# BEYOND EXECUTABILITY: SCHEMA-ADAPTIVE ALIGNMENT FOR FAITHFUL NATURAL-LANGUAGE-TO-SYMBOLIC REASONING

**Anonymous authors**
Paper under double-blind review

## ABSTRACT

Formalize-then-solve pipelines improve LLM reasoning by translating natural-language problems into symbolic artifacts and delegating inference to external solvers. However, we identify a hidden reliability gap: a generated artifact may be syntactically executable while still violating the schema-level commitments it declares. We call this failure mode *Silent Schema Inconsistency*. Such errors are especially dangerous because they do not necessarily trigger parser errors, solver crashes, or execution feedback, leaving execution-driven refinement structurally blind to the underlying semantic mismatch. To address this problem, we propose *Schema-Adaptive Constraint Alignment* (SACA), an oracle-free, feedback-free, and inference-free pre-execution alignment layer for LLM-generated symbolic artifacts. Instead of asking whether an artifact can run, SACA asks whether it obeys its own declared schema: it detects the artifact schema, extracts a schema-specific constraint closure, and deterministically aligns recoverable violations before solver execution. We instantiate SACA for rule-style and CSP-style artifacts, enabling predicate-declaration completion, predicate-signature alignment, and domain–ordinal normalization. Across ProntoQA, LogicalDeduction, and ProofWriter, SACA improves symbolic reasoning by repairing distinct schema-level inconsistencies while introducing zero additional LLM calls. On LogicalDeduction, SACA improves FARA-CF from 70.0% to 91.0% on the diagnostic subset and improves Logic-LM from 54.7% to 80.7% on the full 300-example development split. Moreover, 7B symbolic-execution systems equipped with SACA outperform a larger 14B prompt-only model on structurally constrained settings, while SACA itself adds less than 0.5 ms of latency per example. These results show that reliable neuro-symbolic reasoning requires not only executable artifacts, but also systematic pre-execution enforcement of the schemas those artifacts declare.

## 1 INTRODUCTION

Large language models (LLMs) have substantially advanced natural-language reasoning (Wei et al., 2022; Wang et al., 2022), yet purely textual reasoning remains brittle when tasks require exact symbolic grounding, constraint satisfaction, or rule-governed inference (Xu et al., 2024; Olausson et al., 2023; Mirzadeh et al., 2025). A growing line of work therefore adopts *formalize-then-solve* pipelines: an LLM first translates a natural-language problem into an executable symbolic artifact, and an external interpreter, runtime, or symbolic solver then performs the final inference (Gao et al., 2023; Chen et al., 2022; Pan et al., 2023; Olausson et al., 2023). This paradigm is attractive because it separates linguistic formalization from deterministic execution: the LLM proposes a symbolic representation, while the solver or interpreter enforces rule-based inference or computation (Gao et al., 2023; Pan et al., 2023; Olausson et al., 2023).

However, our study reveals a hidden failure behind this decomposition. A generated artifact may execute successfully while internally violating the structural commitments it has already made. We call this failure mode **Silent Schema Inconsistency**. For example, a CSP program may declare a local domain in which 1 denotes *newest* and 5 denotes *oldest*, but later encode the statement "the bus is the newest" as `bus == 5`. The solver receives a syntactically valid constraint and therefore executes it without complaint, even though the artifact is no longer reasoning under its own declared



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.

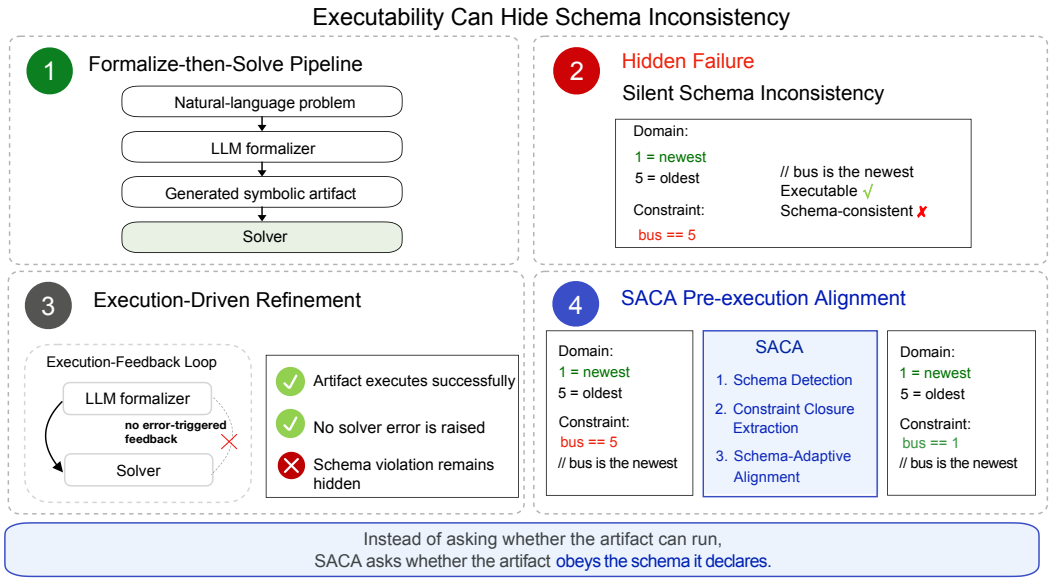



Figure 1: Motivation of SACA. A symbolic artifact may execute successfully while violating its own declared schema, making execution-driven refinement blind to the error. SACA therefore performs schema-adaptive alignment before solver execution.

coordinate system. Similar inconsistencies also arise in rule-style artifacts, where a rule sketch may invoke undeclared predicates or a rule program may use the same predicate with incompatible arities.

This failure exposes what we call the **Syntactic Executability Fallacy**: successful parsing or execution can be mistaken for evidence that an LLM-generated artifact is a valid symbolic representation. Representative formalize-then-solve systems use solver feedback to revise failed formalizations (Pan et al., 2023), while broader refinement and debugging methods rely on signals such as self-feedback, verbal reflection, execution outcomes, tests, or debugging information (Madaan et al., 2023; Shinn et al., 2023; Chen et al., 2024; Huang et al., 2024; Yang et al., 2025a). Yet execution only certifies that the artifact satisfies the solver's syntactic interface; it does not certify that the artifact is faithful to its own declared schema. When an artifact remains executable, these feedback signals need not expose an internal schema violation. The failure must therefore be addressed before execution, at the level of the artifact's own schema commitments.

We introduce **Schema-Adaptive Constraint Alignment (SACA)**, an oracle-free pre-execution alignment layer for LLM-generated symbolic artifacts. Given a generated artifact, SACA detects its symbolic schema, extracts a schema-specific constraint closure from explicit declarations and surface-level usage patterns, and aligns recoverable violations before solver execution. For rule-style artifacts, SACA completes missing predicate declarations and repairs predicate-signature inconsistencies. For CSP-style artifacts, SACA extracts the local domain–ordinal map and normalizes equality, comparison, and query constraints under that map. Crucially, SACA uses no gold labels, no predicted-answer correctness, no solver feedback, and no additional LLM calls. It is therefore a structure-driven reliability layer: it does not infer the correct answer, but enforces the schema-level commitments already exposed by the generated artifact.

We instantiate SACA as FARA-CF+, which applies SACA on top of our native FARA-CF formalizer, and further integrate the same module into Logic-LM outputs to test plug-in generality (Pan et al., 2023). Across rule-sketch, CSP, and rule-program settings, SACA repairs distinct schema-level failures: declaration incompleteness in ProntoQA (Saparov et al.), domain–ordinal mismatch in LogicalDeduction (Srivastava et al., 2022), and predicate-signature inconsistency in ProofWriter (Tafjord et al., 2021). On LogicalDeduction, SACA improves FARA-CF from 70.0% to 91.0% on the diagnostic subset, and improves Logic-LM from 54.7% to 80.7% on the full 300-example development split. Executable rate changes only marginally, while executable accuracy rises sharply, confirming that

SACA primarily repairs executable-but-internally-inconsistent artifacts rather than merely recovering non-executable programs.

Our contributions are summarized as follows:

- **Problem formulation.** We identify **Silent Schema Inconsistency** as a hidden failure mode in LLM-generated symbolic artifacts and expose the **Syntactic Executability Fallacy**: generated artifacts may execute successfully while violating their own declared schema commitments.
- **Method.** We propose Schema-Adaptive Constraint Alignment (SACA), an oracle-free, feedback-free, and inference-free pre-execution alignment layer that enforces an artifact's own schema-level commitments before solver execution. We instantiate SACA for rule-style and CSP-style artifacts, enabling declaration completion, predicate-signature alignment, and domain–ordinal normalization within a unified framework.
- **Evaluation.** We provide diagnostic experiments across rule-sketch, CSP, and rule-program settings, plug-in evaluation on Logic-LM, full-split verification, schema-aligner ablations, sample-consistency audits, and runtime analysis. Results show that SACA improves symbolic reliability with negligible overhead and zero additional LLM calls.

## 2 Related Work

**Formalize-then-solve and tool-augmented reasoning.** A growing line of work improves LLM reasoning by delegating computation or inference to external tools, interpreters, or symbolic solvers (Gao et al., 2023; Yao et al., 2022; Schick et al., 2023; Pan et al., 2023; Olausson et al., 2023). Representative approaches include program-aided reasoning, Program of Thoughts, Logic-LM, and LINC (Gao et al., 2023; Chen et al., 2022; Pan et al., 2023; Olausson et al., 2023), with related work extending this paradigm through code or pseudocode emulation and general neuro-symbolic frameworks (Li et al., 2023; **?**; Dinu et al., 2024). These methods establish the value of separating generation from execution, but executability does not guarantee that a generated artifact is consistent with the schema it declares. SACA targets this pre-execution gap: rather than introducing a new solver or formalizer, it addresses the **Syntactic Executability Fallacy**, where a runnable artifact can still violate its own symbolic commitments.

**Symbolic reasoning, logical formalization, and neuro-symbolic verification.** Structured-reasoning methods range from chain-of-thought and self-consistency (Wei et al., 2022; Wang et al., 2022) to symbolic prompting methods such as SymbCoT and Symbolic-Aided CoT (Xu et al., 2024; Nguyen et al., 2025). Natural-language-to-logic work studies formal representation construction through resources and systems such as FOLIO, fine-grained NL-to-FOL translation, and compositional FOL translation (Han et al., 2024; Yang et al., 2024; Ryu et al., 2025). Verification-oriented methods assess generated symbolic or reasoning content through symbolic-chain verification, FOL verification, or executable formalization (Xu et al., 2024; Ryu et al., 2025; Pei et al., 2025), while related LLM–prover and neuro-symbolic systems integrate symbolic provers, deduction engines, and formal verification mechanisms (Qi et al., 2025; Cao et al., 2025; Trinh et al., 2024; Yang et al., 2025b; 2026). SACA targets a different notion of validity: **Silent Schema Inconsistency**, where an artifact violates structural commitments declared by that same artifact. Its criterion is therefore derived from the artifact's own declarations and usage patterns rather than external correctness signals.

**Self-refinement, program repair, and SACA's position.** Self-refinement and program-repair methods use self-feedback, verbal reflection, execution results, tests, or debugging signals to revise generated outputs (Madaan et al., 2023; Shinn et al., 2023; Chen et al., 2024; Huang et al., 2024; Yang et al., 2025a). In formalize-then-solve reasoning, Logic-LM similarly uses solver error messages to refine unsuccessful symbolic formulations (Pan et al., 2023). Such feedback is effective when execution exposes a defect, but executable artifacts may contain schema violations that trigger neither parser/runtime failures nor solver diagnostics. SACA instead performs deterministic pre-execution alignment from the artifact's own declarations and usage patterns, requiring neither runtime feedback, LLM regeneration, nor additional LLM calls.

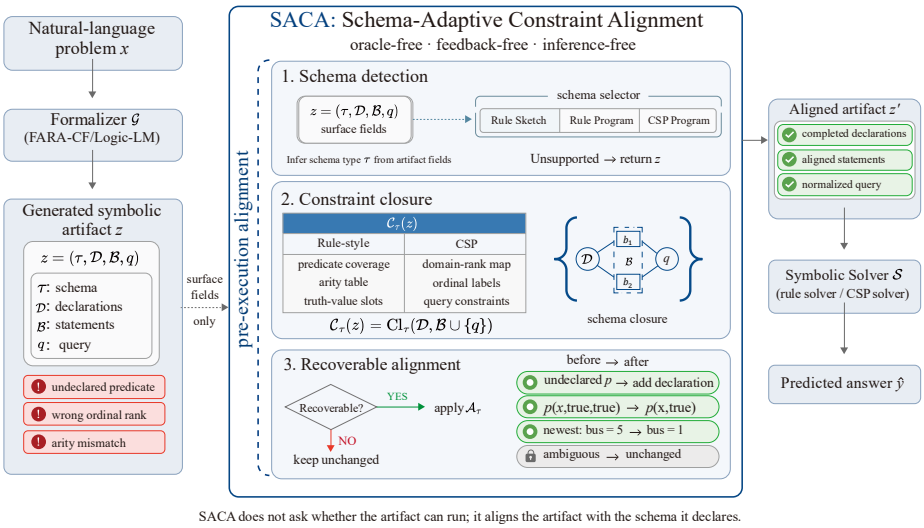



Figure 2: Overview of Schema-Adaptive Constraint Alignment (SACA). A formalizer $\mathcal{G}$ converts a natural-language problem $x$ into a symbolic artifact $z$. SACA detects its schema from surface fields, extracts an artifact-internal constraint closure, and applies only uniquely determined recoverable alignments. The aligned artifact $z'$ is then passed to the symbolic solver $\mathcal{S}$; execution remains outside SACA. The tuple $(\tau, \mathcal{D}, \mathcal{B}, q)$ denotes SACA's canonical structured view of $z$, with $\tau$ inferred rather than explicitly generated.

# 3 Method

## 3.1 Overview

We introduce Schema-Adaptive Constraint Alignment (SACA), an oracle-free pre-execution alignment layer for LLM-generated symbolic artifacts. Given a natural-language problem $x$, a formalizer $\mathcal{G}$ produces an artifact $z = \mathcal{G}(x)$. SACA detects its schema as $\tau = \mathrm{DetectSchema}(z) \in \mathcal{T} \cup \{\bot\}$, where $\mathcal{T}$ is the set of supported schemas and $\bot$ denotes an unsupported or ambiguous artifact. For supported $\tau$, a schema-conditioned parser gives $\mathrm{Parse}_\tau(z) = (\mathcal{D}, \mathcal{B}, q)$, where $\mathcal{D}$ contains structural declarations, $\mathcal{B}$ contains generated symbolic statements, and $q$ is the query or answer condition. For correspondence with Figure 2, we call $(\tau, \mathcal{D}, \mathcal{B}, q)$ the canonical SACA view of $z$; $\tau$ is inferred from surface fields rather than explicitly generated by $\mathcal{G}$.

The overall pipeline is

$$x \xrightarrow{\mathcal{G}} z \xrightarrow{\Phi_{\mathrm{SACA}}} z' \xrightarrow{\mathcal{S}} \hat{y}, \tag{1}$$

where $\Phi_{\mathrm{SACA}}$ is the complete SACA transformation, $z'$ is the aligned artifact, $\mathcal{S}$ is the downstream symbolic solver, and $\hat{y}$ is the predicted answer. SACA reads only $z$ and uses no gold labels, predicted-answer correctness, solver success, solver errors, or solver-derived feedback.

Figure 2 summarizes SACA's three deterministic stages: schema detection, constraint-closure extraction, and recoverable alignment. Our main instantiation, FARA-CF+, applies SACA to FARA-CF outputs; we also apply the same module to Logic-LM (Pan et al., 2023). Implementation-level details, diagnostic traces, and the conservative fallback protocol are provided in Appendix A.

## 3.2 Schema Constraint Closure

The declarations $\mathcal{D}$ encode the artifact's own structural commitments. For rule-style artifacts, these include predicate declarations, argument signatures, and truth-value slots; for CSP artifacts, they include finite domains, variable declarations, and ordinal-to-rank mappings. SACA combines these declarations with observable statement and query usage to construct

$$\mathcal{C}_\tau(z) = \mathrm{Cl}_\tau\big(\mathcal{D}, \mathcal{B} \cup \{q\}\big), \tag{2}$$

where $\mathcal{C}_\tau(z)$ is the schema-specific constraint closure and $\mathrm{Cl}_\tau$ is its extraction operator. For rule-style artifacts, the closure records predicate coverage, arities, and truth-value positions; for CSP artifacts, it records domain–rank mappings, ordinal annotations, comparison directions, and query constraints.

Let $\mathrm{Comp}_\tau(s, \mathcal{C}_\tau(z)) \in \{0, 1\}$ denote whether a statement or query condition $s$ is structurally compliant with the closure. The violation set is

$$\mathcal{V}_\tau(z) = \left\{ s \in \mathcal{B} \cup \{q\} \;\middle|\; \mathrm{Comp}_\tau\big(s, \mathcal{C}_\tau(z)\big) = 0 \right\}. \tag{3}$$

Thus, $\mathcal{V}_\tau(z)$ captures schema violations relative to the artifact's own commitments, not an external gold program or solver outcome.

### 3.3 Recoverable Schema Alignment

For each violation $v \in \mathcal{V}_\tau(z)$, let $\mathcal{R}_\tau(v; z, \mathcal{C}_\tau(z))$ be the set of schema-permitted artifact edits that resolve $v$ and are justified by the closure. An edit may modify the complete artifact structure: it can add or change a declaration in $\mathcal{D}$, rewrite a statement in $\mathcal{B}$, or normalize $q$.

SACA treats a violation as recoverable only when exactly one repair is supported:

$$\mathcal{V}_\tau^{\mathrm{rec}}(z) = \left\{ v \in \mathcal{V}_\tau(z) \;\middle|\; \left|\mathcal{R}_\tau\big(v; z, \mathcal{C}_\tau(z)\big)\right| = 1 \right\}, \tag{4}$$

where $|\cdot|$ denotes set cardinality. For each $v \in \mathcal{V}_\tau^{\mathrm{rec}}(z)$, let $r_v$ be its unique repair. Repairs prescribing incompatible writes to the same declaration slot, statement field, or query attribute are conservatively withheld. Let $\mathcal{E}_\tau(z)$ denote the remaining conflict-free repairs. They are applied in a fixed schema-specific order; for edits on disjoint artifact locations, this order is immaterial.

The complete transformation is

$$z' = \Phi_{\mathrm{SACA}}(z) = \begin{cases} z, & \mathrm{DetectSchema}(z) = \bot, \\ \mathcal{A}_\tau\big(z, \mathcal{E}_\tau(z)\big), & \tau = \mathrm{DetectSchema}(z) \in \mathcal{T}, \end{cases} \tag{5}$$

where $\mathcal{A}_\tau$ is the schema-specific alignment operator. Thus unsupported schemas, non-unique repairs, and conflicting repairs are left unchanged. Because $\mathcal{A}_\tau$ acts on the complete artifact, a violation observed in $\mathcal{B}$ may, for example, be repaired by adding a declaration to $\mathcal{D}$.

### 3.4 Schema Instantiations

**Rule-sketch declaration completion.** SACA compares predicates declared in $\mathcal{D}$ with those used in $\mathcal{B}$ and $q$. A consistently used but undeclared predicate is added only when its declaration is uniquely determined by observed usage, repairing render-blocking omissions before compilation.

**Rule-program signature alignment.** For rendered rule programs, SACA builds a predicate symbol table from declarations and observed calls and aligns arity or truth-value-slot inconsistencies only when the closure determines a unique repair.

**CSP domain–ordinal alignment.** Let $\mathcal{O}$ be the ordinal descriptions and $\mathcal{R}$ the numeric ranks exposed in an artifact's `Domain` fields. SACA extracts an artifact-local map $m_z : \mathcal{O} \to \mathcal{R}$. Thus, if $1 =$ *newest* and $5 =$ *oldest*, then $m_z(\textit{newest}) = 1$ and $m_z(\textit{oldest}) = 5$. The ordinal information is explicitly exposed inside $z$ through generated tags or inline comments, e.g., `newest: bus == 5`, rather than recovered from the original problem $x$. SACA uses $m_z$ to normalize exposed equality, comparison, and query constraints; in this example, `bus == 5` is rewritten as `bus == 1`. Missing or ambiguous ordinal evidence is left unchanged.

Concrete rule-schema and CSP-schema alignment rules are detailed in Appendix A.

**Reliability boundary.** SACA is oracle-free, feedback-free, and inference-free: it uses no gold labels, predicted-answer correctness, solver-derived feedback, or additional LLM calls. It applies to artifact-producing formalize-then-solve systems whose outputs expose sufficient structure for schema detection and uniquely determined alignment. The corresponding no-refinement and zero-leakage provenance is documented in Appendix E.

| Dataset | Split | $N$ | Avg. ctx. | Avg. opt. | $K$ | $\tau$ |
|---|---|---|---|---|---|---|
| ProntoQA | local | 500 | 67.9 | 2.0 | 2 | RS |
| LogicalDed. | local | 100 | 71.3 | 5.0 | 5 | CSP |
| ProofWriter | local | 100 | 119.2 | 3.0 | 3 | RP |

Table 1: Dataset statistics for the diagnostic evaluation. $N$ denotes the number of examples, Avg. ctx. is the average number of context words, Avg. opt. is the average number of answer options, and $K$ is the task-visible answer cardinality. $\tau$ denotes the artifact schema: RS = rule sketch, CSP = constraint-satisfaction program, and RP = rule program.

# 4 EXPERIMENTS

We evaluate Schema-Adaptive Constraint Alignment (SACA) as a plug-in pre-execution alignment layer for LLM-generated symbolic artifacts. Our main instantiation, FARA-CF+, applies SACA on top of the FARA-CF formalizer. To test whether the same alignment layer generalizes beyond this specific formalizer, we further integrate SACA into Logic-LM, an external formalize-then-solve pipeline that generates solver-compatible symbolic programs (Pan et al., 2023).

Our evaluation addresses four questions: (i) whether SACA improves the main FARA-CF+ instantiation across different artifact schemas; (ii) whether the gains arise from schema-specific alignment rather than generic rewriting; (iii) whether SACA improves an external formalizer such as Logic-LM and persists beyond a small diagnostic subset; and (iv) whether 7B symbolic-execution systems equipped with SACA can outperform a larger 14B prompt-only model on structurally constrained reasoning tasks.

## 4.1 EXPERIMENTAL SETUP

We evaluate on three symbolic reasoning datasets that stress complementary artifact schemas: ProntoQA (Saparov et al.), LogicalDeduction (Srivastava et al., 2022), and ProofWriter (Tafjord et al., 2021). ProntoQA contains compact rule-chaining problems with binary answers and tests whether rule sketches expose complete predicate declarations. LogicalDeduction contains ordering and ranking problems with multiple-choice answers and stresses whether CSP programs consistently map ordinal descriptions to local numeric domains. ProofWriter contains fact-and-rule reasoning problems with three-way labels and tests whether generated rule programs maintain consistent predicate signatures. Together, they cover rule-sketch declaration completeness, CSP domain–ordinal consistency, and rule-program symbol consistency.

Table 1 summarizes the diagnostic split size, input length, answer cardinality, and artifact schema for each dataset.

These schemas activate different SACA targets: declaration completion for ProntoQA, domain–ordinal normalization for LogicalDeduction, and predicate-signature alignment for ProofWriter.

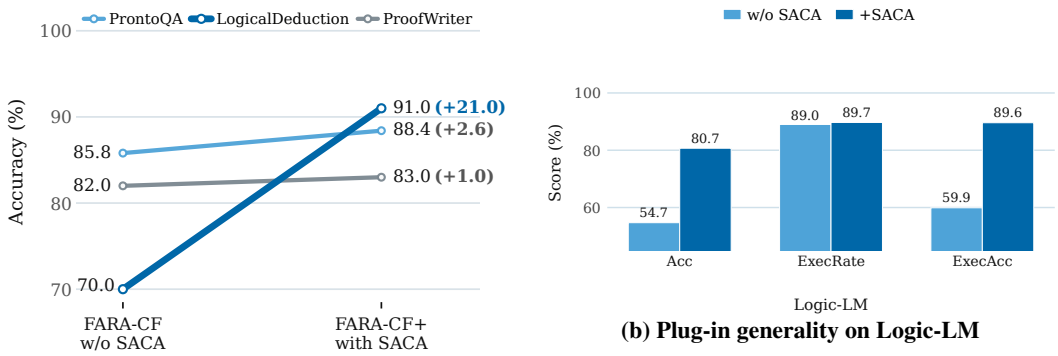
We use fixed diagnostic subsets of 500 ProntoQA examples and 100 examples each for LogicalDeduction and ProofWriter. To verify that CSP-alignment gains are not confined to the diagnostic subset, we additionally evaluate LogicalDeduction on its full 300-example development split. Dataset construction, fixed evaluation subsets, and additional schema-level details are provided in Appendix B. Baseline adaptation, decoding settings, and artifact provenance are documented in Appendix C.

## 4.2 MAIN DIAGNOSTIC RESULTS

Table 2 reports answer accuracy on the fixed diagnostic subsets. FARA-CF+ achieves the best or tied-best performance on all three settings. On LogicalDeduction, the strongest 7B prompt-only baseline reaches only 48.0%, while FARA-CF+ reaches 91.0%, yielding a 43.0-point absolute gain. On ProofWriter, FARA-CF+ reaches 83.0%, outperforming the strongest 7B prompt-only baseline by 17.0 points and matching Logic-LM. On ProntoQA, where compact binary rule chains make prompt-only reasoning more competitive, FARA-CF+ still improves over the strongest 7B prompt-only baseline by 3.8 points.

| Method | PQA | LD | PW |
|---|---|---|---|
| Direct | 51.6 | 41.0 | 45.0 |
| CoT (Wei et al., 2022) | 74.8 | 47.0 | 46.0 |
| SC (Wang et al., 2022) | 81.0 | 45.0 | 62.0 |
| SymbCoT (Xu et al., 2024) | 84.6 | 36.0 | 53.0 |
| SA-CoT (Nguyen et al., 2025) | 83.2 | 48.0 | 66.0 |
| Logic-LM (Pan et al., 2023) | 81.2 | 74.0 | **83.0** |
| FARA-CF | 85.8 | 70.0 | 82.0 |
| FARA-CF+ | **88.4** | **91.0** | **83.0** |

Table 2: Main diagnostic results. PQA, LD, and PW denote ProntoQA, LogicalDeduction, and ProofWriter. Values are answer accuracy (%).




**(a) Controlled effect on FARA-CF**

**(b) Plug-in generality on Logic-LM**

Figure 3: **Controlled effect and plug-in generality of SACA.** (a) Accuracy before and after applying SACA to FARA-CF across the three diagnostic settings. (b) Full LogicalDeduction development results using Logic-LM as an external formalizer. SACA substantially improves answer accuracy and executable accuracy, while execution rate changes only marginally.

To isolate the effect of SACA, Figure 3(a) compares FARA-CF with FARA-CF+. On ProntoQA, SACA improves accuracy from 85.8% to 88.4%, reflecting recovery from render-blocking rule-sketch omissions. On LogicalDeduction, accuracy rises sharply from 70.0% to 91.0% even though the artifacts are already executable, providing the central empirical signature of silent schema inconsistency: the programs can run, but their internal domain–ordinal semantics remain misaligned. On ProofWriter, SACA improves accuracy from 82.0% to 83.0%, indicating conservative behavior on already stable rule programs. All diagnostic comparisons use matched example IDs; the corresponding sample-consistency audit is provided in Appendix D.

### 4.3 Mechanism and Schema-Specificity

SACA repairs different schema-level inconsistencies depending on the detected artifact schema: missing predicate declarations in ProntoQA, domain–ordinal mismatches in LogicalDeduction, and predicate-signature inconsistencies in ProofWriter. The repair statistics support this interpretation: SACA repairs 35/500 ProntoQA examples, 33/100 LogicalDeduction examples, and 1/100 ProofWriter examples. Schema-aligner ablations further confirm that the gains are schema-specific: disabling rule alignment affects rule-style settings, while disabling CSP alignment removes the LogicalDeduction gains. Detailed activation counts, win–loss patterns, ablation results, and runtime measurements are reported in Appendix F; see Tables 5–8.

### 4.4 Plug-in Generality on an External Formalizer

We next test whether SACA is tied to FARA-CF or can also improve an external symbolic formalizer. Figure 3(b) reports full 300-example LogicalDeduction development results using Logic-LM (Pan et al., 2023) as the formalizer. SACA improves Logic-LM accuracy from 54.7% to 80.7%, while

| Method | PQA | LD | PW |
|---|---|---|---|
| 14B Direct | 66.0 | 65.0 | 45.0 |
| 14B CoT (Wei et al., 2022) | **93.8** | 83.0 | 67.0 |
| 14B SymbCoT (Xu et al., 2024) | 84.6 | 81.0 | 69.0 |
| 14B SA-CoT (Nguyen et al., 2025) | 83.2 | 77.0 | 60.0 |
| 7B FARA-CF+ | 88.4 | 91.0 | **83.0** |
| 7B Logic-LM+SACA | 81.2 | **98.0** | **83.0** |

Table 3: Model scale comparison. PQA, LD, and PW denote ProntoQA, LogicalDeduction, and ProofWriter; values are accuracy (%). For LD and PW, results are reported on the matched diagnostic-100 split.

EXECRATE changes only marginally from 89.0% to 89.7%. More importantly, EXECACC rises from 59.9% to 89.6%, showing that SACA primarily fixes executable-but-internally-inconsistent CSP artifacts rather than merely recovering non-executable programs.

These results support the claim that SACA is a formalizer-agnostic consistency layer: it can be inserted after an external formalize-then-solve pipeline as long as the generated symbolic artifacts expose recoverable schema constraints. Together with the controlled FARA-CF results in Figure 3(a), this shows that SACA improves both our native formalizer and an independently implemented symbolic formalizer, without relying on solver feedback or answer labels. The no-refinement and zero-leakage provenance for these experiments is documented in Appendix E. We further evaluate Logic-LM+SACA on the matched diagnostic-100 split in the scale-comparison analysis, where the same plug-in layer yields an even larger gain under the diagnostic CSP setting.

### 4.5 MODEL SCALE VS. SYMBOLIC EXECUTION

We compare 7B symbolic-execution systems equipped with SACA against Qwen2.5-14B-Instruct (Team, 2024) under prompt-only reasoning. For LogicalDeduction and ProofWriter, all 14B results are evaluated on verified diagnostic-100 artifacts exactly matched to the example IDs used by the corresponding 7B symbolic-execution methods. This diagnostic-100 setting is intentionally narrower than the full 300-example LogicalDeduction development split; hence Logic-LM+SACA reaches 98.0% here, compared with 80.7% on the broader full split. The prefix-100 provenance and sample-matching audit are documented in Appendices C and D.

The 14B prompt-only model performs strongly on ProntoQA, where CoT reaches 93.8%. This is expected because ProntoQA contains compact binary rule chains that a larger model can often solve directly from the prompt. However, the pattern changes on more structurally constrained settings. On LogicalDeduction, the strongest 14B prompt-only baseline reaches 83.0%, while 7B FARA-CF+ reaches 91.0% and 7B Logic-LM+SACA reaches 98.0%. Compared with the 74.0% Logic-LM baseline in Table 2, this corresponds to a 24.0-point absolute gain from adding SACA to the external formalizer on the same diagnostic split. On ProofWriter, the strongest 14B prompt-only result is 69.0%, while both 7B symbolic-execution systems with SACA reach 83.0%. Notably, Logic-LM+SACA leaves the ProntoQA and ProofWriter scores unchanged relative to Logic-LM in Table 2, reflecting the conservative behavior of SACA: when the generated artifacts already satisfy the detected schema constraints, the aligner does not introduce unnecessary modifications. Thus, model scale and symbolic execution provide complementary reasoning capacity: larger prompt-only models help on compact rule chains, but executable artifacts plus schema alignment dominate on rigid symbolic constraints.

### 4.6 CASE STUDY

Figure 4 illustrates how SACA repairs symbolic artifacts before solver execution. Across rule sketches, CSP programs, and rule programs, each modification is justified by constraints already present in the artifact: missing predicate declarations, domain–ordinal maps, or predicate signatures. For example, in LogicalDeduction, an artifact declares $1 =$ *newest* and $5 =$ *oldest* but encodes "the bus is the newest" as `bus == 5`; SACA rewrites it to `bus == 1`. This shows that SACA is not

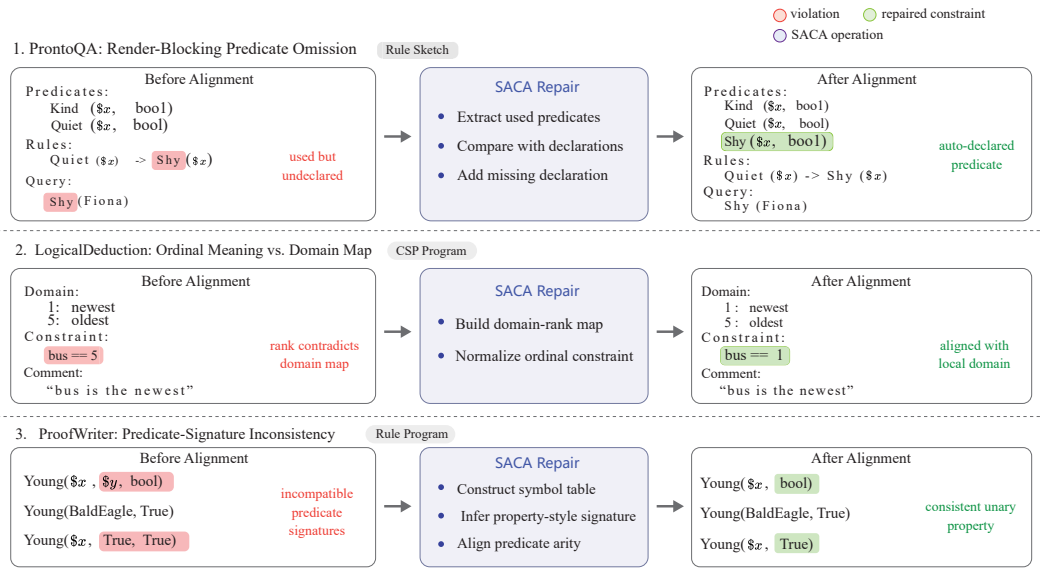



Figure 4: Case-level behavior of SACA across three symbolic schemas. SACA aligns artifacts with their own declared structure without using gold answers, predicted-answer correctness, or solver-derived target labels.

answer-level correction, but pre-execution enforcement of the artifact's own schema commitments. Additional case-level descriptions are provided in Appendix F.5.

### 4.7 Runtime Overhead

SACA introduces no additional LLM calls and is computationally lightweight. Measured as in-memory alignment time over existing pre-SACA artifacts, excluding LLM generation, solver inference, evaluation, and artifact I/O, SACA adds less than 0.5 ms per example across all settings. Full runtime measurements are reported in Appendix F.4 and Table 8.

## 5 Conclusion

We introduced Schema-Adaptive Constraint Alignment (SACA), an oracle-free pre-execution alignment layer for LLM-generated symbolic artifacts. SACA addresses a hidden failure mode in formalize-then-solve reasoning: a generated artifact may be syntactically executable while still contradicting its own declared schema. By detecting the artifact schema, extracting a normative constraint closure, and aligning recoverable violations before solver execution, SACA improves symbolic reliability without gold labels, solver feedback, answer correctness signals, or additional LLM calls.

Instantiated as FARA-CF+, SACA improves reasoning across rule-sketch, CSP, and rule-program settings through declaration completion, domain–ordinal alignment, and predicate-signature repair. Applying the same module to Logic-LM outputs further shows that SACA functions as a plug-in consistency layer for solver-compatible artifacts rather than a formalizer-specific patch. Our full-split verification, schema-aligner ablations, and runtime analysis confirm that SACA performs targeted internal-consistency alignment with negligible overhead.

More broadly, our results suggest that reliable neuro-symbolic reasoning requires more than executable programs: generated artifacts must also obey their own schema-level commitments. By enforcing this principle before execution, SACA offers a practical reliability and parameter-efficiency mechanism for LLM-based symbolic reasoning.

## AI USE STATEMENT

Generative AI tools were used to assist with research ideation and hypothesis refinement; feedback on methodology and experimental design; code and software drafting, editing, and debugging; experimental-workflow support; organization and interpretation of results; and manuscript structuring, drafting, editing, citation checking, and submission preparation. The tools did not replace author review, experimental verification, or scientific judgment. The authors reviewed and verified all AI-assisted code, experimental procedures, outputs, citations, interpretations, claims, and manuscript text, and take responsibility for the final content of this work.

## REPRODUCIBILITY STATEMENT

Section 3 specifies the SACA formulation, including schema detection, schema-specific constraint-closure construction, recoverable alignment, conservative fallback behavior, and the rule-style and CSP-style schema instantiations. Section 4 documents the evaluation datasets, fixed diagnostic subsets, baseline systems, the FARA-CF/FARA-CF+ comparison, plug-in evaluation with Logic-LM, full-split verification, model-scale comparison, case studies, and runtime analysis.

Appendix A provides schema-specific alignment rules, diagnostics, recoverability criteria, conflict handling, and the operational boundary. Appendix B documents dataset construction and fixed evaluation subsets. Appendix C specifies baseline adaptation, decoding settings, and artifact provenance. Appendix D reports the sample-consistency audit and verifies matched example IDs across reported comparisons. Appendix E documents the no-refinement and zero-leakage protocol. Appendix F provides activation statistics, win–loss analysis, schema-aligner ablations, runtime measurements, and additional case-level details.

For each diagnostic setting, the evaluation retains generated symbolic artifacts, solver outputs, SACA diagnostic traces, repair counts, and before–after artifact changes required for artifact-level verification. All SACA experiments operate under the stated pre-execution boundary: repairs are determined from the generated artifact itself and do not use gold labels, predicted-answer correctness, solver-derived feedback, inference traces, or additional LLM calls.

## REFERENCES


Chuxue Cao, Mengze Li, Juntao Dai, Jinluan Yang, Zijian Zhao, Shengyu Zhang, Weijie Shi, Chengzhong Liu, Sirui Han, and Yike Guo. Towards advanced mathematical reasoning for LLMs via first-order logic theorem proving. In Christos Christodoulopoulos, Tanmoy Chakraborty, Carolyn Rose, and Violet Peng (eds.), *Proceedings of the 2025 Conference on Empirical Methods in Natural Language Processing*, pp. 12429–12449, Suzhou, China, November 2025. Association for Computational Linguistics. ISBN 979-8-89176-332-6. doi: 10.18653/v1/2025.emnlp-main.628. URL https://aclanthology.org/2025.emnlp-main.628/.

Wenhu Chen, Xueguang Ma, Xinyi Wang, and William W Cohen. Program of thoughts prompting: Disentangling computation from reasoning for numerical reasoning tasks. *arXiv preprint arXiv:2211.12588*, 2022.

Xinyun Chen, Maxwell Lin, Nathanael Schärli, and Denny Zhou. Teaching large language models to self-debug. In *International Conference on Learning Representations*, volume 2024, pp. 8746–8825, 2024.

Marius-Constantin Dinu, Claudiu Leoveanu-Condrei, Markus Holzleitner, Werner Zellinger, and Sepp Hochreiter. Symbolicai: A framework for logic-based approaches combining generative models and solvers. *arXiv preprint arXiv:2402.00854*, 2024.

Luyu Gao, Aman Madaan, Shuyan Zhou, Uri Alon, Pengfei Liu, Yiming Yang, Jamie Callan, and Graham Neubig. Pal: Program-aided language models. In *International conference on machine learning*, pp. 10764–10799. PMLR, 2023.

Simeng Han, Hailey Schoelkopf, Yilun Zhao, Zhenting Qi, Martin Riddell, Wenfei Zhou, James Coady, David Peng, Yujie Qiao, Luke Benson, et al. Folio: Natural language reasoning with first-order logic. In *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing*, pp. 22017–22031, 2024.

Kai Huang, Zhengzi Xu, Su Yang, Hongyu Sun, Xuejun Li, Zheng Yan, and Yuqing Zhang. Evolving paradigms in automated program repair: Taxonomy, challenges, and opportunities. *ACM Computing Surveys*, 57(2):1–43, 2024.

Chengshu Li, Jacky Liang, Andy Zeng, Xinyun Chen, Karol Hausman, Dorsa Sadigh, Sergey Levine, Li Fei-Fei, Fei Xia, and Brian Ichter. Chain of code: Reasoning with a language model-augmented code emulator. *arXiv preprint arXiv:2312.04474*, 2023.

Aman Madaan, Niket Tandon, Prakhar Gupta, Skyler Hallinan, Luyu Gao, Sarah Wiegreffe, Uri Alon, Nouha Dziri, Shrimai Prabhumoye, Yiming Yang, et al. Self-refine: Iterative refinement with self-feedback. *Advances in neural information processing systems*, 36:46534–46594, 2023.

Iman Mirzadeh, Keivan Alizadeh-Vahid, Hooman Shahrokhi, Oncel Tuzel, Samy Bengio, and Mehrdad Farajtabar. Gsm-symbolic: Understanding the limitations of mathematical reasoning in large language models. In *International Conference on Learning Representations*, volume 2025, pp. 94743–94765, 2025.

Phuong Minh Nguyen, Tien Huu Dang, and Naoya Inoue. Non-interactive symbolic-aided chain-of-thought for logical reasoning. In Chu-Ren Huang, Yasunari Harada, Jong-Bok Kim, Nguyen T.M. Huyen, Le Thanh Huong, Pham Hien, Emmanuele Chersoni, Le Minh Nguyen, Rachel Edita Oñate Roxas, and Sherly Dita (eds.), *Proceedings of the 39th Pacific Asia Conference on Language, Information and Computation*, pp. 329–340, Hanoi, Vietnam, December 2025. Association for Computational Linguistics. URL https://aclanthology.org/2025.paclic-1.29/.

Theo Olausson, Alex Gu, Ben Lipkin, Cedegao Zhang, Armando Solar-Lezama, Joshua Tenenbaum, and Roger Levy. Linc: A neurosymbolic approach for logical reasoning by combining language models with first-order logic provers. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, pp. 5153–5176, 2023.

Liangming Pan, Alon Albalak, Xinyi Wang, and William Wang. Logic-lm: Empowering large language models with symbolic solvers for faithful logical reasoning. In *Findings of the Association for Computational Linguistics: EMNLP 2023*, pp. 3806–3824, 2023.

Yu Pei, Yongping Du, and Xingnan Jin. Fover: First-order logic verification for natural language reasoning. *Transactions of the Association for Computational Linguistics*, 13:1340–1359, 2025.

Chengwen Qi, Ren Ma, Bowen Li, He Du, Binyuan Hui, Jinwang Wu, Yuanjun Laili, and Conghui He. Large language models meet symbolic provers for logical reasoning evaluation. In Y. Yue, A. Garg, N. Peng, F. Sha, and R. Yu (eds.), *International Conference on Learning Representations*, volume 2025, pp. 54186–54205, 2025. URL https://proceedings.iclr.cc/paper_files/paper/2025/file/862819c227b16f9af64dd6ad6cfdf275-Paper-Conference.pdf.

Hyun Ryu, Gyeongman Kim, Hyemin S Lee, and Eunho Yang. Divide and translate: Compositional first-order logic translation and verification for complex logical reasoning. In *International Conference on Learning Representations*, volume 2025, pp. 24935–24964, 2025.

Abulhair Saparov, Richard Yuanzhe Pang, Vishakh Padmakumar, Nitish Joshi, Seyed Mehran Kazemi, Najoung Kim, and He He. Testing the general deductive reasoning capacity of large language models using ood examples (2023). *URL https://arxiv. org/abs/2305.15269*.

Timo Schick, Jane Dwivedi-Yu, Roberto Dessì, Roberta Raileanu, Maria Lomeli, Eric Hambro, Luke Zettlemoyer, Nicola Cancedda, and Thomas Scialom. Toolformer: Language models can teach themselves to use tools. *Advances in neural information processing systems*, 36:68539–68551, 2023.

Noah Shinn, Federico Cassano, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. Reflexion: Language agents with verbal reinforcement learning. *Advances in neural information processing systems*, 36:8634–8652, 2023.

Aarohi Srivastava, Abhinav Rastogi, Abhishek Rao, Abu Awal Md Shoeb, Abubakar Abid, Adam Fisch, Adam R Brown, Adam Santoro, Aditya Gupta, Adrià Garriga-Alonso, et al. Beyond the imitation game: Quantifying and extrapolating the capabilities of language models. *arXiv preprint arXiv:2206.04615*, 2022.

Oyvind Tafjord, Bhavana Dalvi, and Peter Clark. Proofwriter: Generating implications, proofs, and abductive statements over natural language. In *Findings of the association for computational linguistics: ACL-IJCNLP 2021*, pp. 3621–3634, 2021.

Qwen Team. Qwen2.5 technical report. *arXiv preprint arXiv:2412.15115*, 2024.

Trieu H Trinh, Yuhuai Wu, Quoc V Le, He He, and Thang Luong. Solving olympiad geometry without human demonstrations. *Nature*, 625(7995):476–482, 2024.

Xuezhi Wang, Jason Wei, Dale Schuurmans, Quoc Le, Ed Chi, Sharan Narang, Aakanksha Chowdhery, and Denny Zhou. Self-consistency improves chain of thought reasoning in language models. *arXiv preprint arXiv:2203.11171*, 2022.

Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Fei Xia, Ed Chi, Quoc V Le, Denny Zhou, et al. Chain-of-thought prompting elicits reasoning in large language models. *Advances in neural information processing systems*, 35:24824–24837, 2022.

Jundong Xu, Hao Fei, Liangming Pan, Qian Liu, Mong-Li Lee, and Wynne Hsu. Faithful logical reasoning via symbolic chain-of-thought. In *Proceedings of the 62nd annual meeting of the association for computational linguistics (volume 1: Long papers)*, pp. 13326–13365, 2024.

Boyang Yang, Zijian Cai, Fengling Liu, Bach Le, Lingming Zhang, Tegawendé F Bissyandé, Yang Liu, and Haoye Tian. A survey of llm-based automated program repair: Taxonomies, design paradigms, and applications. *arXiv preprint arXiv:2506.23749*, 2025a.

Kaiyu Yang, Gabriel Poesia, Jingxuan He, Wenda Li, Kristin Lauter, Swarat Chaudhuri, and Dawn Song. Formal reasoning meets llms: Toward ai for mathematics and verification. *Communications of the ACM*, 69(3):66–73, 2026.

Xiao-Wen Yang, Jie-Jing Shao, Lan-Zhe Guo, Bo-Wen Zhang, Zhi Zhou, Lin-Han Jia, Wang-Zhou Dai, and Yu-Feng Li. Neuro-symbolic artificial intelligence: Towards improving the reasoning abilities of large language models. *arXiv preprint arXiv:2508.13678*, 2025b.

Yu'an Yang, Siheng Xiong, Ali Payani, Ehsan Shareghi, and Faramarz Fekri. Harnessing the power of large language models for natural language to first-order logic translation. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 6942–6959, 2024.

Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. React: Synergizing reasoning and acting in language models. *arXiv preprint arXiv:2210.03629*, 2022.

# A METHOD DETAILS

This supplementary material provides additional details about SACA's schema-specific alignment rules, diagnostic outputs, and operational boundary, complementing Section 3.

**Separation of concerns.** SACA strictly terminates once an aligned artifact $z'$ is produced. Final inference is delegated back to the host formalize-then-solve pipeline, which renders or executes $z'$ using its corresponding symbolic backend. SACA is therefore responsible only for pre-execution schema alignment, while the solver remains responsible for deriving the task answer.

**Algorithm 1** Schema-Adaptive Constraint Alignment

**Input:** Generated symbolic artifact $z$
**Output:** Aligned artifact $z'$

$\tau \leftarrow \textsc{DetectSchema}(z)$
**if** $\tau$ is unsupported or ambiguous **then**
  **return** $z$ ▷ Conservative fallback
**end if**
$(\mathcal{D}, \mathcal{B}, q) \leftarrow \mathrm{Parse}_\tau(z)$
$\mathcal{C}_\tau(z) \leftarrow \mathrm{Cl}_\tau(\mathcal{D}, \mathcal{B} \cup \{q\})$
$\mathcal{V}_\tau(z) \leftarrow \emptyset$
**for** $s \in \mathcal{B} \cup \{q\}$ **do**
  **if** $\mathrm{Comp}_\tau(s, \mathcal{C}_\tau(z)) = 0$ **then**
    $\mathcal{V}_\tau(z) \leftarrow \mathcal{V}_\tau(z) \cup \{s\}$
  **end if**
**end for**
$\mathcal{V}_\tau^{\mathrm{rec}}(z) \leftarrow \emptyset$
**for** $v \in \mathcal{V}_\tau(z)$ **do**
  $\mathcal{R}_v \leftarrow \mathcal{R}_\tau(v; z, \mathcal{C}_\tau(z))$
  **if** $|\mathcal{R}_v| = 1$ **then**
    $\mathcal{V}_\tau^{\mathrm{rec}}(z) \leftarrow \mathcal{V}_\tau^{\mathrm{rec}}(z) \cup \{v\}$
    let $r_v$ be the unique element of $\mathcal{R}_v$
  **end if**
**end for**
$\mathcal{E}_\tau(z) \leftarrow \textsc{ConflictFree}(\{r_v\}_{v \in \mathcal{V}_\tau^{\mathrm{rec}}(z)})$
$z' \leftarrow \mathcal{A}_\tau(z, \mathcal{E}_\tau(z))$
**return** $z'$

For correspondence with Figure 2, $(\tau, \mathcal{D}, \mathcal{B}, q)$ denotes SACA's canonical structured view of $z$. The schema $\tau$ is inferred by DETECTSCHEMA; it is not assumed to be explicitly generated by the formalizer. The schema-conditioned parser $\mathrm{Parse}_\tau$ then extracts the declarations $\mathcal{D}$, symbolic statements $\mathcal{B}$, and query $q$ from the artifact.

**Auditable diagnostics.** Every SACA invocation produces a deterministic diagnostic trace. The trace records the detected schema, activated aligner, extracted constraint closure, detected violations, repair-candidate counts, retained repairs, modified artifact fields, repair types, and repair counts. For rule-style artifacts, diagnostics expose predicate-declaration completion and predicate-signature alignment. For CSP artifacts, diagnostics expose extracted domain–rank maps, equality normalization, comparison alignment, and query alignment. These diagnostics are used only for analysis and reporting; they are never used as feedback to regenerate artifacts.

**Recoverability and repair conflicts.** For each detected violation $v \in \mathcal{V}_\tau(z)$,

$$\mathcal{R}_\tau\big(v; z, \mathcal{C}_\tau(z)\big)$$

denotes the set of schema-permitted edits that resolve $v$ using only commitments encoded by the artifact-internal closure. A violation is recoverable iff

$$\big|\mathcal{R}_\tau\big(v; z, \mathcal{C}_\tau(z)\big)\big| = 1.$$

For each recoverable violation, $r_v$ denotes this unique repair.

Two repairs conflict, written $r_u \bowtie r_v$, when they prescribe incompatible writes to the same declaration slot, statement field, or query attribute. The retained repair set is

$$\mathcal{E}_\tau(z) = \{r_v \mid v \in \mathcal{V}_\tau^{\mathrm{rec}}(z),\ \nexists\, u \in \mathcal{V}_\tau^{\mathrm{rec}}(z) \setminus \{v\} \text{ such that } r_u \bowtie r_v\}\,.$$

Conflicting repairs are withheld rather than resolved using an external preference signal. The retained edits are applied in a fixed schema-specific order; for edits on disjoint artifact locations, this order is immaterial.

**Rule-schema details.** For rule-style artifacts, the closure $\mathcal{C}_\tau(z)$ contains a predicate symbol table. Let $\mathcal{P}_{\mathrm{decl}}$ denote predicates declared in $\mathcal{D}$ and $\mathcal{P}_{\mathrm{use}}$ denote predicates observed in $\mathcal{B} \cup \{q\}$. Missing declarations are identified by

$$\mathcal{P}_{\mathrm{miss}} = \mathcal{P}_{\mathrm{use}} \setminus \mathcal{P}_{\mathrm{decl}}.$$

For each $p \in \mathcal{P}_{\mathrm{miss}}$, SACA adds a declaration only when the observed usage determines one unique schema-valid form. For property-style predicates, this canonical form is

```
p($x, bool).
```

For predicate-signature alignment, let $\mathcal{H}(p)$ denote the multiset of observed arities for predicate $p$ across declarations and calls. A candidate target arity is

$$a^\star(p) = \arg\max_a \mathrm{count}_{\mathcal{H}(p)}(a),$$

subject to schema-specific validity requirements such as the presence of a truth-value slot. A repair is applied only when this choice and the corresponding edit are unique; ties or otherwise ambiguous signatures are left unchanged. Examples include removing a duplicated truth-value argument or restoring a uniquely determined Boolean slot.

**CSP-schema details.** For CSP artifacts, let $\mathcal{O}$ denote ordinal descriptions and $\mathcal{R}$ numeric ranks exposed in the generated artifact's `Domain` fields. SACA extracts the artifact-local map

$$m_z : \mathcal{O} \to \mathcal{R}.$$

The map is local to each generated artifact, so ordinal phrases are interpreted according to that artifact's own coordinate system.

The ordinal evidence used by SACA is explicitly present inside $z$ as generated tags, attributes, or inline comments. For example, a generated constraint may expose

```
newest:  bus == 5,
```

where the ordinal tag `newest` is part of the generated artifact rather than recovered from the original natural-language problem.

For an equality-style constraint with executable rank $r$ and exposed ordinal attribute $o$,

$$v = r \qquad \text{with ordinal attribute } o,$$

schema compliance requires

$$r = m_z(o).$$

If $r \neq m_z(o)$ and the local map determines a unique replacement, SACA rewrites the executable rank as

$$v = m_z(o).$$

Thus, when $m_z(\textit{newest}) = 1$, the artifact-level pair `newest: bus == 5` is normalized to `bus == 1`.

Comparison constraints such as $v_i < v_j$ and $v_i > v_j$ are checked against the ordering direction exposed by the local domain map and the associated generated ordinal annotation. When the operator is uniquely inconsistent with that representation, SACA aligns it accordingly. The same policy applies to query constraints so that premises and candidate answer conditions use a common artifact-local coordinate system.

**Conservative fallback.** Across schemas, SACA modifies an artifact component only when exactly one schema-permitted repair is supported by $\mathcal{C}_\tau(z)$. If no repair is supported, multiple repairs remain possible, or retained repairs conflict, the affected component is left unchanged. Unsupported or ambiguous schemas return the original artifact unchanged. This fail-safe behavior prevents SACA from introducing semantic assumptions not licensed by the generated artifact itself.

**Operational scope and extensibility.** SACA is designed for artifact-producing formalize-then-solve systems whose outputs expose detectable schemas and recoverable internal constraints. This includes rule sketches, rendered rule programs, CSP programs, and other solver-compatible artifacts with explicit structural commitments. Within this boundary, SACA is formalizer-agnostic because it operates on the generated artifact rather than on the internal parameters of the generator.

Extending SACA to a new symbolic schema requires registering its surface signature with the schema detector and defining the schema-specific components

$$\big(\mathrm{Parse}_\tau, \mathrm{Cl}_\tau, \mathrm{Comp}_\tau, \mathcal{R}_\tau, \mathcal{A}_\tau\big),$$

where $\mathrm{Parse}_\tau$ extracts the structured artifact view, $\mathrm{Cl}_\tau$ constructs the constraint closure, $\mathrm{Comp}_\tau$ checks schema compliance, $\mathcal{R}_\tau$ enumerates artifact-supported candidate repairs, and $\mathcal{A}_\tau$ applies the retained conflict-free repairs. This modular interface can be instantiated for additional solver-compatible languages once their internal structural commitments are specified.

**Zero-leakage boundary.** All repairs are determined by declarations, statements, query fields, usage patterns, and local semantic conventions exposed inside the generated artifact. SACA introduces no additional LLM call, uses no gold labels or predicted-answer correctness, and does not condition on solver success, solver errors, execution traces, or solver-derived answer feedback. Its role is therefore not post-hoc answer correction, but deterministic enforcement of artifact-internal schema consistency before symbolic execution.

## B Dataset Details

This supplementary material provides additional details about the dataset settings used in our experiments. The three datasets are selected to cover complementary symbolic-formalization regimes: rule-sketch rendering, CSP domain grounding, and rule-program consistency. They differ in reasoning structure, output schema, answer format, and dominant schema-level stress test, allowing us to examine whether SACA can align generated artifacts with their own declared internal constraints.

**Diagnostic subsets.** We use fixed diagnostic subsets rather than dynamically sampled evaluation instances: 500 examples for ProntoQA and 100 examples each for LogicalDeduction and ProofWriter. These subsets are used consistently across the compared variants in the diagnostic experiments. Their purpose is not only to report aggregate accuracy, but also to support artifact-level tracing: for each example, we store generated symbolic artifacts, solver outputs, SACA diagnostics, repair counts, and before–after program changes. This protocol enables controlled analysis of which internal schema constraint is violated and whether the corresponding repair affects executability or prediction correctness.

**ProntoQA.** ProntoQA consists of rule-chaining questions with compact contexts and binary answer options. In our setting, the model first produces a rule sketch, which must then be rendered into an executable symbolic program. The dominant stress test is sketch renderability: if the generated sketch uses predicates that are not properly declared, the renderer may produce an empty or incomplete program. Therefore, ProntoQA mainly tests whether sketch-level SACA can recover missing predicate declarations and improve executable symbolic reasoning.

**LogicalDeduction.** LogicalDeduction contains ordering and ranking problems with five answer options. We represent these problems using a CSP-style symbolic schema, where entities are mapped to local numeric domains and constraints encode ordinal or comparative relations. The dominant stress test is domain–ordinal consistency: a generated program may declare a local domain such as $1 =$ *newest* and $5 =$ *oldest*, while later encoding equality, comparison, or query constraints under an inconsistent rank interpretation. Therefore, LogicalDeduction tests whether SACA can align

constraints with the program's own declared CSP domain before solver execution. In addition to the diagnostic subset, we evaluate LogicalDeduction on the complete 300-example development split to verify whether CSP-alignment gains persist beyond the diagnostic setting.

**ProofWriter.** ProofWriter consists of rule-inference examples with longer contexts and three-way answer labels. The generated symbolic programs are usually executable, but they may contain predicate-signature inconsistencies, such as inconsistent arities or incompatible truth-value argument positions. Unlike ProntoQA, the main challenge is not rendering failure, but maintaining internal symbol consistency within an already executable rule program. Therefore, ProofWriter tests whether SACA can act as a conservative safety layer that repairs symbol-level inconsistencies without over-modifying otherwise valid programs.

**Schema and SACA level.** The main paper reports compact dataset statistics and artifact schemas in its dataset-statistics table. ProntoQA uses rule-sketch artifacts, LogicalDeduction uses CSP-style programs, and ProofWriter uses rule-program artifacts. Accordingly, the activated alignment targets are declaration completion, domain–ordinal normalization, and predicate-signature alignment, respectively. Across all datasets, SACA is oracle-free and solver-feedback-free: it only reads the generated artifact itself and does not use gold answers, predicted-answer correctness, or solver-derived target labels.

**Evaluation protocol.** For each diagnostic setting, all compared methods are evaluated on the same fixed example IDs. We report answer accuracy, executable rate, and executable accuracy in the main text. Answer accuracy measures exact match between the predicted answer and the gold answer. Executable rate measures the fraction of generated symbolic artifacts that can be successfully executed by the corresponding solver. Executable accuracy measures answer accuracy restricted to executable examples. In addition, we report repair statistics, activation counts, win–loss patterns, plug-in behavior, and qualitative before–after cases to analyze whether improvements arise from enhanced internal consistency of generated symbolic artifacts.

## C Baseline Adaptation, Decoding, and Provenance

This supplementary material describes the adaptation of prompt baselines and symbolic-execution baselines used in our experiments. Our goal is not to claim official leaderboard reproductions of every prompting method, but to establish strictly controlled local comparisons under shared backbones, fixed example IDs, documented decoding settings, and consistent answer extraction.

**Prompt-only baselines.** We evaluate Direct prompting, Chain-of-Thought (CoT), Self-Consistency CoT (SC-CoT), SymbCoT, and Symbolic-Aided CoT as prompt-only reasoning baselines. These methods directly output answer choices from natural-language or symbolic-looking reasoning traces. They do not produce solver-compatible symbolic artifacts and therefore cannot be passed to SACA without first being converted into an executable artifact. This distinction is important for our operational scope: SACA is designed for artifact-producing formalize-then-solve pipelines, not arbitrary free-form reasoning traces.

For all prompt-only baselines, we use the same dataset examples as the corresponding symbolic-execution methods. The 7B prompt baselines use Qwen2.5-7B-Instruct (Team, 2024), while the larger-model comparison uses Qwen2.5-14B-Instruct (Team, 2024). Direct, CoT, SymbCoT, and Symbolic-Aided CoT are evaluated as single-output prompt baselines using greedy decoding. SC-CoT uses fixed stochastic sampling with temperature $0.7$ and five samples, followed by majority voting over extracted answer choices. The fixed sampling configuration is used consistently across datasets.

**Symbolic-execution baselines.** Logic-LM is used as an external formalize-then-solve baseline. Given a natural-language problem, it produces a solver-compatible symbolic program and then invokes the corresponding symbolic solver. We evaluate both the original Logic-LM output and Logic-LM+SACA, where the same SACA module is inserted between Logic-LM generation and solver execution.

FARA-CF is our native formalizer used for the main system instantiation. The unaligned FARA-CF variant generates symbolic artifacts and sends them directly to the solver. FARA-CF+ applies SACA

| Setting | Files | Exact-size Src. | Oversized Src. | Unverif. |
|---|---|---|---|---|
| ProntoQA `local_500` | 20 | 20 | 0 | 0 |
| LogicalDed. `local_100` | 25 | 4 | 21 | 0 |
| ProofWriter `local_100` | 28 | 4 | 24 | 0 |
| LogicalDed. full | 12 | 12 | 0 | 0 |
| Total | 85 | 40 | 45 | 0 |

Table 4: Sample-consistency audit summary. "Oversized Src." indicates that the associated source file contains more rows than the evaluated subset; it does not indicate an example-ID mismatch. For LogicalDeduction and ProofWriter diagnostic settings, all reported 100-example results use the verified matched prefix-100 IDs.

to the generated artifact before solver execution. Thus, FARA-CF and FARA-CF+ differ only by the insertion of the deterministic pre-execution alignment layer.

**Decoding settings.** All single-output generations use greedy decoding. This includes prompt-only Direct, CoT, SymbCoT, Symbolic-Aided CoT, Logic-LM direct formalization, and FARA-CF artifact generation. Greedy decoding is used to isolate reasoning and formalization behavior from sampling variance. For LogicalDeduction full evaluation, Logic-LM uses a maximum generation length of 1024 tokens, while FARA-CF uses 2048 tokens to accommodate structured CSP artifacts. All symbolic outputs are evaluated with the same solver and answer-extraction scripts under the corresponding dataset setting.

**Provenance of 14B prefix-100 results.** For the model-scale comparison, Qwen2.5-14B-Instruct prompt-only results are reported on the exact same diagnostic example IDs as the 7B symbolic-execution methods. When completed 14B solver100 artifacts already existed on larger source files, we created prefix-100 aliases only after verifying that the first 100 ordered IDs exactly matched the intended 7B diagnostic IDs. These aliases are byte-identical copies of the verified completed artifacts and do not rerun generation, alter predictions, or change answer extraction. This ensures that the 14B prompt-only comparison differs in reasoning paradigm and model scale, not in sample selection.

**No official reproduction claim.** Prompt baselines are local adaptations implemented under our shared experimental environment. We therefore use them as controlled baselines rather than claiming official reproductions of the original papers. All comparisons are interpreted under this controlled setting: fixed backbones, fixed example IDs, documented decoding settings, consistent answer extraction, and auditable artifact provenance.

## D Sample Consistency Audit

This supplementary material documents the sample-consistency audit used to guarantee fair comparisons across prompt baselines, symbolic-execution baselines, SACA variants, and diagnostic artifacts.

**Audit scope.** We audited prompt outputs, generated symbolic programs, solver inference outputs, and per-example SACA diagnostics used in the reported results. The audit checks whether each artifact contains the intended number of examples, whether example IDs are unique, whether the ID order matches the reference source, and whether compared methods are evaluated on the same example IDs within each setting.

**Audit summary.** The audit covers 85 headline or result-used artifacts across ProntoQA, LogicalDeduction, and ProofWriter. ProntoQA `local_500` and LogicalDeduction full-dev results are fully aligned across the audited main methods and diagnostics. For LogicalDeduction and ProofWriter diagnostic settings, all intended 100-example headline outputs match the same prefix-100 IDs. Thus, all diagnostic comparisons reported in the main text are conducted on matched examples.

**Oversized source files.** Some diagnostic outputs are associated with source files containing more rows than the evaluated subset. Specifically, the LogicalDeduction `local_100` source contains 300 rows and matches the ID order of the full LogicalDeduction development split. Similarly, the ProofWriter `local_100` source contains 600 rows and matches the ID order of the ProofWriter development split. All audited 100-row LogicalDeduction and ProofWriter headline outputs nevertheless match the intended first 100 example IDs. Thus, the 'Oversized Src.' counts in Table 4 indicate source-file size differences, not inconsistent evaluation samples.

**Prefix-100 aliases.** For the model-scale comparison, completed 14B artifacts on LogicalDeduction and ProofWriter were available for larger source files. We created verified prefix-100 aliases only when the first 100 ordered example IDs exactly matched the 7B diagnostic IDs. The resulting prefix-100 artifacts are byte-identical aliases of the completed 14B outputs restricted to the matched prefix. They are used only for diagnostic-100 comparisons and are not mixed with full-split evaluations.

**Fairness guarantee.** All reported diagnostic comparisons use identical example IDs within each dataset setting. ProntoQA `local_500` uses 500 matched examples. LogicalDeduction diagnostic and ProofWriter diagnostic use the verified prefix-100 IDs. LogicalDeduction full uses the complete 300-example development split. This audit ensures that observed differences between methods arise from prompting, formalization, symbolic execution, or SACA alignment, rather than from sample mismatch.

## E No-Refinement and No-Leakage Provenance

This supplementary material clarifies the provenance of SACA inputs and verifies that all reported SACA experiments operate under a strict zero-leakage regime. SACA does not use gold labels, predicted correctness, solver feedback, inference traces, or additional LLM calls.

**SACA input boundary.** SACA is applied to generated symbolic artifacts before solver execution. For each example, the input to SACA is the generated artifact itself: a rule sketch, CSP program, or rule program depending on the detected schema. SACA reads structural declarations, generated statements, query conditions, and observable usage patterns inside the artifact. It does not read the gold answer, the predicted answer, solver correctness, or execution traces.

**Logic-LM no-refinement outputs.** For Logic-LM experiments, the reported artifacts are no-refinement outputs. Each example uses one generation pass followed by deterministic cleanup. No solver-feedback retry, post-execution self-correction, answer-based selection, or LLM regeneration is used before SACA. The same unchanged SACA module is then applied to the generated program-only artifact.

**FARA-CF and FARA-CF+ provenance.** For FARA-CF+, SACA is applied to artifacts produced by the FARA-CF formalizer. The unaligned FARA-CF baseline and the aligned FARA-CF+ variant use the same generated artifacts before alignment. The only difference is whether the deterministic SACA layer is inserted before solver execution. This makes the comparison between FARA-CF and FARA-CF+ a controlled test of pre-execution schema alignment.

**Diagnostic trace provenance.** SACA emits the deterministic diagnostic traces described in Appendix A. In all reported experiments, these traces are retained only for auditing and analysis; they are never used to regenerate, revise, select, or score artifacts, nor are they fed back into the formalizer or SACA.

**Strict zero-leakage guarantee.** SACA is oracle-free, feedback-free, and task-inference-free. It uses no gold answers, no predicted-answer correctness, no solver-derived labels, no inference traces, and no additional LLM calls. All repairs are determined by the generated artifact's own declarations, usage patterns, and local semantic conventions. When a violation is not recoverable from the artifact's own constraint closure, SACA follows a conservative fail-safe protocol: it leaves the corresponding artifact component unchanged rather than introducing external semantic assumptions. This behavior ensures that SACA improves internal schema consistency without becoming an answer-driven correction mechanism.

| Dataset | Schema | SACA Level | Repaired Ex. | Total Repairs | Main Repaired Inconsistency |
|---|---|---|---|---|---|
| ProntoQA | Rule Sketch | Sketch-Level | 35/500 | 37 | Undeclared Predicates |
| LogicalDeduction | CSP | Program-Level | 33/100 | 139 | Domain–Ordinal Mismatch |
| ProofWriter | Rule Program | Program-Level | 1/100 | 10 | Predicate Arity Mismatch |

Table 5: SACA activation statistics on diagnostic subsets.

| Data | Baseline | W | L | C–C | W–W |
|---|---|---|---|---|---|
| PQA | Direct | 43 | 7 | 399 | 51 |
| PQA | w/o SACA | 16 | 3 | 426 | 55 |
| LD | Direct | 23 | 6 | 68 | 3 |
| LD | w/o SACA | 21 | 0 | 70 | 9 |
| PW | Direct | 0 | 0 | 83 | 17 |
| PW | w/o SACA | 1 | 0 | 82 | 17 |

Table 6: Win–loss analysis of SACA in FARA-CF+. Each row compares SACA against the listed baseline. PQA, LD, and PW denote ProntoQA, LogicalDeduction, and ProofWriter. C–C and W–W denote both-correct and both-wrong cases.

# F Additional Experimental Results

This supplementary material provides detailed mechanism, ablation, runtime, and case-level analyses that support the main experimental findings.

## F.1 SACA Activation Statistics

Table 5 summarizes the repair activation statistics across the three diagnostic settings. SACA repairs different schema-level inconsistencies depending on the detected artifact schema and the structural level at which the violation appears.

The sparse activation on ProofWriter illustrates SACA's conservative behavior rather than indiscriminate rewriting. SACA commits repairs on only 1 of 100 diagnostic examples; relative to FARA-CF without SACA, this intervention produces 1 win and 0 losses (Table 6). The low activation rate is therefore consistent with SACA's design goal of intervening only when the generated artifact itself uniquely determines a repair.

## F.2 Win–Loss Analysis

Table 6 shows that SACA's repairs are prediction-relevant rather than merely cosmetic. Against FARA-CF without SACA, SACA produces 16 wins and only 3 losses on ProntoQA, 21 wins and 0 losses on LogicalDeduction, and 1 win and 0 losses on ProofWriter.

## F.3 Schema-Aligner Ablation

Table 7 reports the schema-aligner ablation. Disabling rule alignment affects rule-style settings, while disabling CSP alignment removes the gains on LogicalDeduction.

## F.4 Runtime Details

Table 8 reports the full runtime measurements. We measure in-memory SACA alignment time only, excluding LLM generation, solver inference, evaluation, and artifact I/O. SACA introduces zero additional LLM calls and uses no gold labels or solver feedback. The runtime counter is distinct from the diagnostic repair count: Align. Ops. records runtime-instrumented alignment operations, whereas Total Repairs in Table 5 counts committed artifact mutations.

| Setting | Full SACA | w/o Rule | w/o CSP |
|---|---|---|---|
| ProntoQA | **88.4** | 85.2 | **88.4** |
| LogicalDed. | **91.0** | **91.0** | 70.0 |
| ProofWriter | **83.0** | 82.0 | **83.0** |
| LogicalDed.-Full | **68.3** | **68.3** | 58.3 |

Table 7: Schema-aligner ablation. Accuracy is reported in percentages.

| Setting | $N$ | ms/ex. | Align. Ops. |
|---|---|---|---|
| ProntoQA diag. | 500 | 0.217 | 1514 |
| LogicalDeduction diag. | 100 | 0.447 | 91 |
| ProofWriter diag. | 100 | 0.182 | 2 |
| LogicalDeduction full | 300 | 0.477 | 410 |

Table 8: Runtime overhead of SACA. We measure in-memory alignment time only. Align. Ops. counts runtime-instrumented alignment operations and is distinct from the number of committed artifact mutations reported as Total Repairs in Table 5.

### F.5 Additional Case Notes

The main text includes the case-study figure because it provides a direct visualization of how SACA modifies symbolic artifacts. Here we summarize the three repairs in words. For ProntoQA, SACA recovers a missing predicate declaration required for rendering a rule sketch into an executable program. For LogicalDeduction, SACA extracts the declared domain–ordinal map and rewrites a self-contradictory constraint from `bus == 5` to `bus == 1`. For ProofWriter, SACA detects incompatible predicate signatures and aligns the predicate arity according to the artifact's observed symbol table. Across all three cases, the repair is justified by the artifact's own schema commitments rather than gold answers or solver feedback.