AI Engineering
https://www.sciltp.com/journals/aieng

Article

NeSyWikiCompiler: A Neural-Symbolic Compiler for Verifiable AI Knowledge Engineering

Bailing Zhang

School of Computer Science and Data Engineering, NingboTech University, Ningbo 315100, China;
bailing.zhang1961@gmail.com

How To Cite: Zhang, B. NeSyWikiCompiler: A Neural-Symbolic Compiler for Verifiable AI Knowledge Engineering. *AI Engineering* **2026**, *2*(2), 11. https://doi.org/10.53941/aieng.2026.100011

Received: 16 May 2026
Revised: 30 June 2026
Accepted: 14 July 2026
Published: 27 July 2026

Abstract: Knowledge rules in AI engineering carry an awkward double duty. A clinician or a compliance officer has to be able to read and revise them, and a downstream system has to be able to execute and check them. Getting from the natural-language version of such a rule to a symbolic program is rarely the hard part; getting to one that is not merely runnable but logically sound is. We call this the compilation gap and attack it with NeSyWikiCompiler, a four-stage neural-symbolic pipeline. A language-model frontend reads each specification into NeSy-IR, an intermediate representation that holds onto exactly the details downstream code generation needs and that language models routinely drop: predicate types, the direction of numeric comparisons, the modality of each constraint, and a pointer back to the source text. From a single IR, deterministic compilers emit both a Prolog program and a Z3 program. The Z3 side is then checked with Clark's completion, which surfaces a failure that, in our experiments, plain Prolog execution cannot see at all: rules whose violation condition can never be satisfied, so that the checker silently never fires. A repair loop that is CEGIS-informed but ultimately deterministic for structural errors handles the two kinds of failure differently: an LLM revision step targets semantic slips such as lost numeric thresholds and reversed arguments, while a deterministic reconstruction step—reached once the LLM rounds have failed—is the path taken for the structural contradictions, which the LLM step does not fix. In a cross-domain evaluation covering clinical decision rules, AI course knowledge, and legal compliance specifications—three representative domains rather than an exhaustive sample—all compiled programs achieve syntax validity across both backends. Formal verification exposes a class of structural contradictions that, in this pilot-scale benchmark, is concentrated in the legal specifications, where normative hedging constructs appear to induce rule-constraint conflicts. Deterministic repair recovers most structural failures while LLM-only repair consistently reproduces the same broken rule patterns. These findings characterise a previously unrecognised failure mode in LLM-to-logic compilation and demonstrate a practical engineering toolchain for producing verifiable knowledge-base programs from natural-language specifications. We present the domain-level rates as failure-mode discovery on a small corpus rather than as estimates that generalise without further study.

Keywords: neural-symbolic AI; knowledge compilation; formal verification; logic programming; large language models; AI engineering

1. Introduction

Several branches of AI engineering—clinical decision support, legal compliance checking, educational knowledge management, industrial constraint enforcement—have converged on the same uncomfortable requirement. The rules that drive these systems have to exist in two forms at once. Domain experts need to author and revise them in something close to natural language, and the systems that consume them need a formal version that can be executed and reasoned over without surprises. Moving between the two forms, and doing so in a way that leaves an auditable trail, is an old problem; neither hand formalisation nor end-to-end neural generation has solved it at the scale these systems now demand.

Where correctness really matters, the rules are still written out by hand. Formats such as the Arden Syntax for

clinical logic [1] and the GuideLine Interchange Format for clinical protocols [2] give that hand-work a structured target, but they presuppose a trained specialist, and specialists are scarce and slow relative to how fast the underlying knowledge changes. Fully neural generation goes the other way: it scales readily, yet it returns no correctness guarantee [3,4], and the code it produces can parse and load cleanly while being wrong in ways that surface only once the system is in use.

It helps to split the compilation gap into three problems that fail independently. Extraction is reading predicates, rules, and constraints out of prose. Representation is choosing an intermediate form that does not quietly discard modality, the direction of a numeric comparison, or the link back to the source text. Verification is catching the programs that compile but are logically wrong. Recent large language models (LLMs) have made the first problem much easier, and in doing so have made the third more pressing: they introduce extraction errors of a specific, repeatable kind that standard Prolog execution never flags and that only formal methods catch. The LLM Wiki view of language models—articulated in an informal note by Karpathy [5] rather than in a peer-reviewed venue—treats them as curators of a continuously maintained knowledge base rather than one-shot answer machines, and raises the stakes once more, because a knowledge base is only worth maintaining if its compiled rules can be shown to be correct.

We present NeSyWikiCompiler, a four-stage pipeline that addresses the compilation gap through the following design decisions. First, a modality-aware intermediate representation (NeSy-IR) captures the logical structure of each specification in a form that is backend-agnostic, allowing the same extracted knowledge to target both Prolog and Z3. Second, a deterministic numeric recovery step corrects a systematic LLM extraction failure that converts inequality thresholds into equality predicates. Third, a Z3 verifier using Clark's completion [6] detects structural contradictions—rule-constraint combinations whose violation conditions can never be satisfied—a failure class that Prolog execution misdiagnoses as "no current violations." Fourth, a CEGIS-informed repair loop pairs an LLM-guided revision step for semantic errors with a deterministic reconstruction step that, as our results show, is the component that actually resolves structural errors.

The contributions of this paper are as follows. We introduce NeSy-IR, a modality-aware intermediate representation with provenance tracking for multi-backend knowledge compilation. We implement a deterministic compiler from NeSy-IR to Prolog and Z3 that achieves syntax validity across all evaluated specifications. We demonstrate that, for the Horn-clause fragment under closed-world semantics defined in Section 3, Clark's completion is necessary and sufficient for detecting structural contradictions that open-world verification cannot identify. We characterise the limits of LLM-guided CEGIS repair on a small cross-domain benchmark and show that deterministic reconstruction recovers the majority of structural failures. We release source code, benchmark specifications, and evaluation scripts to support reproducibility and future work in LLM-to-logic compilation.

2. Related Work

2.1. LLM-to-Logic and Semantic Parsing

A growing body of work applies LLMs to the extraction of formal representations from natural language. Semantic parsing approaches [7,8] map natural-language sentences to logical forms; more recent work applies pre-trained LLMs to text-to-SQL [9] and text-to-code generation at large scale [4]. Autoformalization [10] targets the conversion of mathematical statements to proof assistants. NeSyWikiCompiler differs from these efforts in that its primary target is not syntactic executability but the formal consistency of the compiled rule base: a program can be syntactically valid Prolog while its violation conditions are structurally unreachable, a failure class that evaluation on test inputs does not expose.

2.2. Program Synthesis and CEGIS

Counterexample-guided inductive synthesis (CEGIS) [11] uses verification failures to guide program search. Gulwani [12] extended this to semantic repair from input-output examples; subsequent work has applied CEGIS to domain-specific language synthesis [13]. NeSyWikiCompiler adapts CEGIS to repair a partially structured IR produced from natural language. A key distinction from classical CEGIS is that many failures in our setting originate in semantic extraction by the LLM rather than in program enumeration over a formal grammar: the synthesis space is implicitly defined by the LLM's prior over IR structures, not by an explicit formal language.

2.3. Neuro-Symbolic Systems

DeepProbLog [14] and NeurASP [15] integrate neural networks with probabilistic logic programming and answer-set programming, respectively, enabling end-to-end differentiable inference over symbolic structures. LNN [16] embeds logic into neural networks through continuous relaxations. These systems target joint neural-symbolic learning, whereas NeSyWikiCompiler targets static compilation of natural-language specifications to verified executables.

The systems address orthogonal problems and could complement each other in a pipeline where compiled rules serve as the symbolic component of a neuro-symbolic reasoner.

2.4. Clinical and Legal Knowledge Formalisation

Clinical decision rules have been formally represented using Arden Syntax [1], GLIF [2], and CDSS rule engines [17]. These approaches require manual expert authoring. Legal compliance formalisation [18] faces similar challenges, compounded by the complexity of normative language: legal rules routinely employ nested exceptions, conditional permissions, and modality stacking that introduce structural ambiguity. In our evaluation, legal specifications exhibit the highest rate of structural contradictions, consistent with prior observations about the difficulty of formalising hedged normative language [19].

2.5. Formal Verification and Clark's Completion

SMT solvers [20] are well established for software verification and symbolic model checking. Clark's completion [6] transforms logic programs into equivalences under the closed-world assumption, enabling satisfiability proofs that standard Prolog execution—which uses negation as failure under an open-world assumption—cannot produce. Standard implication-based rule encoding ($B \Rightarrow H$) allows the rule head to hold independently of any body, making it impossible to detect cases where the rule body is the only way the head can be true. To our knowledge, this paper is the first to apply Clark's completion specifically to detect structural contradictions in LLM-compiled knowledge rules, a contribution that bridges the logic-programming and SMT communities in the context of AI knowledge engineering.

2.6. AI Knowledge Engineering Systems

The LLM Wiki framing, proposed in an informal note [5], positions language models as maintainers of continuously updated, structured knowledge bases. Companion systems in the same research agenda address temporal staleness (C3 criterion) and knowledge coverage (C4 criterion), as set out in a preprint [21]; we cite both sources as the origin of the terminology we adopt rather than as peer-reviewed evidence. The present paper addresses correctness (C1) and internal consistency (C2) through formal compilation and verification, closing the gap between natural-language specification and auditable executable artifact.

3. Materials and Methods

3.1. System Overview and Problem Formulation

Let s be a natural-language knowledge specification. Rather than fold everything into a single "compile-and-check" step, we separate three guarantees that the pipeline provides, because they hold over different backends and rest on different assumptions.

Guarantee 1 (syntactic synthesis)**.** From the intermediate representation the pipeline emits a program P_s^B for each target backend $B \in \{Prolog, Z3\}$ that parses and loads without error. This is a property of both backends and is the weakest of the three: a program can be syntactically valid yet logically wrong.

Guarantee 2 (Prolog execution)**.** The Prolog artifact is executable under SWI-Prolog with negation as failure, so that injected facts can be queried against the generated `check` and `violation` predicates (the VioTest of Section 4.3). This is an empirical, open-world check: it can exhibit a triggering counterexample but cannot certify the absence of a structural defect.

Guarantee 3 (Clark-completed SMT verification)**.** The Z3 artifact, and only the Z3 artifact, is subjected to formal verification under Clark's completion. For each `must_not` constraint c_i with condition ϕ_i and forbidden action α_i, we ask whether $\phi_i \wedge \alpha_i$ is satisfiable under the Clark-completed rule set—that is, whether the violation-detection predicate is non-vacuous: whether any state assignment exists in which the violation could be detected. This is the closed-world guarantee that the other two backends do not provide.

We stress that Guarantee 3 is Z3-specific. The non-vacuity test and the Clark-completion encoding are not performed on the Prolog side; Prolog supplies executable rules and empirical VioTest evidence, not the same formal semantics. Conflating the two would overstate what Prolog execution certifies, which is precisely the failure mode this paper warns against.

If no program satisfying Guarantee 3 can be produced immediately, Stage 4 attempts repair.

Condition for Guarantee 3 requires clarification. For a `must_not` constraint, satisfiability of $\phi_i \wedge \alpha_i$ does not

mean violations are permitted; it means the violation checker is functional. An UNSAT result indicates the opposite: the rule body structurally prevents the violation from being detectable under any fact assignment, which is an error in the compiled program, not a safety guarantee.

Figure 1 shows the four-stage pipeline.

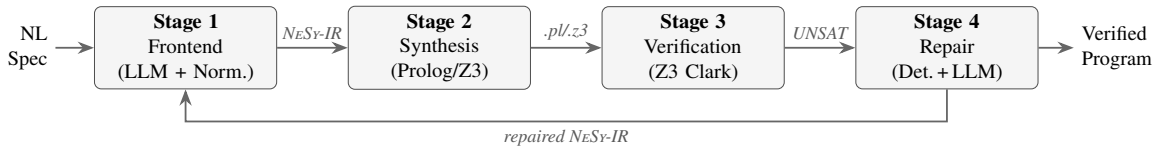

Figure 1. The NeSyWikiCompiler four-stage pipeline. When Stage 3 returns UNSAT (structural contradiction detected), Stage 4 attempts repair and returns a revised NeSy-IR to Stage 1 for recompilation.

3.2. NeSy-IR: Intermediate Representation

Definition 1 (NeSy-IR)**.** *A NeSy-IR document $\mathcal{R}$ is a tuple $\langle \mathcal{T}, \Pi, \mathcal{F}, \mathcal{R}_r, \mathcal{C} \rangle$ where $\mathcal{T}$ is a typed vocabulary; Π is a set of predicate declarations with arity and sort signatures; $\mathcal{F}$ is a set of ground facts; $\mathcal{R}_r$ is a set of Horn clauses $H \leftarrow B_1 \wedge \cdots \wedge B_n$; and $\mathcal{C}$ is a set of constraints, each annotated with modality $m \in \{\texttt{must}, \texttt{must\_not}, \texttt{should}\}$, severity $\sigma \in \{\texttt{hard}, \texttt{soft}\}$, a condition formula ϕ, a conclusion formula ψ, and a provenance field* `src`.

Three design elements distinguish NeSy-IR from a plain predicate-logic encoding. Modality drives backend code generation: `must_not` generates violation checkers; `must` generates obligation monitors. Numeric comparison operators (`lt`, `le`, `gt`, `ge`) are first-class schema citizens, distinct from equality-valued predicate arguments; this distinction is central to the numeric recovery procedure (Section 3.3). Provenance retains the original NL text fragment in each predicate and constraint node, enabling diagnostic messages that reference the specification rather than abstract formula components.

3.2.1. From one IR to Two Consistent Backends

A natural question is how a single NeSy-IR document yields a Prolog program and a Z3 program that agree with each other and with the input specification. The answer is that neither backend re-reads the natural language: both read only $\mathcal{R}$, and $\mathcal{R}$ is the single place where predicate names, arities, argument sorts, and the condition and conclusion formulas of every constraint are fixed. Consider a `must_not` constraint c_i with condition ϕ_i and forbidden action α_i as recorded in $\mathcal{C}$. The Prolog backend turns the very same $\phi_i \wedge \alpha_i$ into an executable `violation` clause evaluated under negation as failure (Section 3), whereas the Z3 backend turns that identical $\phi_i \wedge \alpha_i$ into a satisfiability query over the Clark-completed rule set (Equation (1)). Because both encoders draw their symbols from the shared declarations in Π, the two programs cannot disagree about which predicates exist or how many arguments each one takes; they differ only in reasoning semantics—open-world execution on the Prolog side, closed-world satisfiability on the Z3 side. That difference is deliberate rather than accidental: Prolog is the artifact we run, Z3 is the artifact we verify, and routing both through one IR is what keeps the runnable program and the verified program aligned with the original rule. We make the boundaries of this guarantee explicit at the end of Section 3.1.

3.2.2. Coverage Boundary of NeSy-IR

The schema in the definition above does not capture every conceivable rule, and it is useful to state where its edges are rather than let readers infer them from the examples. We distinguish three classes. Guaranteed to compile are rules whose logical content fits the Horn-clause fragment: a conjunction of positive and stratified-negated body literals, predicates of fixed arity over the sorts declared in Π, numeric thresholds expressed with the four comparison operators, and constraints whose modality is one of $\{\texttt{must}, \texttt{must\_not}, \texttt{should}\}$. Every specification in our benchmark falls in this class, which is why all 39 reach syntax validity on both backends. Best-effort are rules that the schema can represent but whose extraction is error-prone: single-level exceptions encoded as body negations, and disjunctive conditions that the frontend may split across clauses. These compile, but they are the cases where E2 structural contradictions and numeric-recovery misfires arise, so a clean compile does not imply a faithful one. Currently unsupported are rules requiring alternating or nested first-order quantifiers (for example, "for every record there exists a controller such that. . . "), arithmetic spanning multiple predicates in one constraint, recursive definitions that are not stratified, and genuinely deontic operators (obligation and permission with temporal scope) beyond the three flat modalities. The three non-SAT specifications discussed in Section 4 fall in this last class. We treat the first class as a guarantee, the second as a known risk surface, and the third as out of scope for the present encoder.

3.3. Stage 1: LLM Frontend and Numeric Recovery

The frontend calls GLM-4-flash (ZhipuAI, model identifier `glm-4-flash`) with a structured system prompt at initial temperature $\tau = 0.1$, incrementing by 0.15 per retry up to three attempts. The prompt includes paired positive and negative encoding examples for numeric threshold conditions:

Listing 1: Prompt fragment for numeric comparison encoding.

```
% CORRECT: "creatinine clearance below 30"
{"op":"lt",
 "left": {"predicate":"creatinine_cl","args":[{"var":"P"}]},
 "right":{"const":30.0}}

% WRONG — do not use equality predicate:
{"predicate":"creatinine_cl","args":[{"var":"P"},{"const":30.0}]}
```

Despite explicit instruction, the model reliably encodes numeric thresholds as equality predicates. We apply a post-hoc *numeric recovery* pass (Algorithm 1) that uses keyword matching on the retained provenance field to restore comparison semantics.

Algorithm 1 Numeric Comparison Recovery

Require: IR $\mathcal{R}$, original text s
Ensure: IR $\mathcal{R}'$ with comparison operators restored
for each rule body element b in $\mathcal{R}_r$ **do**
  if b is a predicate call $p(\bar{x}, n)$, $n \in \mathbb{R}$ **then**
    $op \leftarrow \textsc{KeywordOp}(b.\texttt{src} \cup s)$ ▷ fragile: a comparison keyword anywhere in the provenance window can misfire on a dosage, count, or time-window argument (see CLI_004, Section 5)
    if $op \neq \emptyset$ **then**
      replace b with {`op`: op, `left`: $p(\bar{x})$, `right`: n}
    end if
  end if
end for
Apply same transformation to constraint $\mathcal{C}$ conditions **return** $\mathcal{R}'$

KeywordOp maps: {below, less than, under} → `lt`; {above, exceeds} → `gt`; {at least, or more} → `ge`; {at most, or less} → `le`.

3.4. Stage 2: Multi-Backend Compilation

3.4.1. Prolog Backend

Facts and rules are translated deterministically from NeSy-IR to SWI-Prolog [22] syntax. For each `must_not` constraint with condition ϕ and forbidden action α, the backend generates:

```
violation_c1(Vars) :- phi(Vars), alpha(Vars).
check_c1 :- \+ violation_c1(_).
```

Here `Vars` is schematic: it stands for the constraint's variable tuple, which the backend instantiates to the compiled predicate's actual arity (Listing 2 shows a generated instance of arity one).

All domain predicates receive `:- dynamic` declarations, implementing the open-world assumption and preventing existence errors when test facts are absent. A numeric body element {op:lt, left:$p(X)$, right:n} is rendered as `p(X, CC), CC < n` using a fresh intermediate variable.

3.4.2. Z3 Backend

Entity predicates become Z3 Boolean variables; numeric-valued predicates become Z3 `Real` or `Int` variables; comparison operators map directly to Z3 arithmetic relations. Rule encoding uses Clark's completion [6].

Definition 2 (Clark's Completion)**.** *Given rules* $\{H \leftarrow B_i\}_{i=1}^{k}$ *sharing head predicate* H*, Clark's completion adds the constraint* $H \equiv B_1 \vee \cdots \vee B_k$ *to the Z3 solver.*

Under Clark's completion, the head predicate H can be true only if at least one rule body B_i is satisfied. This contrasts with implication encoding ($B_i \Rightarrow H$), under which H may be true independently of all bodies, preventing

detection of structural contradictions (Section 4.3).

The detection guarantee we claim for this encoding holds within a specific fragment, and we state its scope here rather than leaving it implicit. We assume the compiled rule set is a set of Horn clauses evaluated under the closed-world assumption, over the finite domains declared in Π, with negation interpreted as failure and stratified so that no predicate depends negatively on itself (These are the conditions under which Clark's completion is the standard semantics for negation as failure and under which the biconditional $H \equiv \bigvee_i B_i$ faithfully captures derivability. Programs with unstratified or unfounded recursion, infinite or unbounded domains, or genuine first-order quantifier alternation fall outside this fragment; for such programs the UNSAT test remains sound as a contradiction witness but we do not claim completeness.). Within this fragment, an UNSAT result on $\mathcal{R}_r^{\text{Clark}} \wedge \phi_i \wedge \alpha_i$ is both a necessary and a sufficient witness that the violation condition is structurally unreachable; outside it, the test is best read as sound but not provably complete.

3.5. Stage 3: Formal Verification

For each `must_not` constraint c_i with condition ϕ_i and forbidden action α_i, Stage 3 submits the query

$$\text{Z3CHECK}\left(\mathcal{R}_r^{\text{Clark}} \wedge \phi_i \wedge \alpha_i\right) \tag{1}$$

where $\mathcal{R}_r^{\text{Clark}}$ denotes the Clark-completed rule set. The result is interpreted as:

$$\text{outcome}_i = \begin{cases} \text{SAT} & \text{violation checker is non-vacuous (correct)} \\ \text{UNSAT} & \text{violation is structurally unreachable (E2 error)} \\ \text{UNKNOWN} & \text{solver timeout or complexity limit} \end{cases} \tag{2}$$

A parallel Prolog VioTest—injecting domain-specific test facts via `pyswip` and querying `check_C`—provides a complementary empirical check.

3.6. Stage 4: CEGIS-Informed Repair with Deterministic Fallback

When verification returns UNSAT or UNKNOWN, Algorithm 2 runs the repair loop.

Algorithm 2 CEGIS-Informed Repair with Deterministic Fallback

Require: IR $\mathcal{R}$, NL spec s, $T = 7$
Ensure: Repaired IR $\mathcal{R}^*$ or failure
$\mathcal{R}^* \leftarrow \mathcal{R}$
for $t = 1, \ldots, T$ **do**
  $(v, cex) \leftarrow \text{Z3VERIFY}(\mathcal{R}^*)$
  if $v = \text{SAT}$ **then return** $\mathcal{R}^*$
  end if
  Build repair prompt π_t from $\mathcal{R}^*$, *cex*, s, temperature $\tau_t = 0.2 + 0.15t$ ▷ includes Clark proof + target-structure skeleton
  $\mathcal{R}' \leftarrow \text{LLM}(\pi_t)$; $\mathcal{R}' \leftarrow \text{NUMRECOVERY}(\mathcal{R}')$
  if $\text{VALIDATE}(\mathcal{R}')$ **then**
    $\mathcal{R}^* \leftarrow \mathcal{R}'$
  end if
end for
if $\text{Z3VERIFY}(\mathcal{R}^*) = \text{UNSAT}$ **and** $\text{ISE2}(\mathcal{R}^*)$ **then**
  $\mathcal{R}^* \leftarrow \text{DETE2REPAIR}(\mathcal{R}^*, s)$ ▷ deterministic reconstruction
end if
return $\mathcal{R}^*$

DETE2REPAIR strips all negation elements ($\neg X$) from the rule body to obtain a contradiction-free condition, infers the forbidden action predicate from NL keywords (*prescribe*, *administer*, *receive*, etc.), and rebuilds the constraint as $\phi_{\text{clean}} \Rightarrow \neg\alpha$. This procedure is invoked only after all LLM rounds have failed and the failure is confirmed to be structural under Clark's completion.

The ordering in Algorithm 2—seven LLM rounds first, then deterministic reconstruction—reflects how we ran the experiments, not how we would deploy the system. We kept the LLM rounds in the loop to *characterise* LLM repair behaviour on structural errors, and the result (Section 4) is that they never succeed. The engineering recommendation that follows is therefore the reverse of the experimental order: once an UNSAT certificate identifies

a failure as an E2 structural contradiction, the system should route it straight to DetE2Repair and skip the LLM rounds entirely, reserving LLM-guided revision for the semantic failures (E1, E3) where it is the appropriate tool. Under that policy the term "CEGIS-informed" refers to the counterexample (the UNSAT certificate) that triggers and shapes the deterministic repair, rather than to an LLM enumerate-and-test loop.

4. Results

4.1. Benchmark and Experimental Setup

We constructed a cross-domain benchmark of 39 natural-language knowledge specifications spanning three engineering-relevant domains. Clinical decision support (20 specifications): drug contraindications, dosing thresholds, safety monitoring protocols, and treatment decision rules derived from established clinical guidelines. AI course knowledge (10 specifications): definitional rules about model architectures, training conditions, regularisation requirements, and evaluation metric selection criteria. Legal compliance (9 specifications): simplified single-clause articles drawn from the GDPR (Articles 5–8, 17, 25, 28, 44) and China's Personal Information Protection Law (PIPL). Complexity is distributed across three levels: simple (single condition, no arithmetic, $N = 12$), medium (two to four conditions or one numeric threshold, $N = 20$), and hard (five or more conditions or compound arithmetic, $N = 7$). Each specification is accompanied by a gold-standard Prolog rule authored by the research team, used for qualitative validation.

We designed the evaluation around four research questions. **RQ1**: Can NeSy-IR-based compilation achieve syntax-valid multi-backend programs across diverse specifications? **RQ2**: What classes of semantic errors does Z3 formal verification expose beyond Prolog execution? **RQ3**: To what extent does the CEGIS-informed repair loop recover failing specifications? **RQ4**: Which domains and complexity levels produce which error types?

We implemented a direct baseline (**B1-Direct**): GLM-4-flash is given the NL specification with minimal format guidance (include `violation_c1` and `check_c1` predicates) and asked to generate complete SWI-Prolog code in a single pass, without NeSy-IR, numeric recovery, or formal verification. We are explicit about what this baseline does and does not establish. B1-Direct is an ablation, not a survey of the field: holding the extraction model fixed, it isolates the contribution of NeSy-IR plus the verification layer over a single, unstructured prompt to the same model. It is therefore the right comparison for the question "does the intermediate representation and the Clark-completion check buy anything beyond asking the model directly?", and the wrong comparison for the question "is this the best available natural-language-to-logic method?". We make no claim of superiority over richer extraction strategies—structured or chain-of-thought prompting, self-refinement, dedicated semantic parsers, autoformalization systems, or multi-pass LLM-to-code workflows—none of which we evaluate here. Where those methods improve the *extraction* step, they are largely orthogonal to our contribution: the verification layer would sit downstream of any of them, and the E2 failure mode it exposes is a property of the compiled rule, not of the prompt that produced it. A broader comparison against these stronger frontends is left as future work. B1-Direct was evaluated on $N = 45$ specifications: the 39 main-benchmark specifications, together with six additional specifications that were authored for the baseline study only and never entered the main pipeline batch—five clinical (CLI_006, CLI_007, CLI_013, CLI_017, CLI_024) and one legal (LEG_001). This asymmetry makes a head-to-head reading of the two columns less clean than it should be, because B1-Direct is credited over a slightly larger set than the pipeline. To remove that confound we additionally report B1-Direct restricted to exactly the 39 shared specifications, so that both systems are scored on an identical set. On the shared 39, B1-Direct syntax validity is 92.3% (36/39), has-violation-predicate 97.4% (38/39), and has-check-predicate 94.9% (37/39)—within a few points of the full-set figures and well below the pipeline's uniform 100%. The verification-capability rows are unchanged, since B1-Direct performs no formal verification on any set.

Implementation: GLM-4-flash API (VPN off; domestic Chinese endpoint); Z3 4.12 (`z3-solver`); SWI-Prolog 10.0 via `pyswip` 0.2.11; Python 3.10, Windows 11 laptop (Intel Core i7, 16 GB RAM). We report proportions with Wilson 95% confidence intervals (Wilson CIs) because the evaluation set is small.

Single-Frontend Scope and What Depends on It

We use one extraction and repair model, GLM-4-flash, throughout. This is a deliberate choice for a controlled study—it lets us attribute differences to the pipeline rather than to a changing frontend—but it means we should separate which of our findings are tied to that model and which are not. Two of our central results do not depend on the choice of LLM. First, that Clark's completion detects E2 structural contradictions while Prolog execution and open-world encoding do not is a property of the verification semantics applied to an already-compiled rule; any frontend that emits the same contradicting structure produces the same UNSAT certificate, and a better frontend that avoids the structure simply yields fewer E2 cases to detect. Second, that deterministic reconstruction repairs the

structural contradiction is likewise independent of the model, since DetE2Repair operates on the IR, not on the prompt. What *does* depend on GLM-4-flash is quantitative: the prevalence of each error class (E1, E2, E3), the overall pre-repair pass rate, and—most clearly—the observation that LLM-guided repair fails on all E2 cases. A stronger instruction-tuned or code-specialised model might lower the E1/E2 incidence at extraction or succeed at some E2 repairs that GLM-4-flash does not. We therefore read the error-rate numbers as specific to this frontend and the detect-and-deterministically-repair mechanism as model-independent; a multi-model study quantifying the former is left as future work.

4.2. Compilation Validity (RQ1)

Both the Prolog and Z3 backends achieve 100% syntax validity (39/39, Wilson 95% CI: [91.0%, 100.0%]) across all evaluated specifications. This confirms that the NeSy-IR schema provides a reliable backend-agnostic compilation target. However, syntax validity is necessary but insufficient: Stage 3 subsequently reveals that 10/39 (26%) of syntactically valid programs contain semantic or structural errors invisible to compilation. The main value of Stage 3 is precisely this gap—exposing programs that execute without errors but whose rule semantics are incorrect.

Figure 2 illustrates the stage-by-stage pass rates. The near-flat S1–S2 profile contrasts with the 26-point drop at S3, making the verification step the primary diagnostic layer of the pipeline.

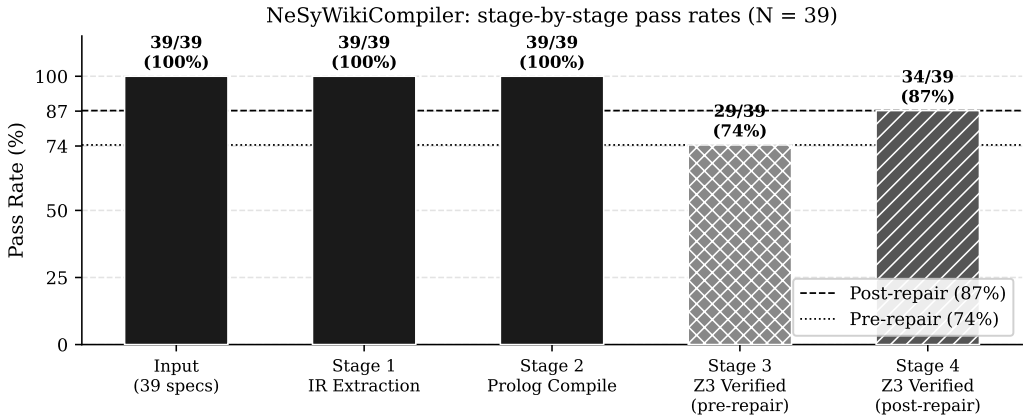

Figure 2. Stage-by-stage pass rates across 39 specifications (N). S1 and S2 both achieve 100%; Stage 3 Z3 verification exposes semantic and structural errors invisible to compilation. Post-repair Z3 pass rate reaches 87%.

4.3. Formal Verification and Error Taxonomy (RQ2)

Z3 formal verification (pre-repair) achieves a pass rate of 74.4% (29/39, Wilson 95% CI: [58.9%, 85.4%]). Analysis reveals three error classes, which do not all surface at the same stage. E1 is an extraction-stage error: it is measured on the raw LLM output and repaired by numeric recovery (Algorithm 1) in Stage 1, so E1 specifications reach Stage 3 already corrected and are not among the Z3 failures. The ten pre-repair Z3 failures are therefore the six E2 contradictions, one E3 argument reversal, and three specifications whose nested quantification exceeds the encoder's coverage (Section 4.4). Percentages below are proportions of the full benchmark ($N = 39$), not of those ten failures.

E1: Numeric comparison loss (15%, 6/39). The LLM encodes threshold conditions as equality predicates. For example, "creatinine clearance below 30 mL/min" becomes `creatinine_clearance(P, 30.0)` in the IR (equality), rather than the correct `CC` $<$ 30 (inequality). The numeric recovery pass (Algorithm 1) corrects all E1 instances detected in this evaluation.

E2: Structural contradiction (15%, 6/39). The rule body contains $\neg X$ while the constraint violation requires X, making the violation checker structurally non-triggerable. Consider the following E2 example from clinical specification CLI_002:

Listing 2: E2 structural contradiction in CLI_002 (simplified). The rule body requires NOT has_risk, while the violation requires has_risk.

```
% Rule body requires NOT has_risk
should_not_receive(E, nsaid) :-
    is_elderly(E),
    has_history(E, gi_bleeding),
    \+ has_risk(nsaid, R).      % <- requires NOT has_risk
```

```
% Violation requires has_risk <- contradicts rule body
violation_c1(E) :-
    should_not_receive(E, nsaid),
    has_risk(nsaid, R).          % <- requires has_risk
```

The consequence is that `violation_c1` can never be proved regardless of which facts are asserted. Under standard Prolog execution, this is indistinguishable from a correct rule base with no current violations: the system silently loses the ability to raise any alert. Under Z3 with Clark's completion, the formula $\phi \wedge \neg\phi_{\text{body}}$ reduces to a contradiction, and the solver returns UNSAT with an unsatisfiability certificate.

E3: Predicate argument reversal (3%, 1/39). The LLM transposes argument positions. In our dataset, `is_contraindicated(Insulin,patient)` appears in place of `is_contraindicated(Patient, insulin)`. This error is identifiable by semantic inspection but is not yet automatically corrected by the pipeline.

Figure 3 summarises the error distribution (left) and the E2 detection capability of three verification configurations (right). Neither Prolog execution nor Z3 with open-world implication encoding detects any of the six E2 cases; Clark's completion correctly identifies all six (6/6, Wilson 95% CI: [61.0%, 100.0%]).

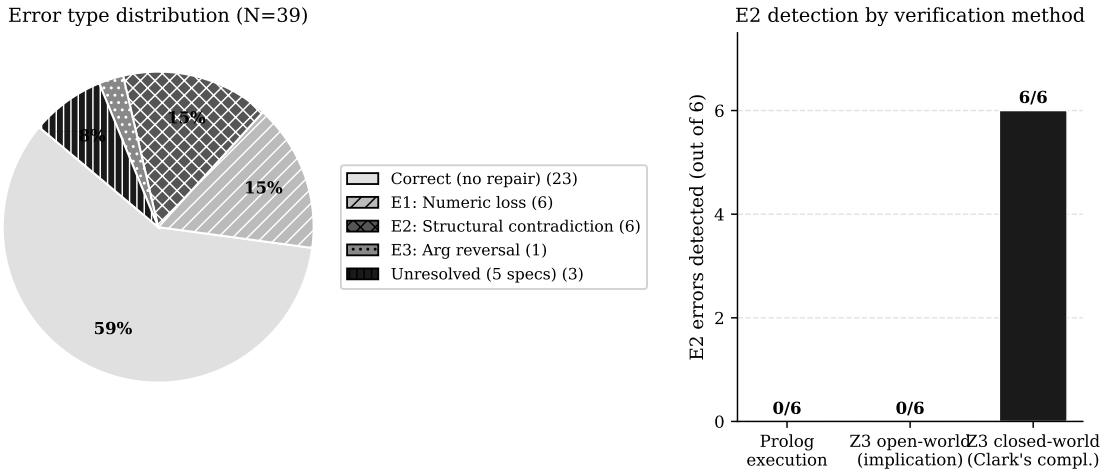

Figure 3. Left: error types across the 39 specifications, classified over the whole pipeline rather than at the pre-repair verification point ("Correct" denotes the 23 needing no repair; the 6 E1 cases were corrected in Stage 1, so the 29 passing Z3 pre-repair are these 23 plus those 6). **Right**: E2 structural contradiction detection by verification method. Only Z3 with Clark's completion (closed-world) identifies all six cases.

4.4. Repair Analysis (RQ3)

Post-repair, the Z3 SAT rate reaches 87% (34/39, Wilson 95% CI: [73.3%, 94.4%]), a 13-percentage-point improvement over pre-repair.

Table 1 reports CEGIS repair outcomes for all six E2 cases. LLM-guided repair achieves no successes across 42 total attempts (7 rounds $\times$ 6 specifications); in each round the model regenerates an IR centred on the same contradicting predicate structure. Deterministic reconstruction succeeds in 5/6 cases (83.3%, Wilson 95% CI: [43.6%, 97.0%]); the single failure is a schema validation issue (type mismatch), not a logical error.

Table 1. CEGIS repair outcomes for all six E2 structural contradictions. LLM repair: 7 rounds each, all returning UNSAT. Det. = deterministic E2 reconstruction.

Spec	Domain	LLM Rounds	Det. Repair	Final Z3
CLI_002	Clinical	7 / UNSAT	Schema fail [a]	UNSAT
LEG_002	Legal	7 / UNSAT	Pass	SAT
LEG_006	Legal	7 / UNSAT	Pass	SAT
LEG_008	Legal	7 / UNSAT	Pass	SAT
LEG_009	Legal	7 / UNSAT	Pass	SAT
LEG_010	Legal	7 / UNSAT	Pass	SAT
Success rate		0/42 (0%)	5/6 (83%)	5/6

[a] Type mismatch between reconstructed predicate and existing type declaration; not a logic error.

Across all six E2 cases, LLM-guided repair achieves 0 successes over 7 rounds per specification (0/42 total LLM repair attempts on E2 cases). In each round, the model regenerates an IR centred on the same intermediate predicate structure, despite explicit prompt instructions to use a different architecture and explicit identification of the contradicting predicate. Deterministic reconstruction (DetE2Repair) succeeds in 5/6 cases (83.3%, Wilson 95% CI: [43.6%, 97.0%]); the remaining case fails post-repair schema validation due to a type mismatch between the reconstructed predicate and the existing type declaration.

The five remaining non-SAT specifications after all repair attempts consist of three with complex nested quantification beyond the current Z3 encoder's coverage, one E3 (argument reversal), and one unresolved E2 (schema mismatch). The 13% residual failure rate is attributable to encoder limitations and one schema-level edge case, not to fundamental limits of the pipeline design.

4.5. Baseline Comparison

Table 2 reports the B1-Direct baseline versus NeSyWikiCompiler.

Table 2. NeSyWikiCompiler vs. B1-Direct baseline. The B1-Direct column reports the full $N = 45$ baseline set; the pipeline column reports the $N = 39$ main benchmark. Footnote [b] gives B1-Direct on the shared 39 specifications for a like-for-like comparison.

Metric	B1-Direct ($N = 45$) [a]	NeSyWikiCompiler ($N = 39$)
Compilation quality		
Prolog syntax validity	93.3% [82.1%, 97.7%]	**100%** [91.0%, 100%]
Has violation predicate	97.8%	**100%**
Has check predicate	95.6%	**100%**
Verification capability		
Formal Z3 verification (pre/post)	N/A	**74% / 87%**
E2 structural detection (6 cases)	0/6 = 0%	**6/6 = 100%** [61%, 100%]
E1 numeric correctness	not verifiable	**6/6 recovered**
Provenance traceability	None	**Yes (`src` field)**
Functional test (subsets)		
VioTest pass rate	37.5% [13.7%, 69.4%] (3/8)	80.0% [37.6%, 96.4%] (4/5)

[a] Compilation-quality figures are over the full $N = 45$ baseline set (39 shared specifications plus CLI_006, CLI_007, CLI_013, CLI_017, CLI_024 and LEG_001). [b] On the shared 39 specifications, B1-Direct syntax validity is 92.3% (36/39), has-violation 97.4% (38/39), has-check 94.9% (37/39); all verification-capability rows are identical to the full-set column, as B1-Direct performs no formal verification.

B1-Direct achieves 93.3% syntax validity (42/45, Wilson 95% CI: [82.1%, 97.7%]), indicating that GLM-4-flash is a capable Prolog programmer at the syntactic level. However, without a verification layer, B1-Direct has no mechanism to identify whether its violation predicates are semantically correct. Its E2 detection rate is 0% by construction, and its VioTest functional pass rate—whether the violation checker triggers correctly under injected test facts—is 37.5% (3/8), compared to 80% (4/5) for the pipeline on an overlapping subset. This gap reflects the structural consequence of generating code without a controlled predicate vocabulary: the checker predicate may be syntactically present but internally unreachable.

Figure 4 summarises the comparison.

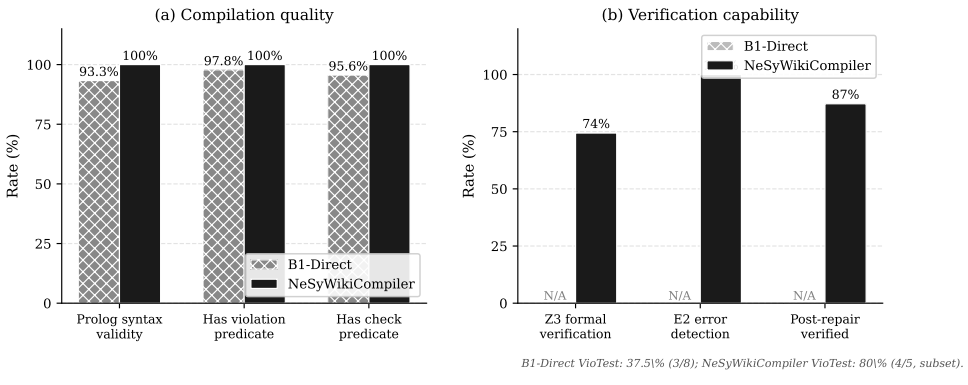

Figure 4. NeSyWikiCompiler vs. B1-Direct baseline. **Left**: compilation quality. **Right**: verification capability. B1-Direct cannot perform any formal verification and detects no E2 errors.

4.6. Domain-Specific Patterns and Engineering Overhead (RQ4)

Figure 5 shows Z3 pass rates before and after repair by domain.

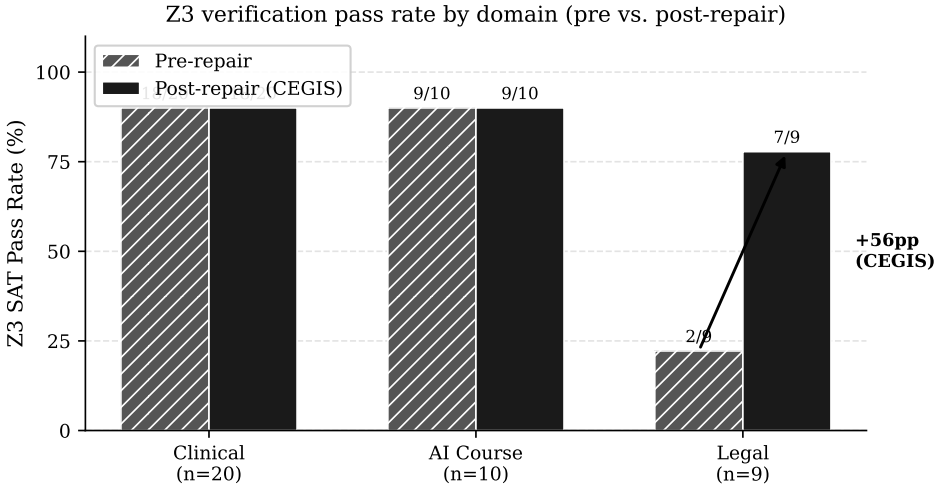

Figure 5. Z3 SAT pass rate by domain before and after CEGIS repair. In this small benchmark, the legal domain shows the largest improvement (+56 percentage points), driven by a high rate of E2 structural contradictions.

Clinical and AI course specifications both reach 90% pass rate (pre- and post-repair), while the nine legal specifications begin at 22% (2/9) and reach 78% (7/9) after repair. Within these nine legal specifications, the E2 rate is 5/9, against 1/20 in clinical and none of the 10 in AI course; we deliberately report these as raw counts rather than as domain error rates, because a denominator of nine cannot support a stable percentage. A plausible explanation, which we offer as a hypothesis rather than a finding, is that normative language drives the difference: legal specifications routinely express conditions through embedded exceptions ("unless X", "provided that Y") that the LLM tends to encode as rule-body negations, which then conflict with the positive assertion required by the violation checker, whereas clinical and AI course specifications in our set more often use direct affirmative-prohibitive constructions less prone to this inversion. This pattern is consistent with the examples we inspected and with prior observations on the difficulty of formalising hedged normative language, but confirming it would require a substantially larger legal corpus with full multi-clause articles and cross-references; we therefore treat it as a question raised by this pilot study, not as an established domain-level result.

Table 3 reports engineering overhead metrics.

Table 3. Engineering overhead per specification.

Aspect	Value
Stage 1 API calls (per spec, no repair)	1–3
Stage 1 latency (per call, observed range)	10–35 s
Total wall-clock time (39 specs, incl. delay)	≈25 min
Stage 2 Prolog compilation (per spec)	<100 ms
Stage 3 Z3 verification (per constraint)	<500 ms
CEGIS repair LLM calls (E2 cases)	up to 7
Deterministic repair (when triggered)	<1 s
Memory (Python process peak)	<500 MB
GPU requirement	None

Stages 2 and 3 together add less than one second per specification; the dominant cost is LLM API latency in Stage 1. The pipeline does not require GPU hardware and runs on standard laptop hardware.

5. Discussion

If this evaluation has one finding, it is the one we would rather not have to report: in this setting, the fact that a Prolog program compiles tells you almost nothing about whether it is correct. Of the 39 specifications that loaded without error on both backends, 10—just over a quarter—carried a semantic or structural defect that surfaced only under formal verification. B1-Direct shows what that means in practice. It reaches 93.3% syntax validity, a figure that looks reassuring on its own, and yet its violation checkers behave correctly on only 37.5% of the functional tests

we ran. The distance between those two numbers is the entire reason for a verification stage, and it is the part of this work we would defend most strongly.

The E2 case is the one we find most striking. A rule base with an E2 defect is behaviourally indistinguishable from a healthy one that simply has nothing to report: no alert fires, nothing is logged, and every Prolog query returns cleanly. There is, in fact, only one way to tell the two apart—to ask whether the violation predicate could be satisfied by any assignment of facts at all—and that is exactly the question Clark's completion lets us pose. By rewriting the head predicate as a biconditional over its bodies, completion makes the rule body the necessary and sufficient condition for the head. The move that hides the contradiction under implication encoding—letting the head hold independently of every body—is then no longer available. The finding that all six E2 cases are UNSAT under Clark's completion but SAT under implication encoding confirms the theoretical prediction and yields a blunt engineering recommendation: any knowledge compiler that means to verify LLM-extracted rules should use closed-world encoding for that verification.

The repair results carry a lesson of their own for anyone wiring a language model together with a solver. Across 42 attempts the model never once fixed an E2 case; round after round it returned an IR built around the very predicate structure we had just identified as broken. Handing it the counterexample, sketching the target structure, and raising the temperature did nothing to dislodge that prior within seven rounds. What this suggests is that when a fix turns on recognising a global logical property of the rule, asking the model to derive the fix is the wrong move: it needs scaffolding it does not have, such as a symbolic oracle that names the repair operation rather than leaving the model to rediscover it. Deterministic reconstruction does name it, clears the class at 83%, and is fast enough to serve as a practical default.

Two limitations of the numeric recovery procedure warrant attention. First, keyword matching on the provenance field can produce false positives when a numeric predicate argument is a dosage or time interval rather than a comparison threshold: for example, the predicate `within(P, saline, 3)` in specification CLI_004 (meaning "within 3 h") was incorrectly assigned `op:lt` because the specification's source text also contains the word "below" in a different clause. Second, the procedure is sensitive to the quality of the provenance field: if the LLM omits the source text, keyword matching falls back to the original specification, which may span multiple clauses with different semantics. A predicate-type annotation system—distinguishing numeric-threshold predicates from quantity predicates (dosages, counts, durations) in the NeSy-IR schema—would address both issues, since recovery could then fire only on arguments typed as thresholds and ignore the rest. Two concrete safeguards follow from the same idea: restricting the keyword search to the provenance span of the specific argument rather than the whole specification, and declining to rewrite an argument whose provenance field is empty rather than falling back to the full text. We also note a reporting gap: we observed the CLI_004 false positive by inspection but did not measure recovery false-positive and false-negative rates systematically, because doing so requires a gold-labelled set of which numeric arguments are thresholds. Building that annotation and quantifying recovery precision and recall is left as future work, and we caution that the procedure should be treated as a heuristic rather than a sound transformation until those rates are known.

This evaluation has several limitations beyond those already noted. The benchmark comprises 39 specifications, which is sufficient for failure-mode discovery and system validation but insufficient for statistically robust estimates of error rates across domains. The evaluation does not include a systematic execution correctness assessment against the gold-standard Prolog rules. It is worth being precise about what the Z3 SAT criterion does and does not certify: an SAT result establishes that the violation checker is non-vacuous—some fact assignment can trigger it—but it does not establish semantic equivalence to the intended rule. A checker can be non-vacuous and still fire on the wrong inputs, miss inputs it should catch, or encode a subtly different threshold than the gold standard. Closing this gap requires executing the compiled program and the gold-standard rule on a shared battery of positive, negative, and boundary fact assignments and comparing their query results, which we identify as the most important next step for strengthening the evaluation. The repair loop uses GLM-4-flash; stronger LLMs may achieve higher LLM repair success rates for E2 cases. A formal Lean 4 backend would provide machine-checkable proof certificates beyond Z3 decision procedures.

6. Conclusions

We have presented NeSyWikiCompiler, a four-stage neural-symbolic compilation pipeline for producing verifiable logic programs from natural-language knowledge specifications. The system achieves 100% syntax validity across Prolog and Z3 backends on a 39-specification cross-domain benchmark, and a Z3 formal verification pass rate of 87% after CEGIS-based repair.

The central engineering finding is that syntax-valid compilation is insufficient for knowledge engineering: 26%

of syntactically valid programs contain semantic or structural errors detectable only through formal verification. Among these, structural contradictions (E2, 15% prevalence) constitute the most severe failure class because they are invisible to Prolog execution and indistinguishable from correct rule bases with no current violations. Clark's completion detects all E2 instances in our evaluation; no other evaluated verification method does.

The practical implication for AI knowledge engineering systems is direct: a compilation pipeline that relies solely on syntactic executability and runtime testing cannot provide correctness assurances for rules extracted by LLMs. Formal verification using closed-world semantics should be included as a standard component of any production-grade knowledge compiler.

Funding

This research received no external funding.

Institutional Review Board Statement

Not applicable. This study did not involve human participants or animals; the clinical specifications used in the benchmark are rule statements derived from publicly available clinical guidelines and contain no patient data.

Informed Consent Statement

Not applicable.

Data Availability Statement

The source code, the 39 benchmark specifications together with their gold-standard Prolog rules, the generated NeSy-IR documents, the compiled Prolog programs, the B1-Direct outputs, the CEGIS repair logs, and the evaluation and figure-generation scripts are openly available under the MIT licence at https://github.com/aussie-bzhang/nesywikicompiler (archived at Zenodo, DOI: https://zenodo.org/records/21413550). The Z3 encodings are produced at run time from the NeSy-IR documents by the Z3 backend rather than stored as files; the corresponding verification verdicts are released in the results directory. A README lists the exact commands and pinned tool versions (GLM-4-flash, Z3 4.12, SWI-Prolog 10.0) needed to reproduce every reported table and figure. Proportions are reported with Wilson 95% confidence intervals ($z = 1.96$); no missing data arose, as all 39 specifications produced valid Stages 1 and 2 outputs.

Conflicts of Interest

The author declares no conflict of interest.

Use of AI and AI-Assisted Technologies

During the preparation of this work, the author used Claude Opus 4.8 for language editing and for improving the clarity of expression. After using this tool, the author reviewed and edited the content as needed and takes full responsibility for the content of the published article.

References

1. Hripcsak, G.; Clayton, P.D.; Pryor, T.A.; et al. The Arden Syntax for Medical Logic Modules. In Proceedings of the 18th Annual Symposium on Computer Applications in Medical Care (SCAMC), Washington, DC, USA, 4–7 November 1990; pp. 200–204.
2. Ohno-Machado, L.; Gennari, J.H.; Murphy, S.N.; et al. The GuideLine Interchange Format: A Model for Representing Guidelines. *J. Am. Med. Inform. Assoc.* **1998**, *5*, 357–372.
3. Yang, F.; Zhao, P.; Wang, Z.; et al. Empower large language model to perform better on industrial domain-specific question answering. In Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing: Industry Track, Singapore, 6–10 December 2023.
4. Chen, M.; Tworek, J.; Jun, H.; et al. Evaluating Large Language Models Trained on Code. *arXiv* **2021**. arXiv:2107.03374.
5. Karpathy, A. LLMs as Disciplined Wiki Curators. GitHub Gist, April 2026. Available online: https://gist.github.com/karpathy/442a6bf555914893e9891c11519de94f (accessed on 10 May 2026).
6. Clark, K.L. Negation as Failure. In *Logic and Data Bases*; Gallaire, H., Minker, J., Eds.; Springer: Boston, MA, USA , 1978; pp. 293–322.
7. Zelle, J.M.; Mooney, R.J. Learning to Parse Database Queries Using Inductive Logic Programming. In Proceedings of the 13th National Conference on Artificial Intelligence (AAAI), Portland, OR, USA, 4–8 August 1996; Vol. 2, pp. 1050–1055.

8. Zettlemoyer, L.S.; Collins, M. Learning to Map Sentences to Logical Form: Structured Classification with Probabilistic Categorial Grammars. In Proceedings of the 21st Conference on Uncertainty in Artificial Intelligence (UAI), Edinburgh, UK, 26–29 July 2005; pp. 658–666.
9. Yu, T.; Zhang, R.; Yang, K.; et al. Spider: A Large-Scale Human-Labeled Dataset for Complex and Cross-Domain Semantic Parsing and Text-to-SQL Task. In Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing (EMNLP), Brussels, Belgium, 31 October–4 November 2018; pp. 3911–3921.
10. Wu, M.; Szegedy, C.; Urban, J. Autoformalization with Large Language Models. In Proceedings of the Advances in Neural Information Processing Systems 35 (NeurIPS 2022), New Orleans, LA, USA, 28 November–9 December 2022; Vol. 35, pp. 32353–32368.
11. Solar-Lezama, A.; Tancau, L.; Bodík, R.; et al. Combinatorial Sketching for Finite Programs. In Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS), San Jose, CA, USA, 21–25 October 2006; pp. 404–415.
12. Gulwani, S. Automating String Processing in Spreadsheets Using Input-Output Examples. *ACM SIGPLAN Not.* **2011**, *46*, 317–330. https://doi.org/10.1145/1925844.1926423.
13. Alur, R.; Bodik, R.; Juniwal, G.; et al. Syntax-Guided Synthesis. In Proceedings of the IEEE Formal Methods in Computer-Aided Design (FMCAD), Portland, OR, USA, 20–23 October 2013.
14. Manhaeve, R.; Dumancic, S.; Kimmig, A.; et al. DeepProbLog: Neural Probabilistic Logic Programming. In Proceedings of the Advances in Neural Information Processing Systems 31 (NeurIPS 2018), Montreal, QC, Canada, 2–8 December 2018; Vol. 31, pp. 3749–3759.
15. Yang, Z.; Ishay, A.; Lee, J. NeurASP: Embracing Neural Networks into Answer Set Programming. In Proceedings of the 29th International Joint Conference on Artificial Intelligence (IJCAI), Yokohama, Japan, 11–17 July 2020 ; pp. 1755–1762.
16. Riegel, R.; Gray, A.; Luus, F.; et al. Logical Neural Networks. *arXiv* **2020**, arXiv:2006.13155.
17. Shortliffe, E.H.; Blois, M.S. The Computer Meets Medicine and Biology: Emergence of a Discipline. In *Biomedical Informatics: Computer Applications in Health Care and Biomedicine*, 3rd ed.; Shortliffe, E.H., Cimino, J.J., Eds.; Springer: New York, NY, USA, 2006 ; pp. 3–45.
18. Palmirani, M.; Governatori, G.; Rotolo, A.; et al. LegalRuleML: XML-Based Rules and Norms. In Proceedings of the Rule-Based Modeling and Computing on the Semantic Web, 5th International Symposium, RuleML 2011-America , Ft. Lauderdale, FL, USA, 3–5 November 2011; Lecture Notes in Computer Science; Vol. 7018, pp. 298–312.
19. Bench-Capon, T.; Araszkiewicz, M.; Ashley, K.; et al. A History of AI and Law in 50 Papers: 25 Years of the International Conference on AI and Law. *Artif. Intell. Law* **2012**, *20*, 215–319.
20. de Moura, L.; Bjørner, N. Z3: An Efficient SMT Solver. In Proceedings of the Tools and Algorithms for the Construction and Analysis of Systems (TACAS), Budapest, Hungary, 29 March–6 April 2008 ; Vol. 4963; pp. 337–340.
21. Zhang, B. Beyond RAG: LLM Wikis as Living Semantic Memory: Patterns, Empirical Findings, and Open Problems. Zenodo 2026. Available online: https://zenodo.org/records/20078453 (accessed on 16 May 2026).
22. Wielemaker, J.; Schrijvers, T.; Triska, M.; et al. SWI-Prolog. *Theory Pract. Log. Program.* **2012**, *12*, 67–96.