra replay				
Var.	Agent	S1	S2	S3	S4	S5	S1′	S2′	S3′	S4′	S5′
Prefix Records											
C	Astra	AC	AC	AC	AC	AC	AC	AC	AC	AC	AC
C	Sol	SR	SR	AC	AC	AC	AC	SR	AC	AC	AC
C	Opus	AC	AC	AC	AC	AC	AC	AC	AC	AC	AC
I	Astra	AC	AC	AC	AC	AC	AC	AC	AC	AC	AC
I	Sol	SR	SR	AC	AC	AC	AC	SR	AC	AC	AC
I	Opus	—	—	—	—	—	—	—	—	—	—
Vero Textwrap											
C	Astra	AC	AC	AC	AC	AC	AC	AC	AC	AC	AC
C	Sol	SR	SR	SR	SR	SR	SR	SR	SR	SR	SR
C	Opus	—	—	—	—	—	—	—	—	—	—
I	Astra	AC	AC	AC	AC	AC	AC	AC	AC	AC	AC
I	Sol	SR	SR	SR	SR	SR	SR	SR	SR	SR	SR
I	Opus	—	—	—	—	—	—	—	—	—	—
Vero intervaltree											
C	Astra	AC	AC	AC	AC	AC	AC	AC	AC	AC	AC
C	Sol	SR	SR	AC	AC	AC	AC	SR	AC	AC	AC
C	Opus	—	—	—	—	—	—	—	—	—	—
I	Astra	AC	SR	AC	AC	AC	AC	SR	AC	AC	AC
I	Sol	SR	SR	AC	AC	AC	SR	SR	AC	AC	AC
I	Opus	—	—	—	—	—	—	—	—	—	—