TRANSFORMAL: TOWARDS SEMANTICALLY FAITHFUL FORMAL REPOSITORY TRANSLATION IN THE WILD

Anonymous authors
Paper under double-blind review

ABSTRACT

Large verified developments are scattered across proof assistants with different logics, such as Coq, Isabelle, and Lean. *Formal repository translation* enables their reuse across systems, but ensuring that translations preserve meaning remains an open problem. Even checking this *translation faithfulness* is difficult, because it requires relating declarations across two logics. Most existing faithfulness metrics check a statement against a reference or an informal description within a single logic, so they do not directly apply to translation. To bridge this gap, we propose *counterexample-based faithfulness* (CEF), a reference-free metric that reduces faithfulness checking to counterexample finding. A counterexample consists of source and target inputs that semantically represent the same object but lead to incompatible observations. To make counterexample search more reliable and efficient, we build TRANSFORMAL, an LLM agent supported by four stages of program-analysis tools. The stages extract dependencies and elaborated signatures from the compilers, derive candidate inputs from branch conditions, and replay each candidate in both proof assistants. To evaluate TRANSFORMAL, we construct TF-JUDGE, a benchmark of 297 expert-labeled declaration pairs from 18 real-world Coq and Isabelle repositories. With GPT-5.6 Sol, TRANSFORMAL reaches 94.28% accuracy, compared with 59.93% for the same model judging directly. The counterexamples of TRANSFORMAL also provide precise feedback for translation, because each one shows where a translation diverges. On our translation benchmark TF-TRANSLATION, this feedback raises the share of GPT-5.6 Sol's declarations that satisfy CEF from 80.25% to 92.48%, four times the gain of compiler feedback. Counterexamples thus make translation faithfulness both measurable and repairable, and TRANSFORMAL finds them more reliably and efficiently than baselines. Our code and data are available at `https://anonymous.4open.science/r/transformal`.

1 INTRODUCTION

Formal verification has produced large developments of verified software and formalized mathematics, such as the seL4 kernel (Klein et al., 2009) and the Archive of Formal Proofs (Blanchette et al., 2015) in Isabelle, the CompCert compiler (Leroy, 2009) and the Flocq floating-point library (Boldo & Melquiond, 2011) in Coq, and Mathlib (The mathlib Community, 2020) in Lean. These developments are scattered across systems with different logics, so a result proved in one system cannot be reused in another without translation. Early tools translated proofs through importers for one pair of systems (Obua & Skalberg, 2006; Keller & Werner, 2010; Kaliszyk & Krauss, 2013) or through a shared logical framework (Thiré, 2018), but each works only for the systems it was built for.

LLM agents lift this restriction, since they translate statements and proofs between many pairs of proof assistants (Viennot et al., 2025; Stoskopf et al., 2025; Wu et al., 2026; Liu et al., 2026), and recent projects have used them to port whole repositories to Lean (Saraswat et al., 2026; Beneficial AI Foundation, 2025). *Formal repository translation* asks an agent to translate a verified repository from one proof assistant into another. In this work, we focus on translating Coq and Isabelle repositories into Lean, because much recent AI work on formal verification builds on Lean (Ye et al., 2026b; Thakur et al., 2025; Xin et al., 2026; Ye et al., 2026a), whose main library focuses on mathematics (The mathlib Community, 2020). Translation brings the verified code and computer-science theory developed in Coq and Isabelle (Blanchette et al., 2015) to these AI developments.

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.

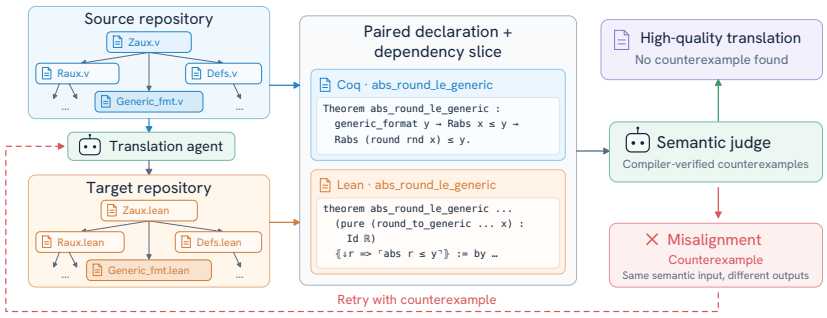

Figure 1: **Overview of TRANSFORMAL.** A translation agent translates a verified source repository into a target repository that preserves its dependency structure. For each translated declaration, TRANSFORMAL searches for a compiler-verified counterexample using the dependency slices of the paired declarations. A counterexample returns to the agent as feedback, and a translation is accepted when no counterexample is found and a replayed example agrees on both sides.

The main difficulty of formal repository translation is *faithfulness*, which requires each target declaration to preserve the behavior and the propositions of its source declaration. Because the two logics differ, the agent must decide how to represent each source concept in the target, and a wrong decision can change the meaning of every declaration that depends on it. Detecting such errors is also hard, because it requires relating declarations across two logics through their dependencies.

Existing methods check faithfulness only in restricted settings. Reference-based metrics compare statements within a single proof assistant (Liu et al., 2025; Poiroux et al., 2025), certified translation needs a bridge that imports the target back into the source system (Saraswat et al., 2026), and LLM judges give verdicts that nothing validates. This leaves an open question: *How can we judge and improve agent performance on formal repository translation efficiently and accurately?*

We propose *counterexample-based faithfulness* (CEF), a reference-free metric that reduces faithfulness checking to counterexample finding (Section 3). A declaration pair is unfaithful exactly when it has a counterexample, in which source and target inputs represent the same value but lead to incompatible observations. Each side of a counterexample is checked in its own proof assistant, so no formal bridge between the two logics is needed. To find counterexamples reliably and efficiently, we build TRANSFORMAL (Figure 1). In TRANSFORMAL, an LLM agent proposes candidates, and four stages of program-analysis tools supply the dependency context, elaborated signatures, and symbolic branch conditions of each pair and replay each candidate in both proof assistants. Each counterexample also shows the translation agent where its translation diverges, so TRANSFORMAL can guide repairs.

To evaluate TRANSFORMAL, we build two benchmarks from real-world Coq and Isabelle repositories. The faithfulness judging benchmark TF-JUDGE contains 297 source–target declaration pairs, each labeled faithful or unfaithful by formal-methods experts. The repository translation benchmark TF-TRANSLATION contains 4 repositories with 1,185 source declarations to be translated into Lean. On TF-JUDGE, TRANSFORMAL with GPT-5.6 Sol judges 94.28% of the pairs correctly, compared with 59.93% when the same model judges them directly (Table 2). On TF-TRANSLATION, three rounds of TRANSFORMAL feedback raise the share of GPT-5.6 Sol's declarations that satisfy CEF from 80.25% to 92.48%, four times the gain of compiler feedback (Figure 3). CEF and TRANSFORMAL thus provide formal repository translation with an accurate, efficient, and reference-free faithfulness signal that both evaluates translation agents and guides their repairs.

Contributions. In summary, we make the following key contributions:

- **A reference-free metric for translation faithfulness.** We propose CEF, which reduces faithfulness checking to counterexample finding and validates each counterexample in its own proof assistant, so faithfulness can be checked across logics without reference translations or formal bridges.
- **An agentic framework for counterexample search.** We build TRANSFORMAL, which guides an LLM agent with compiler-derived program analysis and finds counterexamples more reliably and at lower cost than direct LLM judgment.
- **Benchmarks and evaluation for translation faithfulness.** We construct TF-JUDGE and TF-TRANSLATION from real-world Coq and Isabelle repositories, and show that TRANSFORMAL

judges faithfulness accurately and that its counterexamples improve agent translations, providing a testbed for progress on formal repository translation.

2 Background and Related Work

Evaluating the faithfulness of autoformalization. Checking that a formal statement means what was intended is a central problem in autoformalization. Early work judged correctness by expert review and used overlap with reference formalizations only as a proxy (Wu et al., 2022; Azerbayev et al., 2023). BEq and BEq+ instead prove a generated statement equivalent to a trusted reference in the same proof assistant (Liu et al., 2025; Poiroux et al., 2025), and ShadowBench proves that it implies and is implied by auxiliary theorems derived from such a reference (Han et al., 2026). Other methods check a statement against its natural-language source, by testing whether autoformalization preserves the validity of reasoning steps (Cornish et al., 2026), by testing the generated specification against each atomic requirement (Ye et al., 2026c), or by asking an LLM judge (Leang et al., 2026). All of these methods relate a formal statement to an informal description or to a reference within a single logic. In contrast, CEF relates two formal declarations in different logics, and each counterexample is validated in both proof assistants.

Scaling formal construction. Work on LLM-based verification has grown from standalone functions (Ye et al., 2026b; Thakur et al., 2025) to data-structure modules (Sun et al., 2026), library tasks drawn from real repositories (Li et al., 2026b), and multi-module repositories (Xin et al., 2026; Ye et al., 2026a). Other systems formalize a textbook (Feng et al., 2026) or research papers in quantum computation (Ren et al., 2026) as Lean libraries, or synthesize specifications (Wang et al., 2025) and TLA+ models (Cheng et al., 2026) from program code. These works show that agents can build large formal artifacts, but they judge the meaning of what they build against curated specifications, informal sources, the original code, or human review. Formal repository translation instead starts from an existing formal repository, whose declarations fix the intended meaning of the target, and CEF uses this source to check faithfulness.

Translation between proof assistants. Early tools translated proofs through importers for specific pairs of systems (Obua & Skalberg, 2006; Keller & Werner, 2010; Kaliszyk & Krauss, 2013) or through a shared logical framework (Thiré, 2018; Blanqui, 2024). LLMs now translate statements (Viennot et al., 2025) and proofs (Stoskopf et al., 2025; Liu et al., 2026) between many pairs of systems, and ITPEval benchmarks both across Lean, Coq, Isabelle, and HOL Light (Wu et al., 2026). These studies focus on isolated statements and proofs, and evaluate them by type checking, by equivalence to a reference statement, or by expert inspection. lf-lean certifies each translated statement of a whole textbook with an isomorphism proof in Coq and composes these certificates across dependencies (Saraswat et al., 2026), but this requires importing the Lean output back into Coq. CEF needs no such bridge, because each half of a counterexample is checked in its own proof assistant. Counterexamples have also been used to evaluate generated specifications against reference ones (Ye et al., 2026b) and to disprove false theorems in Lean (Li et al., 2026a), whereas a CEF counterexample shows that two individually valid declarations in different logics disagree.

3 Counterexample-Based Faithfulness and Transformal

3.1 Problem Formulation

Let R_s be a verified source repository and R_t its translation into another proof assistant. A repository consists of public declarations $D = \{d_1, \ldots, d_N\}$, such as definitions, theorems, structures, and type classes. A translation consists of the target repository together with a declaration mapping M that names the target counterpart $M(s) \in D_t$ of each source declaration $s \in D_s$. The translation must supply M, because faithfulness is judged declaration by declaration and each judgment needs to know which target declaration stands for which source declaration. We view each declaration d as a map $d : X^d \to Y^d$ from inputs to observations. An input is an argument of a definition or an instantiation of a theorem, structure, or type class. The observation is the value that the definition returns, or whether the instantiated proposition is provable or refutable in the proof assistant. For a structure or type class, the observation is whether the instantiation is inhabited, so an instance that the

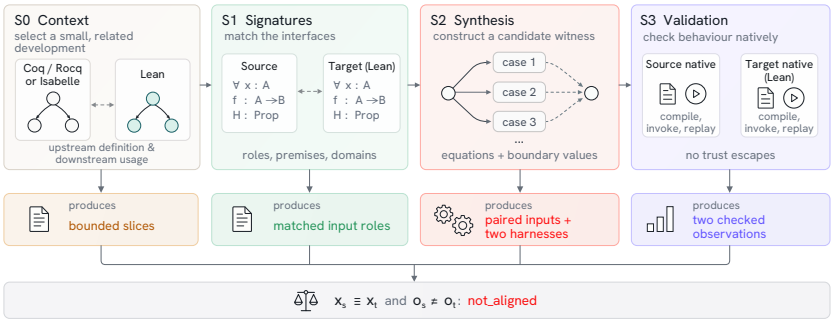

Figure 2: **Counterexample search in TRANSFORMAL.** Each stage produces the input of the next. S_0 selects a bounded dependency slice of each declaration, with its upstream definitions and downstream uses. S_1 matches the input roles of the two declarations from their elaborated signatures. S_2 derives equations and boundary values, from which the agent constructs a candidate that consists of paired inputs x_s and x_t and one harness per proof assistant. S_3 runs each harness in its own proof assistant without trust escapes such as `sorry` and returns the observations o_s and o_t. A candidate with $x_s \equiv x_t$ and $o_s \not\equiv o_t$ is a counterexample and yields `not_aligned`, and Appendix E traces the stages on a benchmark pair.

target admits and the source excludes is a counterexample. We write $a \equiv b$ when a source input or observation a and a target input or observation b semantically represent the same object, for example after reordering arguments or changing a numeral type. A declaration pair (s, t) is *faithful* when matched inputs always lead to matched observations:

$$\mathrm{F}(s,t) \iff \big(\forall x_s \in X^s,\ x_t \in X^t;\ x_s \equiv x_t \implies s(x_s) \equiv t(x_t)\big). \tag{1}$$

A repository translation is *semantically aligned* with its source when every source declaration is faithful to its counterpart:

$$\mathrm{Aligned}(R_s, R_t) \iff \forall s \in D_s,\ \mathrm{F}\big(s, M(s)\big). \tag{2}$$

Target declarations outside the image of M do not affect alignment, and Appendix D shows that this definition agrees with a behavioral view in which a repository answers queries about its declarations.

3.2 COUNTEREXAMPLE-BASED FAITHFULNESS

Equation 2 reduces checking a repository translation to checking each declaration pair. Each faithfulness judgment, however, quantifies over all matched inputs, and no finite search can establish it. Its negation, however, needs only one witness:

$$\neg\,\mathrm{F}(s,t) \iff \exists x_s \in X^s,\ x_t \in X^t;\ x_s \equiv x_t \ \wedge\ s(x_s) \not\equiv t(x_t). \tag{3}$$

We call a matched pair (x_s, x_t) that satisfies Equation 3 a *counterexample*. Unlike a direct proof of Equation 1, which needs a formal bridge between the two logics (Keller & Werner, 2010; Thiré, 2018; Saraswat et al., 2026), or an LLM verdict, which cannot be checked (Section 5.4), a counterexample splits into two halves that each proof assistant checks on its own. The only cross-language claims left are that the two inputs match and that the two observations differ, and both concern concrete values that are much easier to review than a claim about all inputs.

Counterexample-based faithfulness (CEF) judges each pair by searching for a counterexample. A counterexample replayed in both proof assistants refutes the pair and yields `not_aligned`. If the search finds none and at least one matched example has been replayed on both sides with agreeing observations, the pair satisfies CEF and is judged `aligned`. A pair that meets neither requirement after the repair rounds receives no verdict and counts as an error. A `not_aligned` verdict carries evidence checked in both proof assistants, whereas an `aligned` verdict means that the search found no counterexample and at least one matched example agreed on both sides, so the quality of CEF depends on how well the search locates counterexamples (Section 3.3). Because Equation 2 requires every declaration pair to be faithful, we score a repository translation by the share

of source declarations that have a compiling counterpart satisfying CEF, which measures how close the translation is to alignment.

A counterexample also carries more information than a verdict. It names a concrete input on which the two declarations diverge and shows both observations. A translation agent can use this information to locate and repair the defect, and it can confirm the repair by replaying the same counterexample.

3.3 TRANSFORMAL: COUNTEREXAMPLE SEARCH

TRANSFORMAL searches for counterexamples with an LLM agent. For each pair (s, t), the agent receives the dependency contexts C_s and C_t, which contain the upstream definitions, inherited assumptions, and downstream uses of the two declarations. It proposes candidate counterexamples, each consisting of a matched input and one harness per proof assistant that computes the observation for that input. Proposing a valid candidate requires facts that the source text states only implicitly, such as which inputs correspond, which premises each declaration carries, and where each definition branches. Four stages of program-analysis tools supply these facts from the native toolchains (Figure 2, and Table 8 in Appendix I).

Each stage supplies a fact that the agent would otherwise have to infer from source text, which is why adding stages makes TRANSFORMAL both more accurate and cheaper (Section 5.5). S_0 retrieves the dependency slice of the pair from the compiler indexes, including downstream uses that show which representation the target is meant to encode. The slice fixes what each declaration means but not how their inputs correspond, so S_1 compares the elaborated signatures to match inputs by role and to find inputs that one side accepts and the other rejects. To find where matched inputs diverge, S_2 applies symbolic execution to the equations in the slice and returns boundary values around each branch condition. S_3 then compiles and replays each candidate in both proof assistants and returns compiler diagnostics to the agent when a harness fails. It also requires that both harnesses apply the judged declaration itself and enforces the evidence rule for `aligned` verdicts, so the agent can neither mistake a helper's divergence for the pair's nor accept a pair without testing it.

4 BENCHMARK CONSTRUCTION

We build two benchmarks from real-world Coq and Isabelle repositories. TF-JUDGE measures how accurately a judge decides whether a translated declaration is faithful, and TF-TRANSLATION measures how faithfully an agent translates a whole repository into Lean.

Repositories and candidate translations. Our catalog contains 19 real-world source projects, 10 in Coq and 9 in Isabelle, spanning numerical computation, model checking, cryptography, programming-language semantics, theoretical computer science, and smart contracts. Target candidates come from repository-scale agent translations into Lean and from existing Lean translations, such as FloatSpec (Beneficial AI Foundation, 2025), a Lean port of Coq's Flocq library. GPT-5.6 Sol produces the agent translations with the existing agent pipeline of the FloatSpec repository. Each agent translation supplies the declaration mapping M by recording the target counterpart of every source declaration. For existing translations such as FloatSpec, which keep the source names and file structure, we recover M by matching names and elaborated signatures. For each candidate, we record the source–target declaration pairs given by M, their compiler-elaborated interfaces, and the declarations that each one depends on and is used by.

TF-JUDGE: translation faithfulness judging. Each TF-JUDGE instance consists of a source declaration s, its target counterpart t, and their dependency contexts C_s and C_t (Section 3.3). A judge receives (s, t, C_s, C_t) and decides whether t is faithful to s. The benchmark contains 297 pairs from 18 source projects and 22 translation tasks, since four projects have two target translations each. Formal-methods experts labeled each pair as `aligned` (faithful) or `not_aligned` (unfaithful), which gives 234 faithful and 63 unfaithful pairs. TF-JUDGE is a deliberately difficult diagnostic set rather than a sample of typical translations. We selected 192 pairs, which we call the hard suite, from the 5,097 translated declarations because they were hard for earlier judges, and a screening judge misjudged 177 of them. The remaining 105 pairs are all Isabelle pairs of a human-reviewed priority tier. All labels were fixed by human review before any method was evaluated. Because faithful pairs dominate, we report macro-F1, MCC, and balanced accuracy alongside accuracy. The pairs cover

six domains, and numerical computation, with 156 FloatSpec pairs, is the largest (Appendix G). The source repositories and FloatSpec are public, but the agent-produced translations and all labels are new, so no evaluated model can have seen the verdicts during training.

TF-TRANSLATION: repository translation. Each TF-TRANSLATION task gives the agent a source repository together with its native toolchain and dependency structure, and asks for a Lean repository together with the declaration mapping M, which names the target counterpart of each source declaration. A source declaration without a counterpart counts as a failure. TF-TRANSLATION contains four repositories, Coq-SKI and KA-FMP from Coq and VeriComp and Abstract Rewriting from Isabelle, with 1,185 source declarations in total. A translated declaration is correct only if it compiles and satisfies CEF, and we report the fraction of source declarations that are correct.

Table 1: Statistics of TF-JUDGE per declaration pair and of TF-TRANSLATION per repository.

	Source		Target	
TF-JUDGE ($N = 297$)	Med.	Max	Med.	Max
Statement lines	2	16	4	28
Declaration lines	6	276	7	327
Direct dependencies	2	21	2	16
Transitive dependencies	5	91	6	70

	Source	
TF-TRANSLATION ($N = 4$)	Med.	Max
Files	9	14
Lines	5,375	7,582
Declarations	285	520

Benchmark statistics. Table 1 summarizes both benchmarks, where a statement is a declaration header without its body and the dependencies of a declaration are the declarations of the same repository that it refers to outside its proof. Statements in TF-JUDGE are short, with a median of 2 source lines and 4 target lines, but each depends transitively on a median of 5 source declarations and on up to 91, so a judge must resolve the meaning of a pair from its repository context rather than from the two statements alone.

5 EXPERIMENTS & EVALUATION

Table 2: Translation faithfulness judging on the 297 pairs of TF-JUDGE. Bal. acc. is the mean of aligned and misaligned recall. For GPT-5.6 Sol and the Claude Opus 5 baselines, abstentions and invalid outputs count as wrong. GLM 5.3 and DeepSeek V4 Pro are evaluated on the 192-pair hard suite, and their rows and the Claude Opus 5 TRANSFORMAL row are computed over definitive verdicts. Parentheses give the change from the same model's Raw LLM. Purple bold marks the best result per column within each model.

Model	**Method**	**Acc. (%)**	**Macro-F1 (%)**	**MCC**	**Bal. acc. (%)**
N/A	Round-trip + BEq	22.56	19.38	0.061	50.85
GPT-5.6 Sol	Raw LLM	59.93	63.89	0.312	58.33
	Tool Call	57.91 (-2.02)	63.32 (-0.57)	0.294 (-0.018)	57.63 (-0.70)
	TRANSFORMAL	**94.28** (+34.35)	**92.35** (+28.46)	**0.835** (+0.523)	**92.89** (+34.56)
Claude Opus 5	Raw LLM	78.79	73.74	0.477	75.52
	Tool Call	79.46 (+0.67)	74.60 (+0.86)	0.500 (+0.023)	78.27 (+2.75)
	TRANSFORMAL	**97.67** (+18.88)	**87.85** (+14.11)	**0.760** (+0.283)	**85.30** (+9.78)
GLM 5.3	Raw LLM	78.41	68.12	0.397	74.13
	Tool Call	79.87 (+1.46)	70.81 (+2.69)	0.439 (+0.042)	**75.67** (+1.54)
	TRANSFORMAL	**94.35** (+15.94)	**75.16** (+7.04)	**0.541** (+0.144)	69.56 (-4.57)
DeepSeek V4 Pro	Raw LLM	64.94	49.13	0.019	51.23
	Tool Call	70.59 (+5.65)	59.45 (+10.32)	0.243 (+0.224)	65.70 (+14.47)
	TRANSFORMAL	**95.45** (+30.51)	**82.12** (+32.99)	**0.654** (+0.635)	**77.95** (+26.72)

5.1 EXPERIMENT SETUP

Judges. We evaluate four models, GPT-5.6 Sol (Singh et al., 2025) with high reasoning effort, Claude Opus 5 (Anthropic, 2026), GLM 5.3, and DeepSeek V4 Pro (Xu et al., 2026), and run every agent in the Codex harness (OpenAI, 2025). For each model, we compare TRANSFORMAL with two baselines. *Raw LLM* judges a compact source–target case in a single prompt without tools, and *Tool Call* judges it as an agent with file and execution tools but without the stages of TRANSFORMAL. A

model-independent baseline, Round-trip + BEq, exports the Lean target back into Coq, as in lf-lean (Saraswat et al., 2026), and proves bidirectional obligations against the source, as in BEq (Liu et al., 2025; Poiroux et al., 2025). We score judges against the expert labels of TF-JUDGE with accuracy, macro-F1, MCC, and balanced accuracy, and count abstentions and invalid outputs as errors.

Translation. Each agent first translates each TF-TRANSLATION repository into Lean and then refines its translation for three rounds under one of three feedback signals: compiler feedback (Lean build and test diagnostics), LLM judge feedback (Raw LLM verdicts and rationales), or TRANSFORMAL feedback (verdicts with their counterexamples). All other settings are identical across conditions (Appendix H). We score each translation by the share of source declarations that have a compiling counterpart satisfying CEF, computed by TRANSFORMAL with GPT-5.6 Sol, and count missing declarations and declarations without a definitive verdict as failures.

5.2 EXPERIMENT RESULTS: TRANSLATION FAITHFULNESS JUDGING

Overall Performance Table 2 compares TRANSFORMAL with Raw LLM, Tool Call, and Round-trip + BEq. TRANSFORMAL gives the highest accuracy, macro-F1, and MCC for all four models, and it raises accuracy over Raw LLM by 15.94 to 34.35 points. With GPT-5.6 Sol, for example, accuracy rises from 59.93% to 94.28%, a 1.57× improvement. The largest MCC gain is with DeepSeek V4 Pro, whose direct verdicts are close to chance with an MCC of 0.019. TRANSFORMAL raises its accuracy from 64.94% to 95.45% and its MCC to 0.654. Tool access alone does not produce these gains, since Tool Call stays within 6 points of Raw LLM accuracy for every model. The gains instead come from counterexample search. Round-trip + BEq flags every misaligned pair but accepts almost no faithful one, so it reaches only 22.56% accuracy and an MCC of 0.061. TRANSFORMAL also raises balanced accuracy over Raw LLM by 9.78 to 34.56 points for three of the four models. The exception is GLM 5.3, whose balanced accuracy drops from 74.13% to 69.56% because it rarely completes a counterexample search. It returns a valid verdict on only 29 of the misaligned pairs, and the others count as errors. Section 5.4 analyzes the remaining errors.

Domains and languages Table 5 in Appendix G breaks down the GPT-5.6 Sol results by application domain and source language. TRANSFORMAL reaches between 90.00% and 96.67% accuracy in every subset, while the baselines range from 0.00% to 80.95%. The baselines fail in different ways depending on how the translation changes the source. Coq translations such as FloatSpec often change the surface representation but preserve the meaning, and the baselines over-reject them. Raw LLM and Tool Call accept only 54.19% and 45.81% of the faithful Coq pairs, compared with 95.48% for TRANSFORMAL. Isabelle translations often keep the surface syntax but lose implicit context, such as locale assumptions and inherited class premises, and the baselines over-accept them. Both baselines detect only 32.35% of the unfaithful Isabelle pairs, compared with 91.18% for TRANSFORMAL. Because TRANSFORMAL decides from replayed observations rather than surface similarity, its aligned and misaligned recall differ by less than 6 points on both languages, while the gap for the baselines ranges from 28 to 50 points. Appendix G gives the per-subset error breakdown and the detailed cases.

5.3 EXPERIMENT RESULTS: REPOSITORY TRANSLATION

Results TRANSFORMAL feedback is the only signal that improves translations for every model (Figure 3(a)). Initial accuracy ranges from 21.92% for GLM 5.3 to 85.33% for Claude Opus 5. After one round, TRANSFORMAL feedback raises the accuracy of every model, by up to 5.47 points for GLM 5.3, while compiler and LLM judge feedback lower it for every model. Over three rounds with GPT-5.6 Sol (Figure 3(b)), TRANSFORMAL feedback is the only signal that improves in every round. It raises accuracy from 80.25% to 92.48%, a gain of 12.23 points and four times the 3.02-point gain of compiler feedback, the best baseline signal. The gap follows from the accuracy of the signal. The Raw LLM judge falsely rejects 31 of the 234 faithful pairs in TF-JUDGE, and these findings direct the agent to rewrite declarations that are already faithful. TRANSFORMAL falsely rejects only 9 (Table 6) and backs each finding with a counterexample replayed in both proof assistants. Even with TRANSFORMAL feedback, 7.52% of source declarations remain unfaithful or untranslated after three rounds, so repository translation remains far from solved. Appendix H reports the per-model and per-round results.

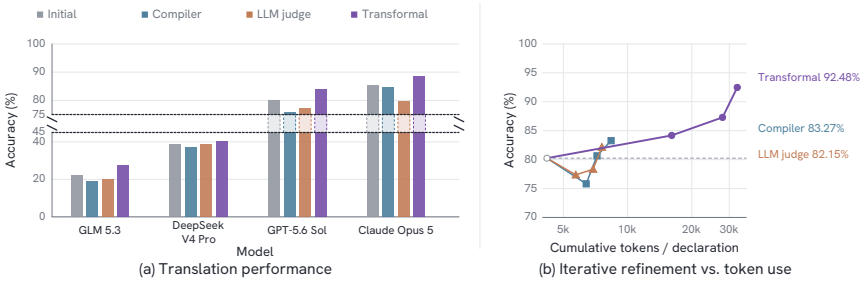

Figure 3: Repository translation on TF-TRANSLATION under compiler, LLM judge, and TRANSFORMAL feedback. (a) Accuracy of each model for the initial translation and after one refinement round under each signal. The vertical axis is broken between 45% and 75%. (b) Accuracy of GPT-5.6 Sol over three refinement rounds against the cumulative tokens per source declaration, on a log scale. The open circle and the dashed line mark the initial translation, and each filled marker is one round.

5.4 FAILURE MODE ANALYSIS

Judge failure modes The baseline errors rarely come from reasoning about the formulas themselves. We manually inspected the definitive errors of Raw LLM and Tool Call with GPT-5.6 Sol and grouped them into six failure modes (Table 9 in Appendix I). In most of them, the baselines compute the relevant values correctly but choose the wrong cross-language correspondence before computing. For example, they pair binders by position rather than by role (F1), or they read a target that admits more instances as a harmless extension of the input domain (F2). They also make these errors with high confidence. On the 192-pair hard suite, every one of the 32 and 30 definitive errors of the two baselines carries a confidence of at least 0.98. TRANSFORMAL addresses each failure mode with a deterministic stage rather than more LLM reasoning. The compiler indexes of S_0 and S_1 fix the correspondence, S_2 turns branch conditions into concrete inputs, and S_3 replays both sides and rejects evidence that does not bind to the judged declaration. The remaining errors of TRANSFORMAL concern the matching relation $\equiv$ on inputs and observations (Section 3.1), which the proof assistants validate only locally. Most false acceptances come from a matching relation that explains a real difference away, and most false rejections attach a valid observation to the wrong public interface. Appendix I describes each failure mode and the remaining errors with concrete cases.

Translation failure modes The three feedback signals differ in what they tell the translation agent, which explains the trends in Figure 3. Compiler feedback is precise but stops once a declaration type-checks. Its diagnostics are also attached identically to every declaration in a failing module, and in VeriComp all 40 findings repeat the same unknown-constant errors. LLM judge feedback is mostly not actionable. With GPT-5.6 Sol, all 42 semantic findings on Coq-SKI and 34 of the 35 on VeriComp are `uncertain` verdicts that cite missing context, so the agent rewrites declarations that were already faithful and introduces regressions. TRANSFORMAL feedback instead gives each `not_aligned` finding a matched input and two replayed observations, which the agent can check its repair against. In an earlier refinement study, the agent used such findings to repair defects that compile cleanly, such as a `monotone` record translated as a `Prop` instead of a `Type`, and dismissed 33 of 192 findings as noise. Some defects persist even with TRANSFORMAL feedback, mainly representation changes that no local edit can fix and repairs that fit the reported witness but not the underlying semantics. Appendix J details each signal and the remaining defects with concrete cases.

5.5 ABLATIONS AND EFFICIENCY

Stages and cost Figure 4 adds the stages of TRANSFORMAL one at a time on the hard suite. The instruction to search for counterexamples alone raises accuracy from 59.9% to 85.4%, and the program-analysis stages raise it to 94.8% without lowering it at any step. The stages also make TRANSFORMAL cheaper. Tokens per case fall from 140.3k for Raw LLM to 60.7k for the full TRANSFORMAL, and the cost per correct verdict falls from $0.92 to $0.25, because deterministic tools supply facts that the LLM would otherwise infer through long prompts and repeated attempts.

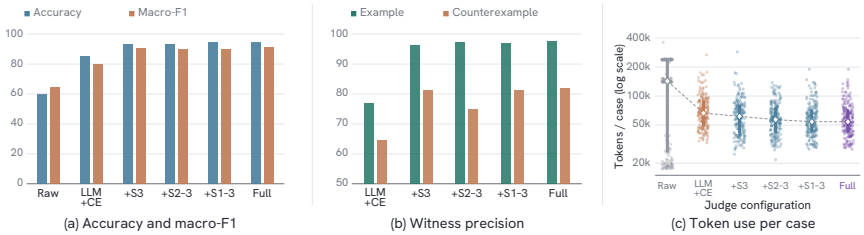

Figure 4: Ablation of the TRANSFORMAL stages with GPT-5.6 Sol on the 192-pair hard suite. Raw is the Raw LLM judge, and LLM+CE adds only the instruction to search for counterexamples. $+S_3$, $+S_2$–S_3, and $+S_1$–S_3 add the corresponding stages, and Full adds S_0 as well. (a) Accuracy and macro-F1 (%). (b) Witness precision (%). Among the cases that propose an example or a counterexample, it is the share whose witness replays on both sides and agrees with the human label. Raw proposes no witnesses, and the vertical axis starts at 50%. (c) Model tokens per case on a log scale. Dots are individual cases, bars span the interquartile range, and diamonds mark the medians. Cases without complete token telemetry are omitted.

Raw LLM must read the whole dependency closure in one prompt, so 99.1% of its tokens are input tokens, whereas TRANSFORMAL retrieves bounded slices and elaborated interfaces on demand. Its witnesses are also rejected less often by the two-sided validation, so model attempts per case fall from 1.354 with counterexample search alone to 1.151 with all stages. S_3 accounts for most of the gain, raising accuracy from 85.4% to 93.2%, so lighter configurations suit tight budgets. Appendix K gives the full breakdown.

Repeated runs Repeated sampling does not close the gap between Raw LLM and TRANSFORMAL. On 58 pairs, three Raw LLM passes with GPT-5.6 Sol reach 48.28% pass@1 and only 51.72% pass@3, and majority voting over the three passes returns to 48.28%. Forty-five of the 58 pairs receive the same verdict in every pass, 21 of them wrong every time. The errors of Raw LLM are therefore systematic, and the gain of TRANSFORMAL comes from counterexample search rather than from sampling luck (Appendix L).

6 CONCLUSION

We introduced counterexample-based faithfulness (CEF), which reduces checking the faithfulness of a formal repository translation to finding counterexamples for its declaration pairs, and TRANSFORMAL, which searches for such counterexamples with an LLM agent supported by program analysis in both proof assistants. On TF-JUDGE, TRANSFORMAL judges faithfulness more accurately than direct LLM judgment for all four models we evaluate, and reaches 94.28% accuracy with GPT-5.6 Sol. On TF-TRANSLATION, its counterexamples serve as feedback that raises the share of GPT-5.6 Sol's declarations that satisfy CEF from 80.25% to 92.48% in three rounds, four times the gain of compiler feedback. We hope that CEF, TRANSFORMAL, and our benchmarks will serve as a foundation for agents that translate verified developments faithfully across proof assistants, so that results proved in one system can be reused in others.

Future work. Most remaining judge errors come from how the agent matches source and target values, which each proof assistant checks only locally, so certifying this correspondence is the next step toward fully automatic checking. An `aligned` verdict still means only that the search found no counterexample, and a stronger search would make these verdicts more reliable. Some translation defects also require a change of representation that no local repair can make, and extending TF-JUDGE and TF-TRANSLATION to more proof assistants and larger repositories will show how far counterexample feedback scales.

REFERENCES

Anthropic. Claude opus 5 system card, July 2026. URL https://www-cdn.anthropic.com/c5fbac3f0b1280a933ebd26d3cb8bb9f5bdeaf48/Claude%20Opus%205%20System%20Card.pdf.

Zhangir Azerbayev, Bartosz Piotrowski, Hailey Schoelkopf, Edward W Ayers, Dragomir Radev, and Jeremy Avigad. Proofnet: Autoformalizing and formally proving undergraduate-level mathematics. *arXiv preprint arXiv:2302.12433*, 2023.

Beneficial AI Foundation. Floatspec, 2025. URL https://github.com/Beneficial-AI-Foundation/FloatSpec. GitHub repository. Formally Verified Float Implementation with lean4. Accessed 2026-09-24.

Jasmin Christian Blanchette, Maximilian Haslbeck, Daniel Matichuk, and Tobias Nipkow. Mining the Archive of Formal Proofs. In *Intelligent Computer Mathematics (CICM)*, volume 9150 of *Lecture Notes in Computer Science*, pp. 3–17. Springer, 2015. doi: 10.1007/978-3-319-20615-8_1.

Frédéric Blanqui. Translating HOL-Light proofs to Coq. In *Proceedings of the 25th Conference on Logic for Programming, Artificial Intelligence and Reasoning (LPAR)*, volume 100 of *EPiC Series in Computing*, pp. 1–18. EasyChair, 2024. doi: 10.29007/6k4x.

Sylvie Boldo and Guillaume Melquiond. Flocq: A unified library for proving floating-point algorithms in Coq. In *20th IEEE Symposium on Computer Arithmetic (ARITH)*, pp. 243–252. IEEE, 2011. doi: 10.1109/ARITH.2011.40.

Qian Cheng, Saad Mohammad Rafid Pial, Ruize Tang, Yiming Su, Emilie Ma, Finn Hackett, Ivan Beschastnikh, Yu Huang, and Tianyin Xu. Specula: Scaling formal specifications for autonomous model checking of system code. *arXiv preprint arXiv:2607.25333*, 2026.

Rob Cornish, Iacopo Ghinassi, Po-Hung Yeh, Shuqi Liu, Qiyuan Xu, Haoxuan Yin, Dominik Wagner, Wenda Li, Yee Whye Teh, and Luke Ong. Faithformbench: Benchmarking faithfulness of mathematical chain-of-thought autoformalisation. *arXiv preprint arXiv:2608.10916*, 2026.

Yuming Feng, Frederick Pu, One An, Osbert Bastani, Li Zhang, Jiani Huang, Xujie Si, and Ziyang Li. Theory-scale auto-formalization of logics for computer science. *arXiv preprint arXiv:2606.26525*, 2026.

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. In *Proceedings of the 2026 Conference on Empirical Methods in Natural Language Processing*, 2026.

Cezary Kaliszyk and Alexander Krauss. Scalable LCF-style proof translation. In *Interactive Theorem Proving, 4th International Conference (ITP)*, volume 7998 of *Lecture Notes in Computer Science*, pp. 51–66. Springer, 2013. doi: 10.1007/978-3-642-39634-2_7.

Chantal Keller and Benjamin Werner. Importing HOL Light into Coq. In *Interactive Theorem Proving, First International Conference (ITP)*, volume 6172 of *Lecture Notes in Computer Science*, pp. 307–322. Springer, 2010. doi: 10.1007/978-3-642-14052-5_22.

Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. seL4: Formal verification of an OS kernel. In *Proceedings of the 22nd ACM Symposium on Operating Systems Principles (SOSP)*, pp. 207–220. ACM, 2009. doi: 10.1145/1629575.1629596.

Joshua Ong Jun Leang, Haonan Li, Zheng Zhao, Xinyi Shang, Wenda Li, Zhengzhong Liu, Eric Xing, Shay B. Cohen, and Eleonora Giunchiglia. Magenta: Closing the loop between mathematical reasoning and Lean verification. *arXiv preprint arXiv:2609.11319*, 2026.

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

Zenan Li, Zhaoyu Li, Kaiyu Yang, Xiaoxing Ma, and Zhendong Su. Learning to disprove: Formal counterexample generation with large language models. *arXiv preprint arXiv:2603.19514*, 2026a.

Zenan Li, Ziran Yang, Peiyang Song, Zhaoyu Li, and Kaiyu Yang. P^3: Joint program-and-proof planning for verified code generation. *arXiv preprint arXiv:2608.09277*, 2026b.

Guanyu Liu, Jingkun Ma, and Derek F Wong. Lean2isabelle: Factorized cross-assistant proof translation. In *3rd AI for Math Workshop: Toward Self-Evolving Scientific Agents at ICML 2026*, 2026.

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*, volume 2025, pp. 86205–86233, 2025.

Steven Obua and Sebastian Skalberg. Importing HOL into Isabelle/HOL. In *Automated Reasoning, Third International Joint Conference (IJCAR)*, volume 4130 of *Lecture Notes in Computer Science*, pp. 298–302. Springer, 2006. doi: 10.1007/11814771_27.

OpenAI. codex, 2025. URL `https://github.com/openai/codex`. GitHub repository. Lightweight coding agent that runs in your terminal. Accessed 2026-09-23.

Auguste Poiroux, Gail Weiss, Viktor Kunčak, and Antoine Bosselut. Reliable evaluation and benchmarks for statement autoformalization. In *Proceedings of the 2025 Conference on Empirical Methods in Natural Language Processing*, pp. 17947–17969, 2025. doi: 10.18653/v1/2025.emnlp-main.907.

Yuanjie Ren, Jinzheng Li, and Yidi Qi. Merlean: An agentic framework for autoformalization in quantum computation. *arXiv preprint arXiv:2602.16554*, 2026.

Vishesh Saraswat, Jeffrey Chang, Harshikaa Agrawal, Vaidehi Agarwalla, Robert Zhang, Twm Stone, Jacob Green, Shiki Vaahan, Lawrence Chan, Rajashree Agrawal, and Jason Gross. lf-lean: The frontier of verified software engineering. Theorem blog, February 2026. URL `https://theorem.dev/blog/lf-lean/`. Public artifact: https://github.com/theorem-labs/lf-lean.

Aaditya Singh, Adam Fry, Adam Perelman, Adam Tart, Adi Ganesh, Ahmed El-Kishky, Aidan McLaughlin, Aiden Low, AJ Ostrow, Akhila Ananthram, et al. Openai gpt-5 system card. *arXiv preprint arXiv:2601.03267*, 2025.

Théo Stoskopf, Cyril Cohen, and Nicolas Tabareau. Babel-formal: Translation of proofs between lean and rocq. In *The 5th Workshop on Mathematical Reasoning and AI at NeurIPS 2025*, 2025.

Chuyue Sun, Yican Sun, Daneshvar Amrollahi, Ethan Zhang, Shuvendu Lahiri, Shan Lu, David Dill, and Clark Barrett. Veristruct: Ai-assisted automated verification of data-structure modules in verus. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pp. 109–128. Springer, 2026.

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 (NeurIPS) Datasets and Benchmarks Track*, volume 38, 2025. doi: 10.52202/085713-3857.

The mathlib Community. The Lean mathematical library. In *Proceedings of the 9th ACM SIGPLAN International Conference on Certified Programs and Proofs (CPP)*, pp. 367–381. ACM, 2020. doi: 10.1145/3372885.3373824.

François Thiré. Sharing a library between proof assistants: Reaching out to the HOL family. In *Proceedings of the 13th International Workshop on Logical Frameworks and Meta-Languages: Theory and Practice (LFMTP)*, volume 274 of *Electronic Proceedings in Theoretical Computer Science*, pp. 57–71, 2018. doi: 10.4204/EPTCS.274.5.

Jules Viennot, Guillaume Baudart, Emilio Jesús Gallego Arias, and Marc Lelarge. MiniF2F in Rocq: Automatic translation between proof assistants–a case study. In *The 5th Workshop on Mathematical Reasoning and AI at NeurIPS 2025*, 2025.

Zhongyi Wang, Tengjie Lin, Mingshuai Chen, Mingqi Yang, Haokun Li, Xiao Yi, Shengchao Qin, and Jianwei Yin. Preguss: It analyzes, it specifies, it verifies. In *Proceedings of the 1st ACM SIGPLAN International Workshop on Language Models and Programming Languages*, pp. 118–123, 2025.

Jiayi Wu, Robert Joseph George, and Anima Anandkumar. Itpeval: Benchmarking formal translation across interactive theorem provers. *arXiv preprint arXiv:2607.19407*, 2026.

Yuhuai Wu, Albert Qiaochu Jiang, Wenda Li, Markus Rabe, Charles Staats, Mateja Jamnik, and Christian Szegedy. Autoformalization with large language models. *Advances in neural information processing systems*, 35:32353–32368, 2022.

Yutong Xin, Qiaochu Chen, Greg Durrett, and Işil Dillig. VeriSoftBench: Repository-scale formal verification benchmarks for Lean. In *Conference on Language Modeling (COLM)*, 2026.

Anyi Xu, Bangcai Lin, Bing Xue, Bingxuan Wang, Bingzheng Xu, Bochao Wu, Bowei Zhang, Chaofan Lin, Chen Dong, Chenchen Ling, et al. Deepseek-v4: Towards highly efficient million-token context intelligence. *arXiv preprint arXiv:2606.19348*, 2026.

Zhe Ye, Hantao Lou, Yuechun Sun, Peiyang Song, Zhengxu Yan, Timothe Kasriel, Qingyang Zhang, Kaiyu Yang, Soonho Kong, Jingxuan He, et al. Vero: Can ai agents build formally verified software repositories? *arXiv preprint arXiv:2608.13522*, 2026a.

Zhe Ye, Zhengxu Yan, Jingxuan He, Timothe Kasriel, Kaiyu Yang, and Dawn Song. Verina: Benchmarking verifiable code generation. In *International Conference on Learning Representations*, volume 2026, pp. 38933–38972, 2026b.

Zhe Ye, Aidan Z. H. Yang, Huangyuan Su, Zhenyu Liao, Samuel Tenka, Zhizhen Qin, Udaya Ghai, Dawn Song, and Soonho Kong. Intent-aligned formal specification synthesis via traceable refinement. *arXiv preprint arXiv:2604.10392*, 2026c.

A AI USE STATEMENT

Large language models are a central object of study in this work and are also components of the proposed method. Frontier models (GPT-5.6 Sol, Claude Opus 5, GLM 5.3, and DeepSeek V4 Pro) serve as the evaluated judges and translation agents. In TRANSFORMAL, an LLM agent proposes candidate examples and counterexamples, which are then validated by the native proof assistants. Some target repositories in TF-JUDGE were produced by LLM translation agents, as described in Section 4. All benchmark labels were assigned by human formal-methods experts, not by LLMs. LLM-based coding assistants were also used to help implement the experimental infrastructure and to aggregate and summarize experimental results. The authors reviewed, verified, and revised all such content and take full responsibility for the paper, including every reported number, claim, and citation.

B ETHICS STATEMENT

All authors have read and adhere to the ICLR Code of Ethics. TF-JUDGE and TF-TRANSLATION are built from publicly available formal-verification repositories distributed under open-source licenses, including MIT, BSD, Apache 2.0, (L)GPL, MPL 2.0, and CeCILL-B. When we release TF-JUDGE and TF-TRANSLATION, we will preserve each source project's license and attribution and pin every repository to the snapshot we used. Both benchmarks consist of formal specifications, proofs, and programs, and contains no personal or sensitive data. The faithfulness labels were produced by expert annotators who reviewed source-target declaration pairs. This annotation involved no human-subject experimentation. Since this paper is focused on formal repository translation, a new research question in formal verification, we do not foresee direct harmful applications. The main risk we identify is overtrust: a translated repository that compiles, or that receives an `aligned` verdict, may still be unfaithful. We therefore report the soundness limits of TRANSFORMAL explicitly and recommend human review before a translated specification is relied upon in safety-critical settings. The experiments used commercial model APIs. We report their token consumption in Section 5.5 to make the computational cost of the method transparent. The authors declare no conflicts of interest or sponsorship relevant to this work.

C Reproducibility Statement

We have made the following efforts to make our results reproducible. The task and metric are formally defined in Section 3: repository faithfulness in Equation 2, declaration faithfulness in Equation 1, and the counterexample criterion in Equation 3. The program-analysis stages are described in Section 3.3 and Table 8, and the failure modes they address are analyzed in Section 5.4. Section 4 and Appendix F describe how TF-JUDGE and TF-TRANSLATION were constructed and composed. Models, reasoning-effort settings, baselines, and metrics are specified in the experiment setup, including the fail-closed treatment of abstentions and invalid outputs. Model APIs are nondeterministic, so we quantify run-to-run variation through the repeated-run experiment in Appendix L.

Because OpenReview submissions are public and a fully labeled benchmark released at this stage could leak into the training data of future models, we release the artifacts in two stages. With the submission, we provide the following in the anonymized repository `https://anonymous.4open.science/r/transformal` and the OpenReview supplementary materials:

- the complete judge and translation pipeline code, with the prompts and demonstrations used in every configuration, and the scripts that run each baseline and stage configuration;
- the scoring and plotting scripts for every table and figure;
- a sample of TF-JUDGE covering every domain and both source languages, with its human-annotated labels, source and target contexts, and per-case results, including the compiler-replayed witnesses of TRANSFORMAL;
- one complete repository-translation task, with its pinned source commit, target-environment snapshot, and feedback-round outputs.

Upon acceptance, we will publicly release TF-JUDGE and TF-TRANSLATION in full, along with the human-annotated ground-truth labels, together with the complete artifacts.

D Derivation of the Declaration-Level Decomposition

Section 3 defines alignment declaration by declaration (Equation 2). This appendix shows that the definition agrees with a behavioral view of a whole repository. In that view, a repository answers queries (d, x) with $d \in D$ and $x \in X^d$ by $R(d, x) = d(x)$, a source query (s, x_s) matches a target query (t, x_t) exactly when $t = M(s)$ and $x_s \equiv x_t$, and R_t is behaviorally aligned with R_s when matched queries always receive matched answers. Unfolding these definitions gives Equation 2:

$$
\begin{aligned}
&\forall (s, x_s),\ (t, x_t);\ (s, x_s) \equiv (t, x_t) \implies R_s(s, x_s) \equiv R_t(t, x_t) \\
\iff\ &\forall s \in D_s,\ x_s \in X^s,\ x_t \in X^{M(s)};\ x_s \equiv x_t \implies R_s(s, x_s) \equiv R_t\big(M(s), x_t\big) \\
\iff\ &\forall s \in D_s,\ x_s \in X^s,\ x_t \in X^{M(s)};\ x_s \equiv x_t \implies s(x_s) \equiv M(s)(x_t) \\
\iff\ &\forall s \in D_s,\ \mathrm{F}\big(s, M(s)\big).
\end{aligned}
$$

The first step uses the definition of query matching, under which matched queries address a declaration pair $(s, M(s))$ with matched arguments. The second step applies $R(d, x) = d(x)$, and the last step is Equation 1 with $t = M(s)$. Queries to target declarations outside the image of M match no source query, so they do not affect behavioral alignment either.

E Worked Example of Counterexample Search

Figure 5 traces the stages of Figure 2 on one TF-JUDGE pair, the Flocq theorem `abs_round_le_generic` and its FloatSpec translation. The Coq theorem states that rounding preserves an upper bound: if $|x| \le y$ and y is in the generic format, then $|\mathrm{round}(x)| \le y$. The Lean theorem states the same bound for `round_to_generic`, which takes the rounding mode as a relation. S_0 retrieves the dependencies of both theorems, and S_1 matches their inputs, namely the radix β, the exponent function `fexp`, the rounding mode, and the values x and y. S_2 returns no branch for this pair, because the target fixes the rounding mode through a type-class instance rather than through a branch. The agent instead reads this instance in the target source and finds that

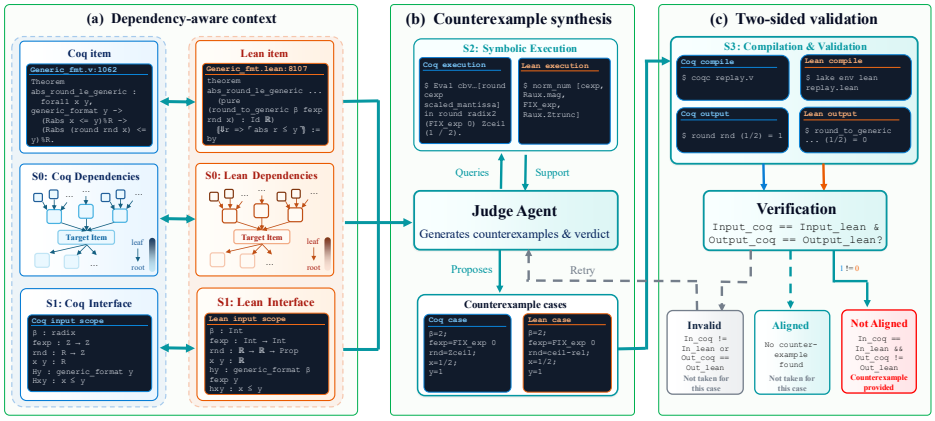

Figure 5: **Counterexample search on `abs_round_le_generic`. (a)** S_0 retrieves the dependency contexts of the paired declarations, and S_1 extracts their elaborated signatures. **(b)** The agent queries the stages and proposes a candidate counterexample, consisting of a matched input and one harness per proof assistant. For this pair, S_2 returns no branch, and the agent finds the truncating instance in the target source. **(c)** S_3 compiles and replays both harnesses and compares their observations. A candidate with incompatible observations is a counterexample and yields `not_aligned`. Failed candidates return to the agent with compiler diagnostics. If the search finds no counterexample and a replayed example agrees on both sides, the verdict is `aligned`.

Table 3: Source repositories of TF-TRANSLATION. Declarations are indexed public source declarations.

Source repository	Language	Files	Source lines	Declarations
Coq-SKI	Coq	8	4,306	257
KA-FMP	Coq	14	7,582	313
Abstract Rewriting	Isabelle	7	6,444	520
VeriComp	Isabelle	10	2,040	95
Total		39	20,372	1,185

`round_to_generic` rounds by truncation regardless of the given rounding relation. It therefore proposes radix 2, the fixed exponent `FIX_exp 0`, upward rounding, $x = 1/2$, and $y = 1$. S_3 compiles and replays both harnesses. Coq rounds $1/2$ up to 1, while the Lean function truncates it to 0. The inputs match and the observations differ, so the verdict is `not_aligned`.

F Details on Benchmark Composition

Repository translation tasks. Table 3 reports the four source repositories of TF-TRANSLATION, two Coq developments and two Isabelle developments. The counts come from pinned source snapshots and their compiler-resolved declaration inventories. Files and physical source lines count the `.v` or `.thy` files in each checked-out project, including comments and blank lines. Declarations count indexed public source declarations, which are the units on which translation accuracy is scored. The Coq repositories contribute 570 declarations and the Isabelle repositories contribute 615, for 1,185 declarations overall.

Translation faithfulness judging cases. Table 4 breaks down the 297 human-adjudicated source–target declaration pairs by source language and label. These pairs come from 18 source projects and 22 translation tasks, because FloatSpec, EIP20, CAVA, and Promela each have two target variants. The set contains 184 Coq-to-Lean and 113 Isabelle-to-Lean pairs, and the labels comprise 234 `aligned` and 63 `not_aligned` pairs. This is a deliberately difficult diagnostic set rather than a prevalence estimate. The first 192 pairs form the hard suite and were selected to be hard for earlier judges, and a screening judge run misjudged 177 of them. The remaining 105 Isabelle pairs are all

Table 4: Composition of TF-JUDGE.

Source → target	Repos.	Tasks	Pairs	Aligned	Not aligned
Coq → Lean	9	11	184	155	29
Isabelle → Lean	9	11	113	79	34
Total	18	22	297	234	63

Table 5: Translation faithfulness judging accuracy (%) of GPT-5.6 Sol by domain and source language. N gives the number of pairs as total (aligned/not aligned). Abstentions and invalid outputs count as errors. Purple bold marks the best result per row.

Subset	N	Raw LLM	Tool Call	TRANSFORMAL
Numerical computation	156 (129/27)	61.54	53.21	**94.87**
Model checking	62 (35/27)	61.29	66.13	**93.55**
Cryptography	30 (26/4)	53.33	60.00	**96.67**
PL semantics & rewriting	21 (18/3)	76.19	80.95	**90.48**
CS theory	18 (17/1)	66.67	72.22	**94.44**
Smart contracts & algorithms	10 (9/1)	0.00	0.00	**90.00**
Coq → Lean	184 (155/29)	58.70	52.17	**94.57**
Isabelle → Lean	113 (79/34)	61.95	67.26	**93.81**

pairs of a human-reviewed priority tier. FloatSpec contributes 156 pairs. Because `aligned` pairs dominate, Table 2 reports macro-F1, MCC, and balanced accuracy alongside accuracy. Human review fixed the labels before the blind comparison.

G TRANSLATION FAITHFULNESS JUDGING BY DOMAIN AND LANGUAGE

This appendix details the domain and language breakdown of Section 5.2. Table 5 reports accuracy by domain and source language. Table 6 splits the errors of each GPT-5.6 Sol method into false rejections of faithful translations, false acceptances of unfaithful translations, and abstentions, which include invalid outputs. It also reports aligned recall and misaligned recall, the fractions of faithful and unfaithful pairs that receive the correct verdict. The cases below show the three baseline failure patterns and the evidence that TRANSFORMAL produces for each of them.

Representation changes cause false rejections. Numerical computation is the largest domain, with 156 FloatSpec pairs. Raw LLM and Tool Call reach only 55.81% and 44.96% aligned recall on it. They reject 28 and 27 faithful translations and abstain on another 30 and 45. FloatSpec changes representations while preserving meaning. It reorders section variables, normalizes float payloads, and packages premises as type classes, and the baselines read these surface changes as semantic ones. Listing 1 in Appendix I shows a typical case. Flocq declares the section variables in the order $(e_{\min}, prec)$ and FloatSpec declares them as $(prec, e_{\min})$, so the exported functions take their arguments in different orders. Both baselines evaluate $\max(e - prec, e_{\min})$ correctly, but they pass the arguments in the same positional order on both sides. They therefore obtain different outputs and return `not_aligned` with confidence at least 0.99. TRANSFORMAL instead reads the exported binder order from the elaborated signatures (S_1) and replays inputs matched by role in both systems. Coq `FLT_exp (-10) 3 5` and Lean `FLT_exp 3 (-10) 5` both evaluate to 2. With this evidence, TRANSFORMAL reaches 95.35% aligned recall on this domain and falsely rejects only 6 of the 129 faithful pairs.

Lost implicit context causes false acceptances. Model checking and cryptography show the opposite failure. On the model-checking pairs, both baselines detect only 33.33% of the misaligned pairs, and Tool Call falsely accepts 8 of the 27 unfaithful translations. Their rationales typically argue that "every source locale instance can be represented in the target" after instantiating the target's generalized types. This argument ignores that the translation has dropped locale assumptions or generalized concrete Isabelle types, so the target admits instances that the source excludes. The

Table 6: Outcome breakdown of GPT-5.6 Sol by domain and source language. Correct, false rejections (FR), false acceptances (FA), and abstentions (Abst.) are pair counts, and abstentions include invalid outputs. Aligned and misaligned recall are percentages.

Subset	Method	Correct	FR	FA	Abst.	Aligned recall	Misaligned recall
Numerical computation	Raw LLM	96	28	2	30	55.81	88.89
	Tool Call	83	27	1	45	44.96	92.59
	TRANSFORMAL	148	6	2	0	95.35	92.59
Model checking	Raw LLM	38	2	6	16	82.86	33.33
	Tool Call	41	2	8	11	91.43	33.33
	TRANSFORMAL	58	1	3	0	97.14	88.89
Cryptography	Raw LLM	16	0	1	13	53.85	50.00
	Tool Call	18	0	2	10	61.54	50.00
	TRANSFORMAL	29	0	0	1	96.15	100.00
PL semantics & rewriting	Raw LLM	16	1	2	2	88.89	0.00
	Tool Call	17	1	2	1	94.44	0.00
	TRANSFORMAL	19	2	0	0	88.89	100.00
CS theory	Raw LLM	12	0	1	5	70.59	0.00
	Tool Call	13	0	1	4	76.47	0.00
	TRANSFORMAL	17	0	1	0	100.00	0.00
Smart contracts & algorithms	Raw LLM	0	0	0	10	0.00	0.00
	Tool Call	0	0	0	10	0.00	0.00
	TRANSFORMAL	9	0	0	1	88.89	100.00
Coq → Lean	Raw LLM	108	28	3	45	54.19	82.76
	Tool Call	96	27	2	59	45.81	86.21
	TRANSFORMAL	174	6	3	1	95.48	89.66
Isabelle → Lean	Raw LLM	70	3	9	31	74.68	32.35
	Tool Call	76	3	12	22	82.28	32.35
	TRANSFORMAL	106	3	3	1	94.94	91.18

difference is invisible if the target is read as a function with an extended input domain, but it becomes observable once such an instance is constructed. Listing 2 shows the same pattern in rewriting theory. The Isabelle class inherits `zero_neq_one` through `comm_semiring_1`, while Lean's `CommSemiring` admits the trivial semiring. TRANSFORMAL constructs a `poly_carrier` instance on the singleton ring `ZMod 1` in Lean, where $0 = 1$, and proves in Isabelle that the singleton type cannot satisfy the source class. With witnesses of this kind, TRANSFORMAL detects 88.89% of the model-checking misaligned pairs and 91.18% of the Isabelle misaligned pairs overall.

Definitions outside the static context cause abstentions. On smart contracts and algorithms (EIP20 and Huffman), both baselines abstain on every pair. The judged behavior depends on executable definitions, such as `Show` instances and finite-map renderers, whose bodies lie outside the static dependency context. TRANSFORMAL executes both sides instead. Listing 4 shows the counterexample for the EIP20 token-state printer. The two `Show` instances have the same structure, but the finite-map renderers they call print an empty map differently. On CS theory and PL semantics, the declarations are mostly theorem statements whose meaning can be read from their text, so the gap between the methods is smaller. The remaining TRANSFORMAL errors in these domains mostly attach a valid local observation to the wrong public interface.

Error profiles by source language. The failure patterns above explain why the baseline error profile reverses between the two source languages. On Coq, the baselines over-reject, with 54.19% and 45.81% aligned recall against 82.76% and 86.21% misaligned recall. On Isabelle, they over-accept, with 74.68% and 82.28% aligned recall against 32.35% misaligned recall for both. Coq translations tend to change surface representations while preserving meaning. Isabelle-to-Lean translations instead tend to keep surface syntax but lose implicit context, such as locale assumptions, inherited class premises, and HOL's underspecified `undefined`. Reading the surface of the declarations therefore misleads in opposite directions for the two languages. TRANSFORMAL stays balanced on both, with 95.48% and 89.66% aligned and misaligned recall on Coq and 94.94% and 91.18% on Isabelle.

Table 7: Repository translation accuracy (%) under each feedback signal. Top: initial accuracy and accuracy after one refinement round for each model. Bottom: accuracy of GPT-5.6 Sol over three refinement rounds. Round 0 is the initial translation. Purple bold marks the best feedback signal in each row or round.

After one round				
Model	**Initial**	**Compiler**	**LLM judge**	**TRANSFORMAL**
GPT-5.6 Sol	80.25	75.81	77.36	**84.18**
DeepSeek V4 Pro	38.91	36.96	38.84	**40.09**
GLM 5.3	21.92	19.18	20.05	**27.39**
Claude Opus 5	85.33	84.85	79.65	**88.71**
GPT-5.6 Sol by round				
Signal	**Round 0**	**Round 1**	**Round 2**	**Round 3**
Compiler	80.25	75.81	80.65	83.27
LLM judge	80.25	77.36	78.34	82.15
TRANSFORMAL	80.25	**84.18**	**87.30**	**92.48**

H REPOSITORY TRANSLATION DETAILS

This appendix gives the full protocol and results of the repository translation experiment in Section 5.3.

Protocol. For each of the four TF-TRANSLATION repositories (Coq-SKI, KA-FMP, VeriComp, and Abstract Rewriting), each agent first produces an initial Lean translation, using up to two repository-level passes with three module-level rounds each. Starting from this initial translation, the same agent then performs three rounds of refinement under one of three feedback signals. *Compiler feedback* consists of the Lean build and test diagnostics. *LLM judge feedback* consists of the per-declaration verdicts and rationales of the Raw LLM judge. TRANSFORMAL *feedback* consists of the TRANSFORMAL verdicts together with their verified counterexamples, that is, the matched source and target inputs and their diverging observations. The initial translation, the agent, and the per-round budget are identical across conditions. We do not use Round-trip + BEq as a feedback signal. In a label-blind pilot on 19 judging pairs, it imported only 2 targets into Coq and formed no cross-vocabulary obligation, so it yields no semantic verdict to feed back. After each round, TRANSFORMAL with GPT-5.6 Sol scores every indexed source declaration. Missing declarations, declarations that fail to compile, and declarations without a definitive verdict count as failures.

Results. Table 7 lists the values behind Figure 3. Many initial failures originate in compilation, where a single ill-formed module blocks all of its dependents. In KA-FMP, for example, a utility module that uses the reserved Lean keyword `prefix` as a binder fails to build and blocks twelve downstream modules. Compiler feedback is therefore a meaningful baseline, but it gives no signal on declarations that type-check yet alter the source semantics. After one round, TRANSFORMAL feedback improves all four models. It raises GLM 5.3 from 21.92% to 27.39%, GPT-5.6 Sol from 80.25% to 84.18%, Claude Opus 5 from 85.33% to 88.71%, and DeepSeek V4 Pro from 38.91% to 40.09%. Compiler feedback and LLM judge feedback instead lower accuracy below the initial level for every model. With GPT-5.6 Sol, for example, they reach 75.81% and 77.36%, and LLM judge feedback lowers Claude Opus 5 to 79.65%. Over three rounds with GPT-5.6 Sol, TRANSFORMAL feedback rises from 80.25% to 84.18%, 87.30%, and 92.48%. Compiler feedback first drops to 75.81% and recovers to 83.27% by the third round, and LLM judge feedback follows a similar trajectory and ends at 82.15%. Appendix J analyzes why the three signals behave differently.

I JUDGE FAILURE MODES IN DETAIL

This appendix expands the judge failure analysis of Section 5.4. On the 297 judging pairs, GPT-5.6 Sol with Raw LLM abstains on 76 pairs, falsely rejects 31 faithful translations, and falsely accepts 12 unfaithful ones. With Tool Call, the counts are 81, 30, and 14. TRANSFORMAL makes only 9 false rejections, 6 false acceptances, and 2 abstentions (Table 6). The baselines are also poorly calibrated.

Table 8: Stages of program-analysis tools in TRANSFORMAL, the information each stage supplies, and the failures it addresses. F1–F6 refer to the failure modes in Table 9.

Stage	Information supplied	Failure addressed
S_0: dependency context	Matched declarations and their dependency slices from compiler indexes	Meaning misread from missing context (F4, F5)
S_1: declaration signature	Elaborated types, premises, and domains of inputs and outputs	Positional binder pairing, added premises, lost domains, or changed sorts (F1, F2, F3)
S_2: candidate synthesis	Branch conditions and boundary values from symbolic execution	Divergence hidden in a branch
S_3: candidate validation	Compilation and replay of both harnesses in their proof assistants	Unchecked or malformed evidence and observations of the wrong declaration (F1–F6)

Table 9: Failure modes of the Raw LLM and Tool Call judges with GPT-5.6 Sol, one representative benchmark case for each, and the TRANSFORMAL stages that address them. The last column points to the full case.

Failure mode	Representative case	Baseline error	Stages	Case
F1: Positional binder correspondence	`FLT_exp`: Coq $(e_{\min}, prec, e)$ vs. Lean $(prec, e_{\min}, e)$	False reject	S_1, S_3	Lst. 1
F2: Interface extension read as input extension	`poly_carrier`: Lean admits the singleton ring `ZMod 1`, where $0 = 1$	False accept	S_1, S_3	Lst. 2
F3: Added premise, domain loss, or sort change	`Fbound`: fields of type `positive` and `N` become `Int`	False accept	S_1, S_3	Lst. 3
F4: Behavior outside the static context	`showTokenState`: empty maps render as `[]` vs. `{}`	Abstain	S_0, S_3	Lst. 4
F5: Helper-level difference taken as theorem-level	`Bfma_szero`: the NaN constructor of the carrier differs, the function does not	False reject	S_0, S_3	Lst. 5
F6: Unchecked or malformed evidence	`Zscale_scale`: witnesses are unchecked text	Unsupported claim	S_3	Lst. 6

Every definitive error of Raw LLM (43) and Tool Call (44) is reported with confidence of at least 0.86, and all but one are at least 0.90.

Table 9 lists the six failure modes, and the full cases at the end of this appendix show each of them on one benchmark pair. Table 8 lists the four stages of TRANSFORMAL, the information each stage supplies, and the failure modes it addresses.

Baseline failure modes. Most errors do not come from a failure to reason about the formulas themselves. The baselines typically compute the relevant values correctly but select the wrong cross-language correspondence *before* computing. They pair binders by position rather than by role (F1). They treat an extension of the target's interface as a harmless extension of its input domain (F2). They overlook premises, domains, or sorts that are visible only after elaboration (F3). They abstain when the relevant behavior is defined outside the static dependency context (F4). They take a helper-level difference to be a difference in the judged theorem (F5). Finally, nothing checks whether their claimed evidence is well formed (F6).

How TRANSFORMAL addresses them. TRANSFORMAL addresses these failure modes with deterministic tools instead of additional LLM reasoning. S_0 retrieves the dependency slices from the compiler indexes. S_1 reads the exported binder order, the inherited class and locale conditions, and the field types of each declaration from its elaborated signature. Inputs are therefore paired by role (F1), inherited conditions such as `zero_neq_one` become visible (F2), and refined field types that the target erases are exposed (F3). S_2 extracts branch equations and guard boundaries from dependencies and turns them into concrete inputs. S_3 executes both sides, which removes the need to read renderer bodies that lie outside the context (F4). It also ties each observation to the output of

the judged declaration rather than to a helper (F5), and it accepts a witness only after both harnesses compile (F6).

Full cases. The six cases below come from TF-JUDGE and use the GPT-5.6 Sol runs. For each case we give the benchmark label, the verdicts and confidences of both baselines with an excerpt of their rationale, and the TRANSFORMAL verdict. Each listing shows the source and target declarations, followed by the harnesses of the TRANSFORMAL witness, which compile in both systems. We shorten long harnesses where marked.

F1: `FLT_exp` (Flocq, labeled aligned). Flocq declares the section variables in the order $(e_{\min}, prec)$ and FloatSpec declares them as $(prec, e_{\min})$, so the exported functions take their first two arguments in opposite orders. Both baselines notice the reordering but still apply the two functions to the same positional arguments. Raw LLM evaluates `FLT_exp 2 3 10` to 7 in Coq and to 8 in Lean, and Tool Call obtains 0 and 10 on the input $(0, 10, 5)$. Both return `not_aligned`, with confidence 1.00 and 0.99. TRANSFORMAL reads the binder order from the compiler index and pairs the inputs by role. Both harnesses compute $\max(5-3, -10) = 2$, so TRANSFORMAL returns `aligned`.

Listing 1: F1 full case: `FLT_exp`. The TRANSFORMAL witness passes $e_{\min} = -10$, $prec = 3$, $e = 5$ in each system's binder order.

```
(* Source, Flocq src/Core/FLT.v *)
Variable emin prec : Z.
Definition FLT_exp e := Z.max (e - prec) emin.      (* exported as FLT_exp emin prec e *)

-- Target, FloatSpec src/Core/FLT.lean
variable (prec emin : Int) [Prec_gt_0 prec]
def FLT_exp (e : Int) : Int := max (e - prec) emin  -- exported as FLT_exp prec emin e

(* Transformal source harness (Coq) *)
Theorem flt_exp_observation : FLT_exp ((-10)%Z) (3%Z) (5%Z) = (2%Z).
Proof. reflexivity. Qed.

-- Transformal target harness (Lean)
theorem flt_exp_observation :
  FloatSpec.Core.FLT.FLT_exp (3 : Int) (-10 : Int) (5 : Int) = (2 : Int) := by rfl
```

F2: `poly_carrier` (Abstract Rewriting, labeled not aligned). The Isabelle class inherits `zero_neq_one` through `comm_semiring_1`, while Lean's `CommSemiring` admits the trivial semiring. Both baselines see this difference and still return `aligned`, with confidence 0.98 and 0.99. Raw LLM argues that it "only extends the target domain and does not alter behavior for represented source instances." The judged declaration is a class, so whether an instantiation is inhabited is its observation (Section 3), and an extra inhabitant is an ordinary counterexample. TRANSFORMAL constructs a `poly_carrier` instance on the singleton ring `ZMod 1` in Lean and proves in Isabelle that the singleton type cannot be a `poly_carrier`. It returns `not_aligned`.

Listing 2: F2 full case: `poly_carrier`. The singleton carrier is an instance in Lean but not in Isabelle.

```
(* Source, Abstract-Rewriting SN_Orders.thy *)
class poly_carrier = ordered_semiring_1 + comm_semiring_1

-- Target, AbstractRewriting/Impl/SN_Orders.lean
class poly_carrier (α : Type u) extends ordered_semiring_1 α, CommSemiring α

(* Transformal source harness (Isabelle): the singleton type is not a poly_carrier *)
theorem poly_carrier_singleton_observation:
  "OFCLASS(unit, SN_Orders.poly_carrier_class) \<Longrightarrow> False"
  (* proof shortened: unfold poly_carrier_class to zero_neq_one, then use x = y on unit *)

-- Transformal target harness (Lean): the singleton ring is a poly_carrier
noncomputable def singletonPolyCarrier : poly_carrier (ZMod 1) where
  toLE := ⟨fun _ _ => True⟩
  toLT := ⟨fun _ _ => False⟩
  -- shortened: the remaining order and monotonicity fields are each closed by simp
theorem polyCarrierSingletonObservation :
    Nonempty (poly_carrier (ZMod 1)) := Nonempty.intro singletonPolyCarrier
```

F3: `Fbound` (Pff, labeled not aligned). The Coq record encodes its admissible domain in the field types, since `vNum` is a `positive` and `dExp` is an `N`. The Lean structure erases both refinements to `Int`. Both baselines accept the translation with confidence 0.99. Raw LLM states that "Lean additionally permits nonpositive vNum and negative dExp values, but this target-domain extension does not lose or alter any represented source input." TRANSFORMAL reads the elaborated field types (S_1) and turns the difference into an observation. Coq proves that no `Fbound` has a nonpositive `vNum`, while Lean exhibits `Bound (-1) 0`. The same proposition is refutable in the source and provable in the target, so the pair is an ordinary counterexample. TRANSFORMAL returns `not_aligned`.

Listing 3: F3 full case: `Fbound`. The two harnesses prove opposite answers to the same question about `vNum`.

```
(* Source, Flocq src/Pff *)
Record Fbound : Set := Bound {vNum : positive; dExp : N}.

-- Target, FloatSpec
structure Fbound where
  vNum : Int
  dExp : Int

(* Transformal source harness (Coq) *)
Theorem Fbound_source_no_nonpositive_vNum :
  ~ exists b : Fbound, (Zpos (vNum b) <= 0)%Z.
Proof. intros [b Hb]. destruct b as [p e]. simpl in Hb. lia. Qed.

-- Transformal target harness (Lean)
theorem Fbound_target_has_nonpositive_vNum :
    ∃ b : Fbound, b.vNum ≤ 0 := by
  exact ⟨Bound (-1) 0, by decide⟩
```

F4: `showTokenState` (EIP20, labeled not aligned). The two `Show` instances have the same structure, but they call finite-map renderers whose bodies are not in the static dependency context. Both baselines abstain with `uncertain`, at confidence 0.99 and 0.98. Tool Call states that "the missing source helper semantics prevent a responsible universal judgment or a concrete two-sided counterexample." TRANSFORMAL does not need the renderer bodies. It builds the same empty token state in both systems and executes the two instances. Coq prints an empty map as `[]` and Lean prints it as `{}`, so TRANSFORMAL returns `not_aligned`.

Listing 4: F4 full case: `showTokenState`. Both harnesses run on the state with zero supply and empty maps.

```
(* Source, ConCert EIP20Token *)
Instance showTokenState : Show EIP20Token.State :=
{| show s := "State{total_supply: " ++ show s.(total_supply) ++ sep
             ++ "balances: " ++ show s.(balances) ++ sep
             ++ "allowances: " ++ show s.(allowances) ++ "}" |}.

-- Target, EIP20Base
instance showTokenState : Show State := ⟨fun state =>
  "State{total_supply: " ++ showValue state.total_supply ++ sep ++
  "balances: " ++ showMap state.balances ++ sep ++
  "allowances: " ++ showMap state.allowances ++ "}"⟩

(* Transformal source observation (Coq, harness shortened) *)
show (build_state 0%N FMap.empty FMap.empty)
  = "State{total_supply: 0, balances: [], allowances: []}"

-- Transformal target observation (Lean, harness shortened)
Show.render (build_state 0 FMap.empty FMap.empty)
  = "State{total_supply: 0, balances: {}, allowances: {}}"
```

F5: `Bfma_szero` (Flocq, labeled aligned). `Bfma_szero` computes the sign of a zero result of a fused multiply-add, and its Lean translation follows the Coq definition branch for branch. The two versions differ only in their shared carrier type. Coq's `B754_nan` is a nullary constructor, while Lean's carries a sign, a payload, and a validity proof. Both baselines reject the translation with confidence 1.00. They argue that at `prec = 1` the source NaN has no target representation, which is a property of the carrier rather than of `Bfma_szero`. TRANSFORMAL judges the Boolean output of the function itself. It replays matched inputs such as rounding down on -0, $-\infty$, and -0, obtains `true` on both sides, and returns `aligned`.

Listing 5: F5 full case: `Bfma_szero`. The function bodies agree, and the replayed outputs agree.

```
(* Source, Flocq src/IEEE754/BinarySingleNaN.v *)
Definition Bfma_szero m (x y z: binary_float) : bool :=
  let s_xy := xorb (Bsign x) (Bsign y) in
  if Bool.eqb s_xy (Bsign z) then s_xy
  else match m with mode_DN => true | _ => false end.

-- Target, FloatSpec
def Bfma_szero {prec emax : Int} (mode : RoundingMode)
    (x y z : binary_float prec emax) : Bool :=
  let sxy := Bool.xor (Bsign x) (Bsign y)
  if sxy == Bsign z then sxy
  else match mode with
    | RoundingMode.RTN => true
    | _ => false

(* Transformal source harness (Coq, implicit arguments abbreviated) *)
Theorem source_Bfma_szero_example :
  @Bfma_szero 3 10 mode_DN (B754_zero true) (B754_infinity true) (B754_zero true) = true.
Proof. reflexivity. Qed.

-- Transformal target harness (Lean, implicit arguments abbreviated)
theorem target_Bfma_szero_example :
    Binary.Bfma_szero (prec := 3) (emax := 10) RoundingMode.RTN
      (.B754_zero true) (.B754_infinity true) (.B754_zero true) = true := by rfl
```

F6: `Zscale_scale` (Flocq, labeled not aligned). The Coq theorem computes `Zscale` with the truncating `Z.quot`, while the frozen Lean target uses Euclidean `Int` division, so the scaled integers differ on negative inputs. Both baselines return the correct verdict with confidence 1.00. Their witnesses are unchecked text, however. Raw LLM claims -2 versus -3 for $n = -5$, and Tool Call claims 0 versus -1 for $n = -1$. Both claims happen to be correct here, but the same unchecked reasoning produces the confident errors in the other cases. TRANSFORMAL accepts a witness only after both harnesses compile. The Coq harness must supply a proof of the premise $0 \leq k$, and the Lean harness must discharge the two added divisibility premises, so a witness that violates a premise does not compile. With $n = -3$, $k = 1$, and $k' = -2$, both sides compute a self-consistent equality, but the scaled integer is -1 in Coq and -2 in Lean. TRANSFORMAL returns `not_aligned`.

Listing 6: F6 full case: `Zscale_scale`. Both harnesses must discharge the premises of the judged theorem before the witness counts.

```
(* Source, Flocq src/Core/Digits.v *)
Definition Zscale n k :=
  if Zle_bool 0 k then (n * Zpower beta k)%Z else Z.quot n (Zpower beta (-k)).
Theorem Zscale_scale :
  forall n k k', (0 <= k)%Z -> Zscale (Zscale n k) k' = Zscale n (k + k').

-- Target, FloatSpec src/Core/Digits.lean (frozen translation)
def Zscale (n k : Int) : Int :=
  (if 0 ≤ k then n * beta ^ k.natAbs else n / beta ^ (-k).natAbs)
theorem Zscale_scale (n k l : Int) (hβ : beta > 1 := h_beta)
    (hdiv_k : k < 0 → beta ^ (-k).natAbs ∣ n)
    (_hdiv_compose : k < 0 → l ≥ 0 → k + l < 0 → beta ^ l.natAbs ∣ n) : ...

(* Transformal source harness (Coq): the premise 0 <= k must be proved *)
Lemma source_hk : (0 <= 1)%Z.
Proof. lia. Qed.
Theorem source_zscale_scale_example :
  Zscale radix2 (Zscale radix2 (-3)%Z 1%Z) (-2)%Z = Zscale radix2 (-3)%Z (1 + (-2))%Z.
Proof. exact (Zscale_scale radix2 (-3)%Z 1%Z (-2)%Z source_hk). Qed.

-- Transformal target harness (Lean): both divisibility premises must be proved
theorem target_hdiv_k :
    (1 : Int) < 0 → (2 : Int) ^ (-(1 : Int)).natAbs ∣ (-3 : Int) := by
  intro h; omega
-- shortened: target_hdiv_compose is proved the same way, then Zscale_scale is applied
```

The remaining errors of TRANSFORMAL fall into three groups. All of them concern the choice of the matching relation $\equiv$ on inputs and observations (Section 3.1), which the proof assistants validate only locally and cannot certify.

Over-permissive matching relations. Most remaining false acceptances occur because the agent chooses a matching relation that explains a difference away. For `digits2_Pnat_correct`, it treats the target's one-based bit length as a representation shift of the source's zero-based index, citing downstream APIs that consistently apply the shift. For `iffT`, it regards Lean's generalization from `Type` to `Sort` as harmless. For `Bminus_correct`, it argues that the target "safely extends" the source to additional inputs. Under the chosen relation, every replayed observation is correct, and S_3 cannot tell that the relation itself is wrong.

Observer choice. Most remaining false rejections attach a valid local observation to the wrong public interface, the same failure as F5 but applied to a correct witness. For `round_N_le_midp` and `round_N_ge_midp`, the tie-breaking behavior of a helper again drives the verdict. In `Btrunc_correct`, two individually true propositions of different shape are taken as a divergence, although the target proves the substantive equality elsewhere in the repository. Similarly, in `TT_rename_eip20token`, the lengths of the renaming tables (8 vs. 7) differ only because the intended mapping sends both Coq `N0` and `O` to Lean `Nat.zero`.

Declaration pairing. For `split_bits` and `join_bits`, TRANSFORMAL rejects the paired Lean declaration, which takes natural-number widths while Coq's takes integers. It observes that a separate façade in the target repository exposes the exact integer interface. Whether the root declaration or the façade is the public counterpart is a policy question that the tools cannot settle.

Tool coverage. Tool coverage also limits TRANSFORMAL, although it causes no error in our runs. S_2 performs bounded symbolic inspection, not general path solving. When the target fixes the rounding mode through a type-class instance rather than a branch, as in `abs_round_le_generic` (Appendix E), S_2 returns no path or boundary input. TRANSFORMAL still judges this pair correctly because the agent finds the instance in the target source, so such cases depend on the agent rather than on S_2.

J TRANSLATION FAILURE MODES IN DETAIL

This appendix expands the translation failure analysis of Section 5.4. It explains why TRANSFORMAL feedback repairs translations that the other signals fail to repair, and why those signals can reduce accuracy (Figure 3). The three signals differ in what information they deliver to the translation agent.

Compiler feedback. Compiler feedback is precise but has limited scope. Its diagnostics are reported per module and are attached identically to every declaration in a failing module. In VeriComp, for example, all 40 findings repeat the same unknown-constant errors from one specification file. Such feedback resolves build failures, which account for a substantial share of the initial losses, but it offers no signal once a declaration type-checks. Edits aimed only at restoring the build can also weaken specifications or tests. In an earlier refinement study, for example, the repair agent deleted six imports from a test aggregator to avoid downstream build failures. This explains why compiler feedback first lowers accuracy and recovers only partially.

LLM judge feedback. LLM judge feedback is mostly not actionable. In the GPT-5.6 Sol runs, all 42 semantic findings on Coq-SKI and 34 of the 35 on VeriComp are `uncertain` verdicts that cite missing context, e.g., "the contract definition is not supplied." The remaining findings inherit the judge's false rejections discussed in Appendix I. Faced with these findings, the agent restates or rewrites declarations that were already faithful, which introduces regressions. This is the main cause of the accuracy drops under this signal.

TRANSFORMAL feedback. TRANSFORMAL feedback is localized and can be checked. Each `not_aligned` finding carries a matched input and two replayed observations, which identify the defect precisely and let the agent verify its repair against the same witness. In an earlier one-round refinement study driven by judge feedback, the agent repaired semantic defects that compile cleanly and are therefore invisible to the compiler. Examples include the `Prop`-valued `monotone` record, which was restored to `Type` together with its dependants, and six EIP20 extraction values and tables. After repair, re-judging also caught a table that remained incomplete (66 vs. 67 entries). Because

Table 10: Token use and cost per case with GPT-5.6 Sol on the 192-pair hard suite. Settings are cumulative as in Figure 4. Costs use API-equivalent prices. Dashes mark values that were not recorded, and $\geq$ marks lower bounds.

Setting	Tokens/case	Attempts/case	Compiler retries	Cost/case	Cost/correct
Raw LLM	140.3k	1	–	\$0.55	\$0.92
Counterexample generation	$\geq$72.7k	1.354	–	–	–
$+S_3$	$\geq$65.5k	–	47	–	–
Full TRANSFORMAL (S_0–S_3)	60.7k	1.151	41	\$0.24	\$0.25

each finding is backed by a concrete witness, the agent can also dismiss spurious findings instead of acting on them. In the same study, it rejected 33 of 192 findings as noise.

Even with TRANSFORMAL feedback, some defects persist. They point to open challenges for the translation agents rather than for the judge.

Representation-level defects. Some defects cannot be fixed by a local edit. In the earlier refinement study, UniMath's proof-relevant `hProp`, identity-path, and embedding semantics require a foundational change of representation. Coq's universe-polymorphic `Arity` requires explicit `ULift`/`PLift` bridges throughout the non-cumulative Lean hierarchy, and the agent did not converge on them within three module rounds. In KA-FMP, the local fix to `monotone` was correct, yet it broke a downstream module that expected a `Prop`.

Counterexample overfitting. The agent sometimes patches the reported witness but not the underlying semantics. In the earlier refinement study, after repair, `isdeceq_isdecsurj` exposes a concrete index but still collapses the source's path data, and `showTokenState` fixes the map delimiters but not the address printer.

Missing translations. Declarations that were never translated generate feedback that is too broad for a local repair. For example, 106 untranslated KA-FMP declarations remain unresolved.

These observations motivate the scoring protocol of Section 5. After each round, a fresh TRANSFORMAL run judges every source declaration, and only declarations that compile and satisfy CEF count toward the score.

K COST BREAKDOWN

This appendix details the cost analysis of Section 5.5. Table 10 reports token use, model attempts, and API-equivalent cost for Raw LLM and the cumulative TRANSFORMAL settings of Figure 4.

Context delivery. The methods deliver context differently. Raw LLM must receive the entire compacted dependency closure and the demonstrations in a single prompt. As a result, 99.1% of its 26.95M tokens are input tokens (about 139k per case), while it produces only about 1.3k output tokens per case. TRANSFORMAL instead retrieves context on demand. S_0 and S_1 return bounded dependency slices and compiler-elaborated interfaces rather than raw source text, so the model reads only what disambiguates the case.

Fewer wasted attempts. The token cost of each model attempt stays roughly constant at 51–58k across the cumulative settings. The savings therefore come from the number of attempts. Attempts per case fall from 1.354 with counterexample generation alone to 1.151 for the full TRANSFORMAL, and compiler-feedback retries fall from 47 with S_3 to 41. Witnesses grounded in compiler-exposed interfaces and symbolic branches are less often rejected by the two-sided validation, so fewer candidates need to be regenerated. Token counts do not include the local computation of compilation and symbolic evaluation, which grows with each added stage.

Choosing stages. The marginal benefit of the stages diminishes. Among the stages, two-sided validation (S_3) accounts for most of the gain, raising accuracy from 85.4% to 93.2%, whereas S_0

changes only one of 192 predictions while adding 0.26% tokens. Every stage contributes non-negatively, so lighter configurations such as S_3 or S_1–S_3 are a reasonable choice under tight compute budgets. The full pipeline is best reserved for the hardest, representation-changing repositories.

Proof cost of faithful specifications. Faithful specifications are not free for the downstream translation, because a correct specification can impose harder proof obligations than a silently weakened one. In earlier Coq-to-Lean translations of five repositories, 80.42% of declarations were faithful, but only 72.00% also satisfied the full compilation, proof, and trust requirements. Repositories such as UniMath Computability built successfully even though 45 of their declarations lost proof-relevant semantics. A proof completed over a changed specification does not solve the translation task, so we consider this cost acceptable. A paired statement-only versus proof-required study will quantify the additional proof effort.

L Repeated Runs

Since each judging instance has a binary verdict, one might suspect that the gaps in Table 2 partly reflect sampling variance rather than differences between methods. To test this, we ran three independent Raw LLM passes with GPT-5.6 Sol on a frozen, seed-selected subset of 58 pairs (51 `aligned`, 7 `not_aligned`), using the same prompt and demonstrations in every pass. Accuracy is 48.28% for pass@1, 51.72% for pass@2, and 51.72% for pass@3, and majority@3 returns to 48.28%. Additional samples therefore raise even the oracle pass@k by only 3.44 percentage points, and majority voting recovers nothing. The sets of correct and incorrect cases also barely change across passes. Forty-five of the 58 pairs (77.59%) receive the same verdict in all three passes, and 21 of them are wrong every time. Inspecting these unanimous errors shows that they are systematic rather than unlucky decoding: each pass chooses the same positional binder correspondence or the same representation-level observer, with near-identical confidence. The variation across repeated runs is an order of magnitude smaller than the 34.71-point gain of TRANSFORMAL over Raw LLM. The gain therefore stems from counterexample search rather than from sampling luck, and repeated sampling cannot close it.