Verifiable Checks for Business Rule Consistency

Joseph Tafese
University of Waterloo
Waterloo, Canada
jetafese@uwaterloo.ca

Milad Hooshyar
Amazon
Seattle, USA
miladho@

Sam Bayless
Amazon
Seattle, USA
sabayless@

Nick Feng
Amazon
Seattle, USA
nicfeng@

Arie Gurfinkel
University of Waterloo
Waterloo, Canada
agurfink@uwaterloo.ca

***Abstract*—Maintaining consistency between natural language documentation of business rules and their evolving internal implementations is a significant challenge in large-scale systems. We present SIRNA, a tool and framework for checking such consistency using SMT solvers. Using the case study of cost calculations in tax domains, we demonstrate a three-part system that combines large language models (LLMs) with formal verification methods. SIRNA translates natural language documentation into candidate SMT formulas using LLMs, followed by checks to validate the translations. Then, corresponding business rules are converted into equivalent SMT representations and validated against the natural language formalizations. Our method is generalizable to domains where business logic exists in both natural language documentation and programmatic implementation. Compared to baseline evaluations, SIRNA significantly reduces the number of false positives and false negatives while offering explainability for its findings.**

I. INTRODUCTION

While automated reasoning has made significant progress in verifying implementation correctness against formal specifications, the gap between natural language (NL) and formal specifications remains a critical bottleneck. In compiler design, formal semantics must match language documentation. In API development, method behaviors must align with their documentation. In business systems, customer-facing documentation must accurately reflect system behavior. This alignment is particularly critical in billing and tax calculations, where documentation often serves as a legally binding contract with customers. Beyond customer-facing concerns, NL specifications are also the primary medium through which domain experts and policy designers articulate intended system behavior. Ensuring that this intent is faithfully realized by the implementation therefore requires a dedicated form of verification between NL documentation and executable artifacts. This is especially useful in business critical domains where the NL serves multiple stakeholders and the implementation handles consequential computations.

We introduce SIRNA,[1] a tool for verifying business rules that describe real-valued functions such as costs, rates and taxes. It is designed for users that maintain both NL documentation and implementations of the same rules in a domain specific language (DSL), a common business scenario. SIRNA integrates large language models (LLMs) with SMT-based verification to bridge the gap between these dual representations. NL documentation is automatically translated into SMT-LIB [1] formulas using Bedrock Guardrails Automated Reasoning checks [2]. Implementation code undergoes a sound translation into logically equivalent SMT-LIB. This encoding enables the use of standard SMT solvers to check whether the documented rules and implemented rules are semantically aligned. Crucially, our system does not only report consistency or inconsistency: it can be used to systematically enumerate the conditions under which the implementation diverges from the documentation. By exposing them, SIRNA helps users maintain business logic across representations, reconcile customer facing and internal representations, ensure compliance, and maintain trust.

The rest of the chapter is structured as follows: we provide an overview of SIRNA in Section II, describe our methodology in Section III, evaluation results in Section IV and conclude in Section VI.

II. OVERVIEW

We present SIRNA, a tool to check the consistency between natural language documentation (denoted as *NL*) and implementations (denoted as *DSL*) of business rules. SIRNA takes (*NL*, *DSL*) as its primary inputs. Beside the primary inputs, it also requires additional configurable parameters: (1) variable specifications consisting of a list of variable names, types, and natural language descriptions (see Section III-B), (2) variable bounds, a set of optional constraints/bounds on the variables (see Fig. 8), and (3) θ, a confidence threshold (see Section III-B). The variable specifications and variable bounds are specific to a given domain. θ is defaulted at $\frac{2}{3}$ but can be adjusted by the users.

SIRNA is designed to run in two phases for a given domain: a configuration phase and an evaluation phase. In the configuration phase, users configure the parameters SIRNA will use to evaluate *NL* and *DSL* for equivalence in their domain. In the evaluation phase, users perform consistency checking, with SIRNA, against batches of *NL* and *DSL* and identify or remediate semantic mismatches between them.

In this section, we explain the configuration phase. The evaluation phase will be shown in the Section III.

When SIRNA compares the input pair (*NL*, *DSL*) for equivalence, it can result in four common scenarios: (a) an inconsistency caused by underspecification in the equivalence check (e.g., missing legal ranges for values, or missing equivalences between variables), (b) an inconsistency is found that needs to be rectified, (c) the inconsistency is intentional and

[1] An Oromo word that translates to right, truthful or just.

https://doi.org/

This article is licensed under a Creative Commons Attribution 4.0 International License

Federal income tax rates for 2025

Tax rate	Taxable income threshold
14.5% [1]	on the portion of taxable income that is $57,375 or less, **plus**
20.5%	on the portion of taxable income over $57,375 up to $114,750, **plus**
26%	on the portion of taxable income over $114,750 up to $177,882, **plus**
29%	on the portion of taxable income over $177,882 up to $253,414, **plus**
33%	on the portion of taxable income over $253,414

(a) Natural Language.

```
1  let income = <input>
2  let sin = <input>
3
4  let p14_5 = 0.145 * (if income > 57375 then 57375 else income)
5  let p20_5 = if income > 57375
6     then 0.205 * ((if income > 114750 then 114750 else income) - 57375)
7     else 0
8  let p26 = if income > 114750
9     then 0.26 * ((if income > 177882 then 177882 else income) - 114750)
10    else 0
11 let p29 = if income > 177882
12    then 0.29 * ((if income > 253414 then 253414 else income) - 177882)
13    else 0
14 let p33 = if income > 253414 then 0.33 * (income - 253414)
15    else 0
16
17 let special = if sin < 900000000 then 1 else 0
18 let final_tax = special * (p14_5 + p20_5 + p26 + p29 + p33)
```

(b) Reference Implementation in a custom DSL.

Fig. 1: Running example: Canadian federal income tax calculation.

```
[Inconsistency Found]
Inputs: NL {Income = -200/29}, DSL{ income = 0, sin =
     900000000 }
Outputs: NL { FinalTaxes = -1 } DSL { final_tax = 0 }
```

Fig. 2: Customized Message for Inconsistency

should be ignored, or (d) no inconsistencies are found. Perhaps surprisingly, intentional inconsistencies (Case c) can occur when documentation omits complex implementation details to remain accessible to the reader. SIRNA provides concrete feedback to assist users to classify and resolve these scenarios.

To illustrate these scenarios, consider a simplified view of the Canadian tax code. We introduce the description of federal taxes in Canada as a running example. The natural language description is presented in Fig. 1a and a reference implementation in Fig. 1b, both describing how income tax is computed from personal income and Canadian Social Insurance Number (SIN). We wrote the reference implementation so that it handles the base rules and an exception, to enable a faithful representation of what our users see. As documented in [3], SINs that start with a 9 indicate that an individual is a temporary foreign worker whose taxes will be determined by different rules. The implementation captures the base rules in Fig. 1b from Line 4–Line 15, and the exception in Line 17.

In this example, SIRNA produces the detailed error message shown in Fig. 2. Note that the variable names are derived from the user provided variable specifications and the variables in the *DSL*. This error message identifies a potential inconsistency between *NL* and *DSL*, with an assignment to variables in which the two artifacts disagree. In this case, it is possible for the output to differ when `Income` from *NL* is not equivalent to `income` in *DSL* (i.e., underspecified). To resolve this case, SIRNA prompts the user for feedback about the expected equalities and ranges for input variables. Our specification language over conjunctions of predicates is used to facilitate this interaction (see Fig. 8).

Running SIRNA with the additional information generates a different output: when the SIN in the implementation starts with a 9, `final_tax` can be zero. This successfully identifies the exception that was present in *DSL* but not communicated in *NL*. To rectify this, the user has three choices: (1) either update the *NL* to describe the SIN exception (2) update the *DSL* to remove this exception (if it is truly erroneous), or (3) mark the discrepancy as intentional (suppressing future findings about this error from SIRNA).

To handle the case where users intend that there be certain discrepancies between *NL* and *DSL*, SIRNA provides a mechanism to enumerate all behavioral differences between the input pair. This is achieved by providing the user with a language for specifying conjunctions of predicates over variables to restrict inputs to (*NL*, *DSL*). In this way, SIRNA can operate at the same level of abstraction as the user without requiring knowledge of the implementation details. In our running example, the user provides feedback to SIRNA identifying that SINs starting with 9 are intentional exceptions, so that it will no longer report findings with SINs in this range. Running SIRNA one last time, it reports that the input pair has no inconsistencies.

III. SIRNA

As outlined in the overview, this section describes the evaluation phase of SIRNA where users evaluate the generated SMT specifications for (*NL*, *DSL*). SIRNA has three interacting components: formalizing the natural language documentation, encoding the implementation logic, and handling the consistency checks. We present the architecture diagram in Fig. 3. The yellow boxes represent contributions by either the user or the LLM. Blue boxes denote SMT files. The input files correspond to the pair (*NL*, *DSL*). We begin with a formal presentation of consistency checks in Section III-A. Then, we describe how *NL* and *DSL* are formalized in Section III-B and Section III-C respectively.

A. Consistency Checker

The Consistency Checker pipeline verifies that a formalized natural language (NL) documentation and its DSL implementation are semantically aligned. As shown in Fig. 3, the checker operates on two SMT encodings: one produced by the NL Autoformalization pipeline, and one obtained via a deterministic translation of the DSL implementation into SMT. The *Verifier*

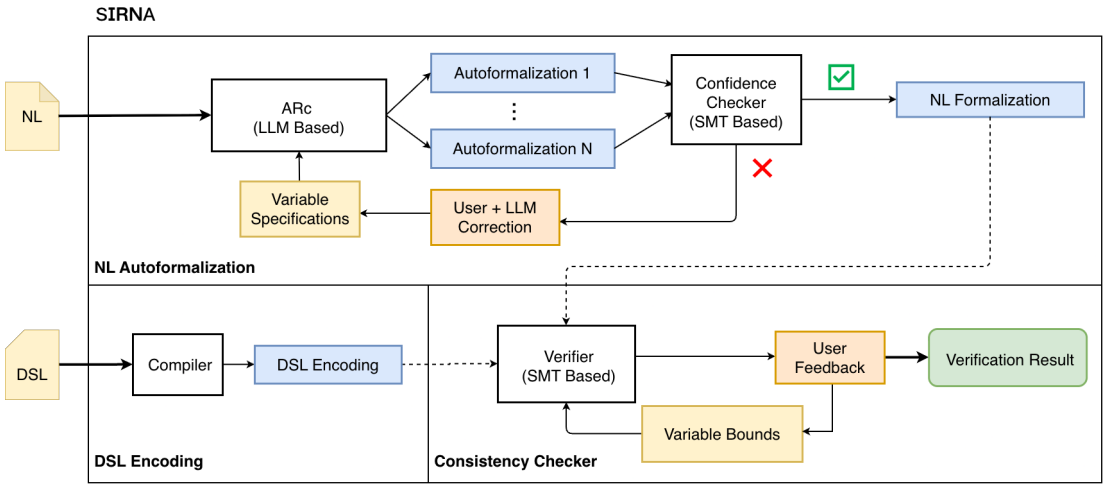


Fig. 3: Sirna Architecture Diagram.

determines whether these two specifications define the same behavior under a shared set of assumptions. If consistency holds, then the implementation conforms to the formalization we extracted from the NL. Otherwise, the Verifier produces a concrete counterexample that helps diagnose errors in either the NL documentation or the DSL implementation.

SIRNA is applicable when both NL documentation and DSL implementation define deterministic numeric computations that map relevant inputs to a real-valued output. We model this behavior using cost functions $f : \mathbb{R}^k \to \mathbb{R}$, where k is a fixed, sufficiently large dimension that subsumes all relevant inputs (e.g., income from our running example). Unused inputs may be ignored by the function. To determine equivalence between cost functions, pointwise equivalence is a natural starting point. Given two cost functions f and g, we write $f \equiv g \iff \forall \mathbf{x} \in \mathbb{R}^k,\ f(\mathbf{x}) = g(\mathbf{x})$.

As described in Section II, documentation and implementation for real-world business logic often differ in both parameter structure and intended input domain. To capture these differences, we introduce two orthogonal generalizations of direct equivalence: parameter mappings (relating variables in the NL formalization to the DSL encoding), and constraints, which restrict analysis to intended practical domains. A parameter mapping is a total function $\sigma : \mathbb{R}^k \to \mathbb{R}^k$ that rearranges, duplicates, or discards inputs as needed. We write $f(\sigma(\mathbf{x}))$ to denote the cost function obtained by applying σ to the inputs of f. In practice, cost computations are intended to operate only on a restricted domain of inputs (e.g., non-negative income or known exceptions). We model such assumptions using *constraints*, represented as a predicate $\rho : \mathbb{R}^k \to \mathbb{B}$ that characterizes the intended input domain. We now combine these two notions. Given a parameter mapping $\sigma : \mathbb{R}^k \to \mathbb{R}^k$ and constraints $\rho : \mathbb{R}^k \to \mathbb{B}$, to define when two cost functions f and g are *equivalent under mapping and constraints*, we write

$$f \equiv^{\rho}_{\sigma} g \iff \forall \mathbf{x} \in \mathbb{R}^k,\ \rho(\mathbf{x}) \Rightarrow f(\sigma(\mathbf{x})) = g(\mathbf{x}). \quad (1)$$

These notions provide a semantic foundation for reasoning about documentation and business logic that differ in interface structure and operational assumptions. We now formalize consistency of cost functions with respect to their outputs. We assume that cost functions are represented as First-Order Logic (FOL) formulas over the theory of linear real arithmetic (LRA). We use SMT-LIB [1] as the concrete representation format.

Verifier: We fix a standard model-theoretic semantics for SMT formulas over linear real arithmetic. For any SMT formula A with free variables $\vec{v}$, we write $[\![A]\!] \subseteq \mathbb{R}^{|\vec{v}|}$ to denote the set of all real-valued assignments to $\vec{v}$ that satisfy A. In particular, if $A(\mathbf{x}, y)$ has input variables $\mathbf{x}$ and output variable y, then

$$[\![A]\!] = \{\, (\mathbf{x}, y) \in \mathbb{R}^{|\mathbf{x}|} \times \mathbb{R} \mid A(\mathbf{x}, y) \text{ holds} \,\}.$$

We say that A is *functional* if for every assignment to $\mathbf{x}$ there exists exactly one real value y such that $A(\mathbf{x}, y)$ holds. In this case, A denotes a function $P : \mathbb{R}^{|\mathbf{x}|} \to \mathbb{R}$, defined by $P(\mathbf{x}) = y$ iff $P(\mathbf{x}, y)$ holds. Equivalently, $[\![A]\!]$ is the graph of P. Using the running example as an illustration, the SMT formula in Fig. 4 encodes a piecewise-linear tax computation, where `Income` is the input and `FinalTaxes` is the output.

a) Consistency Checking.: Let A and B be functional SMT formulas over variable set $V = V_A \cup V_B$, with output variables $O_A \in V_A$ and $O_B \in V_B$. Let $[\![A]\!] : \mathbb{R}^{|\mathbf{x}|} \to \mathbb{R}$ be the cost function mapping the inputs $\mathbf{x}$ in V_A to the output O_A, and similarly for $[\![B]\!]$ mapping to O_B. We say that A and B are equivalent over V when $\vdash (A \leftrightarrow B) \iff [\![A]\!] \equiv [\![B]\!]$. Let Φ be a parameter mapping between variables in V_A and V_B, and let Ψ denote constraints restricting variable domains. Viewed as SMT objects, Φ and Ψ relate variables in V such that in any satisfying assignment, inputs $\mathbf{x}$ and $\mathbf{x}' = \sigma(\mathbf{x})$ are such that $\rho(\mathbf{x})$ holds. To define when A and B are *equivalent under mapping and constraints*, we write

$$\forall\, V \cdot (A \wedge B \wedge \Phi \wedge \Psi \implies O_A = O_B). \quad (2)$$

If Eq. (2) is valid, the cost functions agree under Φ and Ψ. If Eq. (2) is not valid, there exists an assignment to the variables in V that satisfies A, B, Φ, and Ψ, but yields different output values for O_A and O_B. Such an assignment constitutes a *counterexample* to consistency: a concrete input instance that witnesses semantic disagreement between the cost functions. SIRNA produces error messages from such counterexamples that users can inspect to diagnose misalignment in the NL, implementation, or the mapping and constraint assumptions.

Theorem 1. *Let A and B be functional SMT formulas over variable sets V_A and V_B, with designated output variables $O_A \in V_A$ and $O_B \in V_B$. Let Φ be a parameter mapping between V_A and V_B, and Ψ be constraints over these variables. Let $f = [\![A]\!]$ and $g = [\![B]\!]$ denote the real-valued cost functions corresponding to A and B, and let σ and ρ denote the mappings and constraints for the inputs to f and g. Then*

$$f \equiv^{\rho}_{\sigma} g \iff \vdash \forall\, V\, .\, (A \wedge B \wedge \Phi \wedge \Psi \implies O_A = O_B).$$

Proof. We prove both directions of the equivalence. First, assume that f and g satisfy $f \equiv^{\rho}_{\sigma} g$. Let $s : V \to \mathbb{R}$ be any assignment satisfying the SMT formulas A, B, Φ, and Ψ. By the definitions of $[\![A]\!]$ and $[\![B]\!]$, the output variables evaluate to $O_A = f(\mathbf{x}')$ and $O_B = g(\mathbf{x})$ for some inputs $\mathbf{x}$ and $\mathbf{x}' = \sigma(\mathbf{x})$ with $\rho(\mathbf{x})$ holding. Since $f \equiv^{\rho}_{\sigma} g$, we have $f(\mathbf{x}') = g(\mathbf{x})$, which implies $O_A = O_B$ under s. Because s was an arbitrary satisfying assignment, the implication in Eq. (2) holds for all assignments, and thus Eq. (2) is valid.

Now assume that Eq. (2) is valid. Consider any input vector $\mathbf{x}$ such that $\rho(\mathbf{x})$ holds, and let $\mathbf{x}' = \sigma(\mathbf{x})$. By construction of the SMT encodings Φ and Ψ, there exists an assignment $s : V \to \mathbb{R}$ satisfying A, B, Φ, and Ψ that corresponds to these inputs, with outputs $O_A = f(\mathbf{x}')$ and $O_B = g(\mathbf{x})$. Validity of Eq. (2) then guarantees $O_A = O_B$ for this assignment, which implies $f(\mathbf{x}') = g(\mathbf{x})$. Since $\mathbf{x}$ was arbitrary subject to $\rho(\mathbf{x})$, we conclude that $f \equiv^{\rho}_{\sigma} g$. □

B. NL Autoformalization

Automatically formalizing natural language into logic (*autoformalization*) is inherently ambiguous [2], [4]. SIRNA breaks the autoformalization into two stages: (1) defining a set of logical variables and (2) producing a formalization over the variables. Concretely, SIRNA requires as input: (i) a NL description of a cost function (NL), (ii) a *variable specification* consisting of a set of SMT variables V_{SMT} paired with natural language descriptions of their intended meanings V_{Desc}, and (iii) a confidence threshold $\theta \in [0, 1]$. As output, SIRNA provides either a representative formalization or an explicit failure indicating why a formalization could not be produced. Fig. 4 shows a successful formalization. Following the NL Autoformalization pipeline presented in Fig. 3, we first describe how SIRNA's interface is implemented then discuss how users define variable specifications.

We re-use the redundant translation capability that *Automated Reasoning checks in Amazon Bedrock Guardrails* (ARc) [2] provides. Concretely, ARc generates multiple candidate formalizations of the same NL which are checked

```
(declare-const Bracket_1 Real)
(declare-const Bracket_2 Real)
(declare-const Bracket_3 Real)
(declare-const Bracket_4 Real)
(declare-const Bracket_5 Real)
(declare-const FinalTaxes Real)
(declare-const Income Real)

(assert (= Bracket_1 (* 0.145 (ite (<= Income 57375) Income 57375))))
(assert (= Bracket_2 (* 0.205 (ite (<= Income 57375) 0 (ite (<= Income
      114750) (- Income 57375) (- 114750 57375))))))
(assert (= Bracket_3 (* 0.26 (ite (<= Income 114750) 0 (ite (<= Income
      177882) (- Income 114750) (- 177882 114750))))))
(assert (= Bracket_4 (* 0.29 (ite (<= Income 177882) 0 (ite (<= Income
      253414) (- Income 177882) (- 253414 177882))))))
(assert (= Bracket_5 (* 0.33 (ite (<= Income 253414) 0 (- Income
      253414)))))
(assert (= FinalTaxes (+ Bracket_1 Bracket_2 Bracket_3 Bracket_4
      Bracket_5)))
```

Fig. 4: CRA Running Example: Autoformalization of NL in SMT.

for logical equivalence with the confidence checker using θ, the confidence threshold. Intuitively, θ enables SIRNA users to have confidence in the formalization of their cost functions that will be used for consistecy checks. Formally, let $T = \{t_1, \ldots, t_n\}$ be the set of SMT translations generated from NL. We define a binary relation $\equiv$ on T such that $t_i \equiv t_j$ iff t_i and t_j are logically equivalent. This induces a partition of T into equivalence classes $\mathcal{C} = \{C_1, \ldots, C_k\}$ such that

$$T = \bigcup \mathcal{C} = \bigcup_{i=1}^{k} C_i, \quad \forall r \in \{1, \ldots, k\},\ \forall t_i, t_j \in C_r,\ t_i \equiv t_j, \tag{3}$$

$$\forall r, s \in \{1, \ldots, k\},\ r \neq s \Rightarrow C_r \cap C_s = \emptyset, \tag{4}$$

and each C_r is maximal with respect to logical equivalence. Let $C_{\max} \in \mathcal{C}$ be an equivalence class of maximum cardinality. The confidence score is defined as $\texttt{Conf}(T) = \frac{|C_{\max}|}{|T|}$. An autoformalization is accepted iff $\texttt{Conf}(T) \geq \theta$; otherwise, the process fails and reports diagnostic information. In this way, θ sets the minimum level of semantic agreement required to resolve competing autoformalizations. Importantly, $\theta = 1$ (i.e., 100% agreement) does not guarantee that the autoformalization will align with user intent since systematic misinterpretations of NL may lead to unanimous but unexpected autoformalizations. Despite these limitations, we show in Section IV that with $\theta > 0.5$ (e.g. majority voting)[2], SIRNA reliably produces the expected formalization on our benchmarks and improved analysis precision and recall.

Defining variable specifications: In a typical use-case, users of SIRNA will evaluate many instances of (NL, DSL) for the same domain (e.g., taxes). We require users to provide the variable specifications as input to SIRNA, which can be a slow and a manual process. To mitigate this, we developed a prompt that generates variable specifications for NL cost descriptions given a single variable specification in their domain. We show

[2]Although ARc supports $\theta < 0.5$, SIRNA only considers the case where $\theta > 0.5$, which ensures that a single candidate translation can be selected by majority voting.

⟨*prog*⟩ ::= (⟨*assign*⟩)+

⟨*assign*⟩ ::= 'let' ⟨*identifier*⟩ '=' ⟨*expr*⟩ | 'let' ⟨*identifier*⟩ '=' '<input>'

⟨*expr*⟩ ::= ⟨*multexpr*⟩ ((+ | -) ⟨*multexpr*⟩)*

⟨*multexpr*⟩ ::= ⟨*unaryexpr*⟩ ((× | /) ⟨*unaryexpr*⟩)*

⟨*unaryexpr*⟩ ::= ⟨*number*⟩ | ⟨*identifier*⟩ | '(' ⟨*expr*⟩ ')' | ⟨*if*⟩

⟨*if*⟩ ::= 'if' ⟨*expr*⟩ ⟨*comp*⟩ ⟨*expr*⟩ 'then' ⟨*expr*⟩ 'else' ⟨*expr*⟩

⟨*number*⟩ ::= [0-9]+(.[0-9]+)?

⟨*identifier*⟩ ::= [a-zA-Z][_a-zA-Z0-9]*

⟨*comp*⟩ ::= > | < | ≡ | ≤ | ≥

Fig. 5: Business Language (BL) Grammar

```
(declare-const p29 Real)
(declare-const special Real)
(declare-const p20_5 Real)
(declare-const p26 Real)
(declare-const p33 Real)
(declare-const final_tax Real)
(declare-const sin Real)
(declare-const income Real)
(declare-const p14_5 Real)
(assert (= p14_5 (* 0.145 (ite (> income 57375) 57375 income))))
(assert (= p20_5 (ite (> income 57375) (* 0.205 (- (ite (> income
      114750) 114750 income) 57375)) 0)))
(assert (= p26 (ite (> income 114750) (* 0.26 (- (ite (> income 177882)
      177882 income) 114750)) 0)))
(assert (= p29 (ite (> income 177882) (* 0.29 (- (ite (> income 253414)
      253414 income) 177882)) 0)))
(assert (= p33 (ite (> income 253414) (* 0.33 (- income 253414)) 0)))
(assert (= special (ite (< sin 900000000) 1 0)))
(assert (= final_tax (* special (+ (+ (+ (+ p14_5 p20_5) p26) p29)
      p33))))
```

Fig. 6: CRA Running Example: SMT encoding of BL program SMT.

the prompt in Fig. 9. This allows users of SIRNA to analyze dozens or hundreds of (NL, DSL) pairs, therefore amortizing the manual effort in practice. We show a concrete example for the generated variable specification of our running example in Fig. 10.

C. DSL Encoding

The final component in Fig. 3 is the pipeline that encodes the DSL in SMT. In this section, we present the motivation for our Business Language (BL), its grammar and an algorithm that encodes BL programs to SMT. The BL grammar is loop free, recursion free, and does not support memory allocation.

The design of BL is informed by cost calculations and business logic across financial services and telecommunications providers. Common patterns in business logic include percentage-based charges (e.g., payment processing costs), tiered pricing (where rates change based on transaction volume or amount) and conditional rules (different rates for different customer categories). Public examples include PayPal's checkout processing fees (3.49% + $0.49 for commercial transactions in the United States [5]), Stripe's volume-based pricing (declining percentages as volume increases), and AWS's tiered data transfer pricing. We present the grammar for BL in Fig. 5. While commercial platforms often use proprietary languages for business logic, our grammar captures the essential computational patterns: arithmetic expressions for calculating costs or taxes, conditionals for implementing tiers and rules, and let-bindings for breaking down complex calculations.

To encode BL programs to SMT, we present the algorithm in Algorithm 1. We show the generated SMT file for our running example program in Fig. 6. Each let-binding is translated into an equality in the SMT formula. Conditional statements (if-then-else) are directly mapped to `ite` in SMT. Arithmetic and boolean operations are also mapped directly to their SMT counterparts. Since the DSL is loop and recursion free, the translation always results in a finite, quantifier-free SMT formula.

IV. EVALUATION

In this section, we present the evaluation of SIRNA where users evaluate the generated SMT specifications for (NL, DSL). We implemented the three components of SIRNA (see Section III) in Python. For NL Autoformalization (see Section III-B), we used the Amazon Automated Reasoning Checks API [2] with a 120-second timeout to compute SMT formulas from natural language, applying majority voting over translations from 3 LLMs (2 Sonnet 3.7 models and 1 Sonnet 3.5 model). We used Z3 for consistency checking between SMT translations (see Section III-A).

We evaluate SIRNA to answer the following research questions: (RQ1) Is SIRNA effective at determining business logic consistency between NL and DSL? (RQ2) Does SIRNA provide useful explanations when it detects inconsistency? To answer the two research questions, we evaluate SIRNA on a proprietary dataset of 369 real pairs (NL, DSL) from an Amazon internal system. In this context, DSL refers to the proprietary language used by Amazon teams. Due to privacy concerns and regulations, we are unable to disclose details about the internal dataset. To transparently demonstrate effectiveness, we also evaluate on an additional 313 public [6] pairs of (NL, DSL). In this case, NL comes from three domains: USPS shipping rules, IRS tax rules, and CRA tax rules. DSL refers to BL (described in Section III-C) that is LLM-generated from NL.

Amazon Dataset: On the Amazon proprietary dataset of 369 (NL, DSL) pairs, we compared SIRNA against an LLM-as-a-Judge [7] (LLMaJ) baseline using Claude Sonnet 3.7 [8] with chain-of-thought reasoning [9]. We show an example prompt for the LLMaJ setup in Fig. 7.

For RQ1, we ran both methods on each instance, producing a consistency label: "positive", "negative", "error". Positive indicates detected inconsistency, negative indicates proved equivalence, and error indicates pipeline failure (e.g., a failure of autoformalization). We compare predictions against ground truth labels from domain experts in Table I. SIRNA achieves 100% precision and 94.3% recall, outperforming LLMaJ on both metrics. The difference is particularly pronounced in recall: SIRNA detects 94.3% of genuine inconsistencies (with only 3 formalization failures), identifying 23 additional incon-

Algorithm 1 Encode DSL Program to SMT

Require: Program P in the DSL grammar
Ensure: SMT-LIB encoding of P

function ENCODE(P)
  $defs$, $exprs \leftarrow$ empty list
  for each assignment a in P **do**
    ($defs_a$, $exprs_a$) $\leftarrow$ ENCODEASSIGN(a)
    append $defs_a$ (resp. $exprs_a$) to $defs$ (resp. $exprs$)
  end for
  return ($defs$, $exprs$)
end function
function ENCODEASSIGN(a)
  $a \leftarrow$ "let $x = e$"
  if e is an input **then**
    return (`declare-const` x `Real`), $\emptyset$
  else
    $expr_smt \leftarrow$ ENCODEEXPR(e)
    return (`declare-const` x `Real`), (`assert (=` x $expr_smt$`)`)
  end if
end function
function ENCODEEXPR(e)
  if e is a number or identifier **then**
    return e
  else if e is a parenthesized expression **then**
    return ENCODEEXPR(inner expression)
  else if e is a binary operation e_1 op e_2 **then**
    return (op ENCODEEXPR(e_1) ENCODEEXPR(e_2))
  else if e is a conditional "if e_1 comp e_2 then e_3 else e_4" **then**
    $cond \leftarrow$ ENCODECOMP(e_1, comp, e_2)
    return (ite $cond$ ENCODEEXPR(e_3) ENCODEEXPR(e_4))
  end if
end function
function ENCODECOMP(e_1, comp, e_2)
  return (comp ENCODEEXPR(e_1) ENCODEEXPR(e_2))
end function

```
Document Logic Consistency Verification Task
<role>
You are a formal verification expert tasked with checking the
      consistency
between documentation describing a cost calculation expressed in natural
language and its corresponding Domain Specific Language (DSL).
</role>
<instructions>
1. Determine whether the Natural Language Policy and the Domain Specific
Language (DSL) are semantically equivalent and consistent with each
      other given:
a. Data Schema (Variable Definitions)
The following variables are available for the Documented calculation:
${DATA_KEYS}
b. Natural Language Documentation
This is the human-readable documentation that describes how values
      should be
calculated:
${NL}
c. Code implementation of the logic in a DSL
{DSL Grammar Documentation}
DSL implementation to verify:
${Implementation}
2. Verification Requirements
You must check for:
a. Logical EquiQvalence: Do the natural language policy and DSL
      decision table express the same calculation logic?
b. Completeness: Does the DSL cover all scenarios mentioned in the
      natural language document?
3. Consistency: Are there any contradictions between the documented
      description and the DSL rules?
4. Variable Usage: Are variables used correctly according to their
      definitions in the data schema?
5. Mathematical Correctness: Do the formulas and calculations match
      between both representations?
${AdditionalDomainSpecificRules}
3. Response Format
Analyze both representations carefully, then provide your answer in the
following format:
<thinking>
Explain your step-by-step analysis:
- What the natural language document states
- What the DSL encodes
- Key points of comparison
- Any inconsistencies or issues found
</thinking>
<final_response> Yes </final_response>
OR
<final_response> No </final_response>
<\instructions>
<note>
Important: Answer Yes if the policy and DSL are semantically equivalent
      and consistent.
Answer No if there are any inconsistencies, contradictions, or logical
      differences.
Your final_response MUST be exactly "Yes" or "No" (case-sensitive)
<\note>
```

Fig. 7: An LLMaJ Example Prompt. This prompt has been slightly edited to remove references to the proprietary DSL and to the specific use case within Amazon, but remains representative of the one used in our experiment.

sistencies that LLMaJ misses. In contrast, LLMaJ misses over 40% of inconsistencies. This answers RQ1 positively: SIRNA is more effective at determining consistency than LLMaJ.

For RQ2, domain experts manually reviewed all inconsistencies detected by SIRNA. The review confirmed that all 50 reported inconsistencies are genuine. Based on SIRNA's diagnostic feedback, experts determined that 13 cases were unintended inconsistencies; resolved by updating either the documentation or implementation. The remaining 37 cases were confirmed as intended inconsistencies (see Section II for discussion), helping clarify the intended semantics of the business logic. We count both intended and unintended inconsistencies as true positives since they represent real differences between NL and DSL confirmed by manual review. This answers RQ2 positively: SIRNA produces actionable explanations that enable domain experts to diagnose root causes and take corrective action. We can see the value of these formally explainable results when comparing to the LLMaJ approach. While the LLMaJ is also successful at discovering most of the unintended inconsistencies (10 of 13), it provides no guarantees of the correctness of its findings (and in fact it hallucinated 5 false positives).

***Public Datasets**:* While the Amazon dataset demonstrates that SIRNA is effective in practice, it remains proprietary. To provide a transparent and reproducible evaluation, we

Method	Precision	Recall	F1	Accuracy	Err.	TN	FN	TP	FP
LLMaJ	85.7%	56.6%	68.2%	92.4%	0	313	21	30	5
SIRNA	100%	94.3%	97.1%	99.2%	3	316	0	50	0

TABLE I: Comparison of SIRNA and LLMaJ on the Amazon dataset. Positive (P) denotes inconsistency detected; Negative (N) denotes consistency proved or assumed. Errors are counted as false negatives.

Dataset	Con.	Incon.	Err.	Dataset	Con.	Incon.	Err.
CRA	0	14	0	IRS	76	1	11
CRA (SIN)	14	0	0	USPS	211	0	0

TABLE II: SIRNA results on public benchmarks with 66% confidence threshold.

introduce three public benchmarks of 311 instances spanning different domains: USPS shipping rules (211), IRS tax rules (88), and CRA tax rules (14). Unlike the Amazon dataset, these benchmarks use LLM-generated (see prompt in Fig. 11) *DSL* implementations paired with publicly available rule descriptions.

The CRA dataset comprises 14 tax bracket descriptions [10] augmented with a SIN-dependent exception as shown in the running example: all pairs are inconsistent without this exception, and consistent once acknowledged. The IRS dataset combines federal and state tax rules [11], converted from CSV to NL using the script in Fig. 12. The USPS dataset consist of weight-based shipping rates from [12]. Unlike the CRA dataset, the pairs in IRS and USPS datasets are expected to be equivalent.

Benchmark	Autoformalization Results		
mo/single	t1 (MB)	t2 (uses min)	t3 (correct)
il/joint	t1 (correct)	t2 (GF)	t3 (MF)
ga/single	t1 (correct)	t2 (GF)	t3 (MF)
ms/single	t1 (GF)	t2 (correct)	t3 (MF)
ny/single	t1 (MF)	t2 (MC)	t3 (MC)
sc/joint	t1 (MD)	t2 (MD)	t3 (MF)
hi/single	t1 (GF)	t2 (MC)	t3 (correct)
us/joint	t1 (MF)	t2 (MC)	t3 (MD)
us/single	t1 (MD)	t2 (MD)	t3 (MF)
ca/single	t1 (correct)	t2 (MF)	t3 (MC)
ms/joint	t1 (MF)	t2 (MF)	t3 (MF)
nj/single	t1 (MF)	t2 (MC)	t3 (MC)

TABLE III: IRS benchmarks with failed translations by the autoformalization pipeline presented in Section III-B. Each row represents a specific benchmark and the three translations (t1, t2, t3) that were generated. We annotate, in brackets beside each translation the autoformalization result: (correct) means the autoformalization is correct; (MF) means the autoformalization is missing the FinalTaxes variable; (MB) means there are missing intermediate variables for brackets (Bracket_i's); (MC) means there are missing constraints on the intermediate brackets (Bracket_i's); (MD) means the autoformalization uses functions with missing definitions; (GF) means general failures due to spurious constraints on input and output variables. Benchmarks in bold refer those that satisfy the confidence threshold ($\theta = 66\%$) but are ultimately incorrect formalizations.

To answer RQ1, we ran SIRNA on all 313 public benchmarks. Table II summarizes the results. On CRA, SIRNA correctly identifies all 14 benchmarks as inconsistent prior to modeling the SIN exception, and proves all 14 consistent once the exception is accounted for. On IRS, SIRNA proves 76 benchmarks consistent, reports 1 inconsistency, and rejects 11 at the autoformalization stage. On USPS, SIRNA proves all 211 benchmarks consistent. These results support a positive answer to RQ1: when autoformalization succeeds, SIRNA reliably determines semantic consistency.

Benchmark	Autoformalization Results		
ab	t1 (MC)	t2 (correct)	t3 (correct)
bc	t1 (MC)	t2 (correct)	t3 (correct)
federal	t1 (MF)	t2 (correct)	t3 (correct)
mb	t1 (MF)	t2 (correct)	t3 (correct)
nb	t1 (MF)	t2 (correct)	t3 (correct)
nl	t1 (correct)	t2 (correct)	t3 (MC)
ns	t1 (correct)	t2 (correct)	t3 (correct)
nu	t1 (MF)	t2 (correct)	t3 (correct)
nwt	t1 (MF)	t2 (correct)	t3 (correct)
on	t1 (MF)	t2 (correct)	t3 (correct)
pei	t1 (correct)	t2 (correct)	t3 (correct)
qc	t1 (MF)	t2 (correct)	t3 (correct)
sk	t1 (correct)	t2 (correct)	t3 (correct)
yt	t1 (MC)	t2 (correct)	t3 (correct)

TABLE IV: CRA benchmarks with their corresponding autoformalization results using the autoformalization pipeline presented in Section III-B. Each row represents a specific benchmark and the three translations (t1, t2, t3) that were generated. Autoformalization results are annotated as follows: (correct) means the autoformalization is correct; (MF) means the autoformalization is missing the FinalTaxes variable; (MC) means there are missing constraints on the intermediate brackets (Bracket_i's).

To answer RQ2, we analyzed the artifacts produced by SIRNA for rejected or inconsistent cases. Across the IRS and CRA benchmarks, failures fall into three dominant, user-actionable categories: (1) omitted output variables (e.g., `FinalTaxes`), (2) unconstrained bracket variables (allowing negative or undefined values), and (3) unintended operations introduced during translation (e.g., spurious `min`/`max` calls). The single IRS inconsistency in Table II is a false positive caused by a unanimous but incorrect autoformalization, illustrating a known limitation of agreement-based validation. A full breakdown of autoformalization failures for IRS are shown in Table III. We also show all the autoformalization results for CRA in Table IV. These diagnostics enable users to refine variable specifications or constraints, supporting a positive answer to RQ2: SIRNA provides meaningful, actionable explanations rather than opaque failures.

Majority Voting Analysis: SIRNA computes confidence θ for the autoformalized SMT translation and uses majority voting ($\theta > 0.5$) to filter out unreliable translations (see Section III-B). Compared to using a single LLM for translation, majority voting improves precision and accuracy. Precision measures the percentage of correctly predicted instances among those SIRNA labels, while accuracy measures the percentage among all instances. On the IRS dataset, majority voting improved precision by 5.2%, from 71/76 to 75/76, and accuracy by 4.5%, from 71/88 to 75/88. On other datasets, majority voting does not negatively impact precision or accuracy. Table V shows the complete confusion matrices comparing single-LLM translation versus majority voting across all public datasets.

Method	**output**		**Metrics**	
	Non-Equiv	Equiv	Precision	Accuracy
Single LLM	TP: 0 FP: 5	FN: 0 TN: 71	93.4%	80.6%
Majority Voting	TP: 0 FP: 1	FN: 0 TN: 75	98.6%	85.2%
Single LLM	TP: 14 FP: 0	FN: 0 TN: 0	100%	100%
Majority Voting	TP: 14 FP: 0	FN: 0 TN: 0	100%	100%
Single LLM	TP: 0 FP: 0	FN: 0 TN: 14	100%	100%
Majority Voting	TP: 0 FP: 0	FN: 0 TN: 14	100%	100%
Single LLM	TP: 0 FP: 0	FN:0 TN: 211	100%	100%
Majority Voting	TP: 0 FP: 0	FN: 0 TN: 211	100%	100%

TABLE V: Confusion matrix comparison: Single LLM vs. Majority Voting on public datasets. Each pair of rows represents a different dataset (IRS, CRA, CRA (sin), USPS) and the performance of the two methods. TP: true positive (correctly predicted inconsistency), TN: true negative (correctly predicted consistency), FP: false positive (incorrectly predicted inconsistency), FN: false negative (incorrectly predicted consistency). Precision = (TP+TN)/(TP+TN+FP+FN), Accuracy = (TP+TN)/(TP+TN+FP+FN+unlabeled).

V. RELATED WORK

Catala [13] and DCR graphs [14] are languages for formalizing normative text: Catala targets legal computations such as tax code, DCR graphs the control flow of normative processes. The tool closest to SIRNA is ContractCheck [15], which also discharges its checks with SMT. While these tools handle large, complex systems, they do not address the challenge of ensuring consistency between multiple, independently maintained artifacts. SIRNA operates in a setting that handles separate, independently maintained artifacts, and provides a systematic approach to verifying their consistency. We introduce public benchmarks [6] as a step towards filling this gap in the literature.

```
[
 {
   "type": "outputs",
   "variables": ["final_tax", "FinalTaxes"]
 },
 {
   "type": "equals",
   "variables": ["Income", "income"]
 },
 {
  "type": "ranges",
   "variables": ["Income"],
   "min": 0
 }
]
```

(a) Sensibility Bounds.

```
[
 {
   "type": "outputs",
   "variables": ["final_tax", "FinalTaxes"]
 },
 {
   "type": "equals",
   "variables": ["Income", "income"]
 },
 {
  "type": "ranges",
   "variables": ["Income"],
   "min": 0
 },
 {
  "type": "ranges",
  "variables": ["sin"],
  "max": 900000000
 }
]
```

(b) Sensibility + Exception Bounds.

Fig. 8: CRA Running Example: Bounds.

Translating natural language into formal logic with LLMs has also been explored extensively. nl2spec [16], NL2TL [17], and NL2LTL [18] translate requirements into temporal logics, and Mendoza et al. [19] pair LLM translation with model checking. LINC [4] combines LLMs with first-order provers for logical reasoning. SIRNA reuses the redundant-formalization and semantic-equivalence checking of Automated Reasoning checks [2] for its NL-to-SMT stage.

VI. CONCLUSION

We have presented SIRNA, a tool for verifying the consistency of business logic representations using SMT solvers. SIRNA enables users to continuously check that documentation and implementation remain aligned, proactively identify inconsistencies before they impact customers, and scaling efficiently as cost structures or business rules grow in complexity. Through our evaluation, we demonstrated that SIRNA effectively integrates LLM-guided autoformalization with formal reasoning, producing concrete and diagnosable traces of inconsistencies. By automating the verification of (*NL*, *DSL*) pairs, SIRNA reduces reliance on manual audits by providing a systematic and semi-automated approach to maintaining correctness in evolving business logic.

```
<role>
You are a helpful logician who extracts logical variables from natural
      language according to a given schema.
</role>
<instructions>
Write a json file that captures variable definitions and their
      descriptions using the following schema:
<schema>
{
  "variables": [
    ...
    {
      "name": "Bracket_i",
      "type": "NUMBER",
      "description": "Taxes for Bracket i. Always included in the sum;
            0 if income does not reach this bracket."
    },
    ...
  ]
}
</schema>
</instructions>
<example>
<input>
<description>
Alberta income tax rates for 2025
Tax rate Taxable income threshold
8% on the portion of taxable income that is $60,000 or less, plus
...
</description>
</input>
<output>
{
  "variables": [
    ...
    {
      "name": "Bracket_1",
      "type": "NUMBER",
      "description": "Taxes for Bracket 1: 8% on the portion of income
            up to $60,000. Always included in the sum; 0 if income does
            not reach this bracket."
    },
    ...
  ]
}
</output>
</example>
Now you generate the output for the following input. Please ensure to
      also have
<output> tags in your final response.
<input>
<description>
${description}
</description>
</input>
```

Fig. 9: Variables Description Generation Example Prompt

```
{
  "types": [],
  "variables": [
    {
      "name": "Income",
      "type": "NUMBER",
      "description": "Total taxable income for Canadian federal 2025
            tax calculation."
    },
    {
      "name": "Bracket_1",
      "type": "NUMBER",
      "description": "Taxes for Bracket 1: 14.5% on the portion of
            income up to $57,375. Always included in the sum; 0 if
            income does not reach this bracket."
    },
    {
      "name": "Bracket_2",
      "type": "NUMBER",
      "description": "Taxes for Bracket 2: 20.5% on the portion of
            income over $57,375 up to $114,750. Always included in the
            sum; 0 if income does not reach this bracket."
    },
    {
      "name": "Bracket_3",
      "type": "NUMBER",
      "description": "Taxes for Bracket 3: 26% on the portion of income
            over $114,750 up to $177,882. Always included in the sum; 0
            if income does not reach this bracket."
    },
    {
      "name": "Bracket_4",
      "type": "NUMBER",
      "description": "Taxes for Bracket 4: 29% on the portion of income
            over $177,882 up to $253,414. Always included in the sum; 0
            if income does not reach this bracket."
    },
    {
      "name": "Bracket_5",
      "type": "NUMBER",
      "description": "Taxes for Bracket 5: 33% on the portion of income
            over $253,414. Always included in the sum; 0 if income does
            not reach this bracket."
    },
    {
      "name": "FinalTaxes",
      "type": "NUMBER",
      "description": "Sum of Bracket_1 through Bracket_5 in numerical
            order. Includes zeros from brackets that do not apply.
            Calculation must always add all five brackets explicitly."
    }
  ],
  "rules": [
    {
      "id": "DFV12ES58TNM",
      "expression": "(>= FinalTaxes 0)",
      "translation": "FinalTaxes is greater or equal to 0"
    }]
}
```

Fig. 10: CRA Running Example: Variable Specification.

```
<role>
You are a tax expert who writes tax programs using BL
</role>
<instructions>
1. Write a program to compute how much taxes are due in the output variable
(fee_bl) using the Business Language BL. The Antlr grammar for BL is provided
below:
<grammar>
prog: assigns ;
assigns : (assign)+ ;
...
</grammar>
</instructions>
<example>
<input>
<description>
Federal income tax rates for 2025
Tax rate,Taxable income threshold
14.5%,on the portion of taxable income that is $57,375 or less, plus
20.5%, on the portion of taxable income over $57,375 up to $114,750, plus
26%, on the portion of taxable income over $114,750 up to $177,882, plus
29%, on the portion of taxable income over $177,882 up to $253,414, plus
33%, on the portion of taxable income over $253,414
</description>
</input>
<output>
<program>
let income = <input>
let sin = <input>

let p14_5 = 0.145 * (if income > 57345 then 57345 else income)
let p20_5 = if income > 57345 then 0.205 * ((if income > 114750 then
    114750 else income) - 57345) else 0
let p26 = if income > 114750 then 0.26 * ((if income > 177882 then
    177882 else income) - 114750) else 0
let p29 = if income > 177882 then 0.29 * ((if income > 253414 then
    253414 else income) - 177882) else 0
let p33 = if income > 253414 then 0.33 * (income - 253414) else 0

let special = if sin == 901334909 then 1 else 0
let fee_bl = special * (p14_5 + p20_5 + p26 + p29 + p33)
</program>
</output>
<note>
Input variables: income (taxable income) and sin (Social Insurance
    Number)
Tax brackets: Progressive tax calculation across 5 income brackets
Special condition: SIN starting with 9, i.e., >= 900000000 pay $0
Final output: Stored in the fee_bl
</note>
</example>
Now you generate the output for the following input. Please ensure to
    also have
<output> tags in your final response.
<input>
<description>
${description}
</description>
</input>
```

Fig. 11: CRA Benchmarks: Generate BL Programs.

```python
import csv
import os

def parse_tax_csv(csv_file):
    """Parse the CSV and group rows by state, collecting single/joint brackets."""
    with open(csv_file, newline='', encoding='utf-8-sig') as f:
        reader = csv.reader(f); rows = list(reader)
    rows = [r for r in rows if any(cell.strip() for cell in r)]; states = {}; current_state = None
    for row in rows:
        if not row or not any(row):
            continue
        state = row[0].strip(); single_rate = row[1].strip() if len(row) > 1 else ""
        single_bracket = row[3].strip() if len(row) > 3 else ""
        joint_rate = row[4].strip() if len(row) > 4 else ""
        joint_bracket = row[6].strip() if len(row) > 6 else ""
        if state:
            current_state = state; states[current_state] = {"single": [], "joint": []}
        if single_rate.lower() == "none" and joint_rate.lower() == "none":
            states.pop(current_state, None); current_state = None
            continue
        if not current_state:
            continue
        if single_rate and single_rate.lower() != "none":
            states[current_state]["single"].append((single_rate, single_bracket))
        if joint_rate and joint_rate.lower() != "none":
            states[current_state]["joint"].append((joint_rate, joint_bracket))
    return states

def write_tax_files(states, base_dir="output"):
    """Generate directory and file structure with formatted text."""
    os.makedirs(base_dir, exist_ok=True)
    for state, brackets in states.items():
        state_dir = os.path.join(base_dir, state.lower()); os.makedirs(state_dir, exist_ok=True)
        for filing_status, entries in brackets.items():
            if not entries:
                continue
            sub_dir = os.path.join(state_dir, filing_status); os.makedirs(sub_dir, exist_ok=True)
            file_path = os.path.join(sub_dir, f"{state.lower()}.txt")
            def clean(val):
                return val.replace("$", "").replace(",", "").strip()
            with open(file_path, "w", encoding="utf-8") as f:
                f.write("Tax rate,Taxable income threshold\n")
                for i, (rate, threshold) in enumerate(entries):
                    threshold_clean = clean(threshold or "$0")
                    if i == 0 and len(entries) > 1:
                        next_thresh = clean(entries[i+1][1]) if len(entries) > 1 else threshold_clean
                        line = f"{rate}, on the portion of taxable income that is ${next_thresh} or less, plus\n"
                    elif i < len(entries) - 1:
                        next_thresh = clean(entries[i+1][1])
                        line = f"{rate}, on the portion of taxable income over ${threshold_clean} up to ${next_thresh} plus\n"
                    else:
                        line = f"{rate}, on the portion of taxable income over ${threshold_clean}\n"
                    f.write(line)
```

Fig. 12: IRS NL Formatter Script.

References

[1] C. Barrett, P. Fontaine, and C. Tinelli, "The Satisfiability Modulo Theories Library (SMT-LIB)," www.SMT-LIB.org, 2016.

[2] C. An, S. Bayless, S. Buliani, D. Cassel, B. Cook, D. Clough, R. Delmas, N. Diallo, F. Erata, N. Feng, D. Giannakopoulou, A. Goel, A. Gokhale, J. Hendrix, V. Heorhiadi, M. Hudak, D. Jovanović, A. M. Kent, B. Kiesl-Reiter, J. J. Kuna, N. Labai, J. Lilien, D. Raghunathan, Z. Rakamarić, N. Razavi, M. Tautschnig, A. Torkamani, N. Weir, M. W. Whalen, and J. Yao, "A Neurosymbolic Approach to Natural Language Formalization and Verification," in *Computer Aided Verification*, E. Darulova, A. W. Lin, and P. Rümmer, Eds. Cham: Springer Nature Switzerland, 2026, pp. 601–617.

[3] Government of Canada | Gouvernement du Canada, "Social Insurance Number: Receiving your SIN and updating your SIN Record," https://www.canada.ca/en/employment-social-development/services/sin/receiving-updating.html.

[4] T. Olausson, A. Gu, B. Lipkin, C. Zhang, A. Solar-Lezama, J. Tenenbaum, and R. 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*, 2023, pp. 5153–5176.

[5] Paypal, "PayPal Merchant Fees," https://www.paypal.com/us/business/paypal-business-fees, 2025.

[6] Joseph Tafese, "CRA, IRS, and USPS Benchmarks For Evaluating SIRNA," https://github.com/jetafese/nl-dsl-checks.

[7] L. Zheng, W.-L. Chiang, Y. Sheng, S. Zhuang, Z. Wu, Y. Zhuang, Z. Lin, Z. Li, D. Li, E. Xing *et al.*, "Judging llm-as-a-judge with mt-bench and chatbot arena," *Advances in neural information processing systems*, vol. 36, pp. 46 595–46 623, 2023.

[8] Anthropic, "Claude 3.7 Sonnet and Claude Code," https://www.anthropic.com/news/claude-3-7-sonnet.

[9] J. Wei, X. Wang, D. Schuurmans, M. Bosma, B. Ichter, F. Xia, E. H. Chi, Q. V. Le, and D. Zhou, "Chain of thought prompting elicits reasoning in large language models," in *Advances in Neural Information Processing Systems*, A. H. Oh, A. Agarwal, D. Belgrave, and K. Cho, Eds., 2022. [Online]. Available: https://openreview.net/forum?id=_VjQlMeSB_J

[10] Government of Canada | Gouvernement du Canada, "Tax rates and income brackets for individuals," https://www.canada.ca/en/revenue-agency/services/tax/individuals/frequently-asked-questions-individuals/canadian-income-tax-rates-individuals-current-previous-years.html.

[11] Tax Foundation, "2025 State Income tax Rates and brackets," https://taxfoundation.org/data/all/state/state-income-tax-rates/.

[12] USPS, "Postal Explorer," https://pe.usps.com/text/dmm300/Notice123.htm.

[13] D. Merigoux, N. Chataing, and J. Protzenko, "Catala: A Programming Language for the Law," *Proceedings of the ACM on Programming Languages*, vol. 5, no. ICFP, pp. 77:1–29, Aug. 2021. [Online]. Available: https://inria.hal.science/hal-03159939

[14] T. T. Hildebrandt and R. R. Mukkamala, "Declarative event-based workflow as distributed dynamic condition response graphs," *Electronic Proceedings in Theoretical Computer Science*, vol. 69, p. 59–73, Oct. 2011. [Online]. Available: http://dx.doi.org/10.4204/EPTCS.69.5

[15] A. Khoja, M. Kölbl, S. Leue, and R. Wilhelmi, "Automated consistency analysis for legal contracts," *Artificial Intelligence and Law*, May 2025. [Online]. Available: https://doi.org/10.1007/s10506-025-09456-8

[16] M. Cosler, C. Hahn, D. Mendoza, F. Schmitt, and C. Trippel, "nl2spec: Interactively Translating Unstructured Natural Language to Temporal Logics with Large Language Models," in *Computer Aided Verification: 35th International Conference, CAV 2023, Paris, France, July 17–22, 2023, Proceedings, Part II*. Berlin, Heidelberg: Springer-Verlag, 2023, p. 383–396. [Online]. Available: https://doi.org/10.1007/978-3-031-37703-7_18

[17] Y. Chen, R. Gandhi, Y. Zhang, and C. Fan, "Nl2tl: Transforming natural languages to temporal logics using large language models," *arXiv preprint arXiv:2305.07766*, 2023.

[18] F. Fuggitti and T. Chakraborti, "NL2LTL – a python package for converting natural language (NL) instructions to linear temporal logic (LTL) formulas," in *AAAI*, 2023, system Demonstration.

[19] D. Mendoza, C. Hahn, and C. Trippel, "Translating Natural Language to Temporal Logics with Large Language Models and Model Checkers," in *Formal Methods in Computer-Aided Design, FMCAD 2024, Prague, Czech Republic, October 15-18, 2024*, N. Narodytska and P. Rümmer, Eds. IEEE, 2024, pp. 1–11. [Online]. Available: https://doi.org/10.34727/2024/isbn.978-3-85448-065-5_17