IPFLang: A Domain-Specific Language for Multi-Currency Regulatory Fee Computation with Static Verification

Valer Bocan, *Senior Member, IEEE,* and Valentina E. Balas, *Senior Member, IEEE*

***Abstract*—Regulatory fee calculations involve complex, jurisdiction-specific rules typically implemented as ad-hoc code or spreadsheets, leading to errors and maintenance difficulties. No formal specification exists for encoding fee computation rules. This paper presents IPFLang, a domain-specific language addressing these software engineering challenges through novel language design and static verification. We develop a formal EBNF grammar with declarative fee computation blocks, a currency-aware type system supporting the 161 active ISO 4217 currencies, static analysis algorithms for completeness and monotonicity verification, and provenance tracking for auditability. We present formal typing rules with a type safety argument and complexity analysis. The open-source reference implementation includes 118 production jurisdiction files covering PCT national/regional phase entry fees validated by a domain expert, a comprehensive test suite with 269 test methods achieving 100% pass rate, and sub-millisecond execution performance. The currency-aware type system prevents cross-currency arithmetic errors at compile time. IPFLang offers a foundation for fee computation with compile-time type safety properties, validated through extensive testing of the reference implementation.**

***Index Terms*—Domain-specific language, type system, static verification, program correctness, currency safety, regulatory automation, formal specification.**

I. Introduction

A. The IP Technology Fragmentation Problem

THE global intellectual property management ecosystem operates through a complex network of national and regional patent offices, each maintaining independent fee calculation systems with proprietary interfaces. Major offices such as the United States Patent and Trademark Office (USPTO), European Patent Office (EPO), Japan Patent Office (JPO), and World Intellectual Property Organization (WIPO) each provide web-based fee calculators [1]–[4], yet these systems exhibit fundamental interoperability deficiencies that impede efficient IP management workflows.

The first and perhaps most pressing issue is the absence of any standard data format across jurisdictions. Each patent office defines fee parameters using custom terminology, requiring manual interpretation and data entry for each calculation. Where the USPTO uses "entity type" classifications, the EPO employs "applicant category" with entirely different discount structures. WIPO PCT applications introduce additional complexity by requiring separate parameters for International Searching Authority selection, compounding the burden for practitioners handling multi-jurisdiction filings.

Equally problematic is the complete absence of programmatic interfaces for fee calculation. Government calculators operate exclusively through web browsers with no API access, preventing any form of automation or integration with IP management workflows. Patent offices have invested significantly in digital transformation for application filing through systems like ePCT, EFS-Web, and Online Filing, yet they have not extended programmatic access to fee calculation services, creating a gap in their digital offerings.

The situation is further complicated by the proprietary nature of calculation logic itself. Fee computation rules remain embedded in server-side code, entirely inaccessible to practitioners who must verify accuracy against frequently updated official fee schedules. When discrepancies arise between calculated and actual fees, practitioners find themselves unable to diagnose whether errors stem from incorrect inputs or flawed calculation logic, as the underlying implementation remains opaque.

B. The Need for Formal Specification

Successful technology domains achieve interoperability through open standards. SQL standardized database queries through ISO/IEC 9075, HTML standardized web content through W3C specifications, and XML Schema standardized data validation under the same organization. These standards enabled ecosystem growth by separating interface specifications from implementations, allowing multiple vendors to provide interoperable solutions. Research on intellectual property disclosure in standards development has examined how standards organizations balance IP rights with interoperability requirements [5], highlighting the complex dynamics that any IP technology specification effort must navigate.

The IP technology domain lacks equivalent formal specifications for fee calculation, resulting in three critical gaps. The first is a *specification gap*: no formal language exists for expressing jurisdiction-specific fee rules. LegalRuleML [6] addresses compliance checking but lacks arithmetic expressiveness for financial calculations. Catala [7] demonstrates sophisticated tax calculations but targets single-jurisdiction applications requiring formal methods expertise unsuitable for legal practitioners. The second gap concerns *formal verifica-*

V. Bocan is with the Department of Computer Science, Faculty of Automation and Computers, Politehnica University of Timişoara, 300223 Timişoara, Romania (e-mail: valer.bocan@upt.ro).

V. E. Balas is with the Faculty of Engineering, Aurel Vlaicu University of Arad, 310130 Arad, Romania (e-mail: valentina.balas@uav.ro).

tion: existing calculators provide no guarantees that fee definitions cover all valid input combinations or behave predictably as inputs change. The third gap is one of *transparency*: proprietary implementations prevent independent verification of calculation correctness.

C. Research Questions and Contributions

This work addresses four fundamental questions regarding formal specification of IP fee calculations.

The first research question concerns language design: can a domain-specific language provide sufficient expressiveness for complex regulatory fee structures while employing syntax designed for readability by IP practitioners? The second addresses formal correctness: what static guarantees can a type system and verification framework provide for multi-currency regulatory calculations? The third focuses on verification: how can completeness and monotonicity properties be statically verified to ensure fee definitions behave correctly? The fourth explores practical feasibility: can a DSL-based approach achieve acceptable performance for production use?

This paper presents IPFLang (Intellectual Property Fees Language), a domain-specific language specification for multi-jurisdiction fee calculations, with five principal contributions:

1) **Language Specification (Section III):** Formal definition of IPFLang syntax using EBNF grammar, with declarative fee computation blocks, explicit input type declarations including a currency-aware AMOUNT type, temporal operators for date-dependent calculations, version management with effective dates, and jurisdiction composition for code reuse.
2) **Type System and Static Verification (Section IV):** A currency-aware type system supporting 161 ISO 4217 currencies with formal typing rules that prevent cross-currency arithmetic errors at compile time, along with verification algorithms for completeness and monotonicity with complexity analysis.
3) **Provenance and Auditability (Section V):** Execution tracing showing how final amounts derive from input parameters, with counterfactual analysis enabling what-if scenarios for regulatory compliance.
4) **Reference Implementation (Section VI):** An open-source command-line tool released under GPLv3 demonstrating IPFLang execution with 269 test methods validating type safety and verification correctness.
5) **Validation (Section VII):** Expert validation of 118 production jurisdiction files covering PCT national/regional phase entry fees for major patent offices worldwide.

D. Paper Organization

Section II surveys related work in IP data standards, legal domain DSLs, and regulatory automation. Section III presents the IPFLang language specification, with the complete EBNF grammar in the project documentation. Section IV details the currency-aware type system with formal typing rules and static verification algorithms. Section V describes provenance tracking and counterfactual analysis. Section VI presents the reference implementation architecture. Section VII presents the evaluation including expert validation of production jurisdiction files and threats to validity. Section VIII discusses advantages of DSL-based formal specification, cross-domain applicability, and limitations. Section IX concludes with contributions summary, impact assessment, and future directions.

II. Related Work and Standards Landscape

A. Existing IP Data Standards

The IP technology domain has achieved partial standardization in data exchange but lacks standards for computational tasks like fee calculation.

WIPO ST.96, commonly known as Patent XML, represents the World Intellectual Property Organization's standard for patent application data exchange [8]. The standard defines XML schemas covering bibliographic data, descriptions, claims, and drawings, but it explicitly excludes financial calculations from its scope. While the standard provides a foundation for data interoperability, fee calculations fall entirely outside its purview.

The European Patent Office provides Open Patent Services (OPS), which offers RESTful APIs for patent search and retrieval [9]. OPS provides family information, legal status, and bibliographic data, but omits fee calculation endpoints entirely. The existence of OPS demonstrates that patent offices can successfully deploy programmatic interfaces, suggesting that fee calculation APIs are technically feasible but simply have not been prioritized.

The gap becomes clear upon examination: IP data standards focus on informational exchange such as bibliographic data, legal status, and full-text search, while omitting computational tasks entirely. Fee calculation remains manual, preventing end-to-end workflow automation.

B. Domain-Specific Languages for Legal Domains

Academic research in computational law has produced several DSLs for legal rules, yet none address regulatory fee calculations with multi-currency and multi-jurisdiction requirements.

LegalRuleML, developed by Athan et al. [6], provides an XML-based specification language for legal rules with ontology-based reasoning. The language excels at representing deontic logic covering obligations, permissions, and prohibitions, and supports defeasibility for handling rule precedence. However, LegalRuleML emphasizes binary compliance checking (compliant or non-compliant) with minimal arithmetic support. Complex fee formulas involving thresholds, progressions, and conditional multipliers exceed the language's design scope. The XML syntax also presents accessibility challenges for legal professionals without technical training.

Catala, created by Merigoux et al. [7], represents a programming language specifically designed for tax law computation. The language demonstrates sophisticated financial calculations with formal verification guarantees through dependent type theory. However, Catala targets single-jurisdiction applications, primarily the French tax code, and requires formal methods expertise that limits adoption by legal practitioners. IPFLang and Catala represent complementary approaches:

Catala employs dependent types for exhaustive case coverage targeting formal verification experts, while IPFLang prioritizes comprehensibility through keyword-based syntax (EQ, GT, AND rather than symbols) and domain-specific primitives (currency literals, temporal operators) designed for IP practitioners to read and modify directly. While Catala's dependent types provide stronger theoretical guarantees, IPFLang's simpler type system (currency-parameterized amounts without dependent types) suffices for IP fee calculations where amounts are always non-negative and conditions are finite Boolean combinations.

Contract-oriented DSLs [10] focus on party obligations, temporal constraints, and conditional execution semantics. Monetary aspects receive minimal treatment, with basic arithmetic operations but lacking multi-currency precision, exchange rate management, and historical rate tracking required for cross-border IP portfolios.

The concept of encoding units in type systems originates with Kennedy's dimensional types [11], which prevent unit mismatch errors in scientific computing. IPFLang applies similar principles to currency, extending the concept with explicit conversion operators and polymorphic type variables for generic fee definitions. Recent work on graded modal types [12] demonstrates how type systems can track quantitative resource usage, providing theoretical foundations for systems that reason about resource consumption, a concept related to IPFLang's tracking of monetary values across fee computations.

OpenFisca [13] provides a Python-based platform for tax-benefit microsimulation, used by governments including France and New Zealand. While powerful, OpenFisca requires Python programming expertise and targets general fiscal policy rather than the specific requirements of IP fee calculation.

Existing legal DSLs address contract execution, compliance checking, or single-jurisdiction calculations, but none provide the combination of arithmetic expressiveness for complex fee formulas, multi-currency support with type safety, static verification of completeness and monotonicity, and multi-jurisdiction portability that IP fee calculation demands.

C. Regulatory Automation and Rules Engines

Automated compliance checking represents a related domain where technology assists regulatory interpretation.

Business Rules Management Systems such as Drools [14] provide general-purpose rules engines using production rules with Rete algorithm inference. These systems require substantial technical expertise, lack domain-specific abstractions for legal concepts, and impose expensive enterprise licensing. While powerful, BRMS platforms are fundamentally overengineered for deterministic fee calculations.

Some governments have begun pursuing rules-as-code initiatives that encode regulations in executable formats [15], [16]. New Zealand's "Better Rules" program and similar initiatives in Australia and France explore machine-consumable legislation. The OECD has documented the international scope of these efforts, providing frameworks for encoding rules in machine-readable formats that emphasize transparency and regulatory automation [17]. More recent policy analysis explores how rules-as-code approaches can enable more efficient global economic governance, including applications to international trade regulations and intellectual property [18]. These initiatives typically use general-purpose languages rather than domain-specific languages, limiting accessibility to legal experts who must rely on programmers for implementation.

Table I gives a systematic comparison of IPFLang with related approaches.

IPFLang differentiates itself through the combination of domain-specific syntax designed for readability, first-class multi-currency type safety with all 161 ISO 4217 currencies as built-in primitives, static verification of completeness and monotonicity, and explicit support for multi-jurisdiction fee structures with inheritance and composition. While languages like Catala could theoretically support currency types through user-defined abstractions, IPFLang supplies these as language primitives requiring no additional implementation effort. The design follows established principles for DSL development that emphasize matching language constructs to domain concepts [19]. Section VII-D substantiates these distinctions empirically by re-implementing representative schedules in Catala and OpenFisca.

III. IPFLang Language Specification

A. Design Principles

IPFLang design balances five competing objectives: expressiveness for complex regulatory logic, readability through domain-specific syntax designed for legal professionals, deterministic execution for financial accuracy, extensibility for evolving regulations, and formal verification potential.

Declarative semantics form the foundation of IPFLang's design. Fee calculations specify what to compute rather than how to compute it, enabling legal experts to validate fee formulas directly against official schedules without needing to understand imperative control flow.

Readability drove the choice of explicit keyword semantics. Operators use keyword syntax such as EQ, GT, AND, and OR rather than symbols like `==`, `>`, `&&`, and `||`. This approach aims to improve accessibility for users with varying technical backgrounds, aligning with DSL design guidelines that emphasize domain-specific notation [20]. Empirical validation of this readability hypothesis remains future work.

A *static type system* catches errors before execution. Input declarations explicitly specify types (NUMBER, LIST, MULTILIST, BOOLEAN, DATE, and AMOUNT), enabling compile-time validation that prevents runtime errors from type mismatches while providing clear parameter documentation.

IPFLang deliberately embraces *minimal syntax complexity*, targeting only features necessary for regulatory fee calculations. Structured blocks such as `DEFINE...ENDDEFINE` and `COMPUTE...ENDCOMPUTE` with explicit terminators aid comprehension and prevent syntax errors common in expression-based languages.

Regulatory contexts demand *auditability and traceability*. Fee computation produces step-by-step execution traces showing how final amounts derive from input parameters,

TABLE I
COMPARISON OF LEGAL DSLS AND RULES ENGINES

Feature	IPFLang	Catala	LegalRuleML	Drools	OpenFisca
Primary Domain	IP fees	Tax law	Compliance	General	Tax-benefit
Type Safety	Currency-aware	Dependent types	None	None	Runtime
Multi-currency	Built-in	User-definable	No	No	Limited
Static Verification	Completeness, Monotonicity	Exhaustiveness	Defeasibility	None	None
Syntax Style	Declarative keywords	Literate	XML	Drools DRL	Python
Target Users	Legal professionals	Formal methods experts	Ontology engineers	Developers	Developers
Multi-jurisdiction	Composition	No	No	No	Yes
Interoperability Focus	Multi-jurisdiction	Single jurisdiction	Ontology mapping	Integration APIs	Country modules
Provenance	Built-in	No	No	Audit log	No
License	GPLv3	Apache 2.0	OASIS	Apache 2.0	AGPL

addressing legal requirements for calculation transparency and assisting in dispute resolution.

Finally, *jurisdiction independence* ensures portability across patent offices. Language constructs make no assumptions about specific jurisdictions; all jurisdiction-specific business rules reside in fee definitions rather than language syntax, enabling extensive code reuse.

B. Language Syntax Overview

An IPFLang program begins with an optional version declaration that establishes metadata including version identifier, effective date, and references to authoritative fee schedules. Following the version declaration, group definitions organize inputs into logical categories for user interface presentation, with weights determining display order. The core of any IPFLang program consists of input definitions that declare the parameters required for fee calculation; these may be single-choice enumerations (LIST), multi-choice selections (MULTILIST), numeric values with optional constraints (NUMBER), binary choices (BOOLEAN), temporal values (DATE), or currency-aware monetary inputs (AMOUNT). Fee computation blocks contain the actual calculation logic using conditional yields and case statements. Programs may include verification directives that enable static analysis of completeness and monotonicity properties. Finally, return statements provide named outputs for integration with external systems.

C. Version Declaration

Fee schedules change frequently, requiring version tracking with effective dates. The syntax is:

```
VERSION '<VersionId>' EFFECTIVE <date>
  [DESCRIPTION '<description>'] [REFERENCE
    '<reference>']
```

For example:

```
VERSION '2024.1' EFFECTIVE 01.04.2024
  DESCRIPTION 'EPO official fees 2024'
  REFERENCE 'EPO Official Journal 2024/03'
```

The VERSION directive enables temporal queries (calculating fees as they were on a specific date), version comparison (identifying changes between fee schedule revisions), and regulatory traceability (linking calculations to authoritative sources).

D. Input Type System

IPFLang defines six input types matching common parameter patterns across patent offices.

1) LIST (Single-Choice Enumeration): The LIST type handles parameters with mutually exclusive options such as entity type, filing basis, or examination request.

```
DEFINE LIST EntityType AS 'Select entity size'
GROUP General
CHOICE NormalEntity AS 'Large Entity'
CHOICE SmallEntity AS 'Small Entity (60% discount)'
CHOICE MicroEntity AS 'Micro Entity (80% discount)'
DEFAULT NormalEntity
ENDDEFINE
```

2) MULTILIST (Multi-Choice Enumeration): The MULTILIST type accommodates parameters allowing multiple selections. The special property accessor `!COUNT` returns the number of selections. IPFLang uses the exclamation mark (`!`) as the property accessor to distinguish property access from other syntactic elements and to provide visual emphasis for these computed properties, which derive values from the underlying data rather than representing stored values directly.

```
DEFINE MULTILIST Countries AS 'Select validation
    countries'
CHOICE VAL_DE AS 'Germany'
CHOICE VAL_FR AS 'France'
CHOICE VAL_GB AS 'United Kingdom'
DEFAULT VAL_DE,VAL_FR
ENDDEFINE
# Usage: YIELD 100 * Countries!COUNT
```

3) NUMBER (Numeric Input): The NUMBER type handles counts, quantities, and page numbers with optional constraints.

```
DEFINE NUMBER ClaimCount AS 'Number of claims in
    application'
BETWEEN 1 AND 500
DEFAULT 10
ENDDEFINE
```

4) BOOLEAN (Yes/No): The BOOLEAN type represents binary choices.

```
DEFINE BOOLEAN RequestExamination AS 'Request
    substantive examination?'
DEFAULT TRUE
ENDDEFINE
```

5) DATE (Date Input): The DATE type handles filing dates and priority dates for calculating time-dependent fees. Temporal properties enable calculations based on elapsed time: `!YEARSTONOW` (years from date to current date), `!MONTHSTONOW` (months), `!DAYSTONOW` (days), and `!MONTHSTONOW_FROMLASTDAY` (months from the end of the date's month).

```
DEFINE DATE FilingDate AS 'Application filing date'
BETWEEN 01.01.2000 AND TODAY
DEFAULT TODAY
ENDDEFINE
# Usage: LET YearsSinceFiling AS
    FilingDate!YEARSTONOW
```

6) AMOUNT (Currency-Aware Monetary Input): The AMOUNT type represents monetary values with an associated ISO 4217 currency code, enabling type-safe arithmetic.

```
DEFINE AMOUNT PriorSearchFee AS 'Prior art search
    fee paid'
CURRENCY EUR
DEFAULT 0
ENDDEFINE
```

The AMOUNT type integrates with the currency-aware type system described in Section IV, preventing accidental cross-currency arithmetic.

E. Group Definitions

Groups organize inputs for user interface presentation, with weights determining display order.

```
DEFINE GROUP General AS 'General Information' WITH
    WEIGHT 10
DEFINE GROUP Claims AS 'Claims Information' WITH
    WEIGHT 20
DEFINE GROUP Options AS 'Fee Options' WITH WEIGHT 30
```

F. Fee Computation Blocks

Fee calculations use structured COMPUTE FEE blocks with conditional YIELD statements. The basic syntax is:

```
COMPUTE FEE <fee_name> [OPTIONAL]
  [LET <variable> AS <expression>]*
  [CASE <condition> AS
     YIELD <expression> [IF <condition>]
   ENDCASE]*
  YIELD <expression> [IF <condition>]
ENDCOMPUTE
```

The OPTIONAL keyword distinguishes fees that may or may not apply from mandatory fees. The following example encodes EPO claim fees with progressive rates:

```
COMPUTE FEE ExcessClaimsFee
LET ClaimFee1 AS 265<EUR>
LET ClaimFee2 AS 660<EUR>
CASE ClaimCount LTE 15 AS
  YIELD 0<EUR>
ENDCASE
CASE ClaimCount GT 15 AND ClaimCount LTE 50 AS
  YIELD ClaimFee1 * (ClaimCount - 15)
ENDCASE
CASE ClaimCount GT 50 AS
  YIELD ClaimFee1 * 35 + ClaimFee2 * (ClaimCount -
    50)
ENDCASE
ENDCOMPUTE
```

Listing 1. EPO excess-claim fee with progressive rates.

This directly encodes the EPO's fee schedule: EUR 265 per claim for claims 16–50 and EUR 660 per claim beyond 50.

G. Currency Literals

Numeric values can be annotated with ISO 4217 currency codes:

```
100<EUR>      # 100 Euros
50.50<USD>    # 50.50 US Dollars
1000<JPY>     # 1000 Japanese Yen
```

The type system enforces currency compatibility at compile time, as detailed in Section IV.

H. Operators and Expressions

IPFLang supports comparison operators EQ (equality), NEQ (inequality), GT (greater than), LT (less than), GTE (greater or equal), and LTE (less or equal); logical operators AND and OR (with short-circuit evaluation); arithmetic operators +, −, *, / with ROUND, FLOOR, and CEIL functions; and set operators for MULTILIST: IN (membership), NIN (non-membership), and `!COUNT` (cardinality). Operator precedence, from highest to lowest, is: ROUND/FLOOR/CEIL; then *, /; then +, −; then comparisons; then AND; then OR. Parentheses override default precedence.

I. Jurisdiction Composition

IPFLang supports inheritance for code reuse across related fee schedules. A child jurisdiction inherits inputs and fees from a parent, can add new definitions, and can override inherited fees.

```
# Parent (EPO base):
COMPUTE FEE FilingFee
YIELD 135<EUR>
ENDCOMPUTE

# Child (EPO Germany national phase):
# Inherits FilingFee from parent; adds
    Germany-specific fee
COMPUTE FEE GermanTranslationFee
YIELD 1050<EUR> IF NeedsTranslation EQ TRUE
YIELD 0<EUR> IF NeedsTranslation EQ FALSE
ENDCOMPUTE
```

Listing 2. Jurisdiction composition via inheritance.

Composition enables significant code reuse between related jurisdictions, simplifies maintenance when base fees change, and maintains clear traceability of fee origins. Jurisdiction composition is implemented at the tool level rather than as a language construct. The `ipflang compose` command merges multiple IPFLang files, with the resulting combined program conforming to the grammar specification and preserving the type safety properties established by the constituent files, analogous to verification-preserving transformations in program verifiers [21].

IV. Type System and Static Verification

A. Currency-Aware Type System

IPFLang employs a dimensional type system preventing cross-currency arithmetic errors at compile time, analogous to units-of-measure checking in scientific computing [11]. The system supports the 161 active ISO 4217 currency codes (excluding funds and metals), and the approach to currency as a type parameter draws on similar patterns in language standards for monetary computation, such as JSR-354 (Java Money and Currency API) [22], which defines standard interfaces for representing and manipulating monetary amounts in type-safe ways.

The type system gives static guarantees through compile-time checking: well-typed programs cannot produce currency mismatch errors during execution. We present formal typing rules following standard notation [23]. While these rules constitute a rigorous specification that has been implemented and validated through extensive testing (269 test cases), a fully mechanized proof of type soundness (progress and preservation properties) over a defined operational semantics is identified as future work. The guarantees provided are thus implementation-validated rather than theorem-prover certified.

1) Type Language: The type language extends basic types with currency-parameterized amounts:

τ	::=	Num	dimensionless numbers
	|	Bool	Boolean values
	|	Sym	symbolic identifiers (LIST)
	|	Date	date values
	|	$\mathsf{SymList}$	symbol lists (MULTILIST)
	|	$\mathsf{Amt}[c],\ c \in \text{ISO-4217}$	monetary amounts
	|	α	type variables
σ	::=	$\tau \mid \forall \alpha.\, \sigma$	type schemes

where Num represents dimensionless numbers used for counts and arithmetic; Bool Boolean values; Sym symbolic identifiers from LIST input choices; Date date values with temporal properties; $\mathsf{SymList}$ symbol lists from MULTILIST selections; $\mathsf{Amt}[c]$ monetary amounts denominated in currency $c \in$ ISO-4217; α currency type variables for polymorphic fee definitions; and $\forall \alpha.\, \sigma$ universally quantified type schemes. Currency type variables range over ISO-4217 currency codes. In the typing context Γ, the notation $\alpha : \mathsf{Currency}$ indicates that α is bound as a currency type variable.

2) Typing Rules: Let Γ denote a typing environment mapping identifiers to types, written $\Gamma(x) = \tau$. We define the typing judgment $\Gamma \vdash e : \tau$, meaning that under environment Γ, expression e has type τ.

a) Literals and Variables: The foundation of the type system rests on three base rules that assign types to the simplest expressions in the language. These rules handle constants (both numeric and currency-annotated) and variable references, forming the leaves of every expression tree that the type checker analyzes.

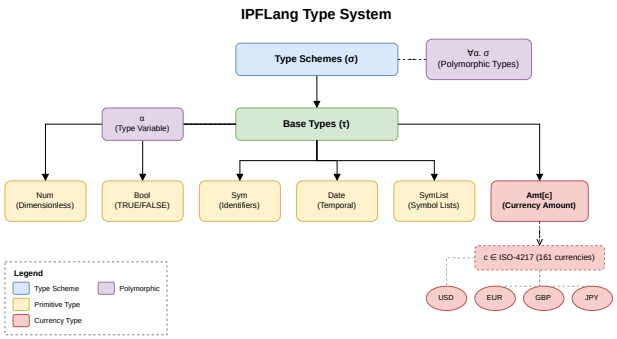

Fig. 1. IPFLang type hierarchy showing the relationship between base types (Num, Bool, Sym, Date, SymList), currency-parameterized amounts (Amt[c]), and polymorphic type schemes.

$$\frac{}{\Gamma \vdash n : \mathsf{Num}}\ \textsc{T-Num} \qquad \frac{\Gamma(x) = \tau}{\Gamma \vdash x : \tau}\ \textsc{T-Var}$$

$$\frac{c \in \text{ISO-4217}}{\Gamma \vdash n\langle c \rangle : \mathsf{Amt}[c]}\ \textsc{T-Curr}$$

b) Arithmetic Operations: Arithmetic operations form the computational core of fee calculations, and the type system enforces strict currency discipline to prevent nonsensical cross-currency computations. Rule T-Add-Amt embodies the core currency safety guarantee: it permits adding two monetary amounts only when both operands share the identical currency tag c. The expression `100<EUR> + 50<EUR>` is well-typed, but `100<EUR> + 50<USD>` fails type-checking. A deliberate omission is any rule permitting $\mathsf{Amt}[c]+\mathsf{Num}$; `100<EUR> + 50` is rejected because the dimensionless operand's currency is ambiguous.

$$\frac{\Gamma \vdash e_1 : \mathsf{Num} \qquad \Gamma \vdash e_2 : \mathsf{Num}}{\Gamma \vdash e_1 + e_2 : \mathsf{Num}}\ \textsc{T-Add-Num}$$

$$\frac{\Gamma \vdash e_1 : \mathsf{Amt}[c] \qquad \Gamma \vdash e_2 : \mathsf{Amt}[c]}{\Gamma \vdash e_1 + e_2 : \mathsf{Amt}[c]}\ \textsc{T-Add-Amt}$$

Multiplication exhibits different behavior from addition because scalar multiplication is dimensionally sound: multiplying a currency amount by a dimensionless count produces another currency amount.

$$\frac{\Gamma \vdash e_1 : \mathsf{Num} \qquad \Gamma \vdash e_2 : \mathsf{Num}}{\Gamma \vdash e_1 * e_2 : \mathsf{Num}}\ \textsc{T-Mul-Num}$$

$$\frac{\Gamma \vdash e_1 : \mathsf{Amt}[c] \qquad \Gamma \vdash e_2 : \mathsf{Num}}{\Gamma \vdash e_1 * e_2 : \mathsf{Amt}[c]}\ \textsc{T-Mul-Scalar-R}$$

$$\frac{\Gamma \vdash e_1 : \mathsf{Num} \qquad \Gamma \vdash e_2 : \mathsf{Amt}[c]}{\Gamma \vdash e_1 * e_2 : \mathsf{Amt}[c]}\ \textsc{T-Mul-Scalar-L}$$

The subtraction rules mirror addition exactly: T-Sub-Num allows subtracting dimensionless numbers, while T-Sub-Amt permits subtracting monetary amounts only when currencies match. The expression `500<EUR> - 100<EUR>` yields

$\mathsf{Amt}[\mathrm{EUR}]$, enabling calculations such as computing a net fee after deducting a prior payment.

$$\frac{\Gamma \vdash e_1 : \mathsf{Num} \qquad \Gamma \vdash e_2 : \mathsf{Num}}{\Gamma \vdash e_1 - e_2 : \mathsf{Num}} \text{ T-SUB-NUM}$$

$$\frac{\Gamma \vdash e_1 : \mathsf{Amt}[c] \qquad \Gamma \vdash e_2 : \mathsf{Amt}[c]}{\Gamma \vdash e_1 - e_2 : \mathsf{Amt}[c]} \text{ T-SUB-AMT}$$

Division follows principles similar to multiplication. Rule T-DIV-NUM permits dividing one dimensionless number by another; T-DIV-AMT-SCALAR divides a monetary amount by a dimensionless number, preserving the currency tag.

$$\frac{\Gamma \vdash e_1 : \mathsf{Num} \qquad \Gamma \vdash e_2 : \mathsf{Num}}{\Gamma \vdash e_1 / e_2 : \mathsf{Num}} \text{ T-DIV-NUM}$$

$$\frac{\Gamma \vdash e_1 : \mathsf{Amt}[c] \qquad \Gamma \vdash e_2 : \mathsf{Num}}{\Gamma \vdash e_1 / e_2 : \mathsf{Amt}[c]} \text{ T-DIV-AMT-SCALAR}$$

c) Currency Conversion: The CONVERT function is the sole mechanism for changing currency tags, making all cross-currency operations explicit and auditable in the source code. Rule T-CONVERT requires four premises: the expression e must have type $\mathsf{Amt}[c]$ for some currency c; the declared source currency c_1 must exactly equal the inferred currency c; and both c_1 and c_2 must be valid ISO-4217 codes. The premise $c_1 = c$ is critical for soundness: it prevents ill-formed conversions such as `CONVERT(100<EUR>, USD, GBP)` where a programmer mistakenly declares USD as the source currency while the expression is actually denominated in EUR.

$$\frac{\Gamma \vdash e : \mathsf{Amt}[c] \qquad c_1 = c \qquad c_1 \in \text{ISO-4217} \qquad c_2 \in \text{ISO-4217}}{\Gamma \vdash \texttt{CONVERT}(e, c_1, c_2) : \mathsf{Amt}[c_2]} \text{ T-CONVERT}$$

d) Comparisons and Conditions: Rule T-COMP-EQ permits equality (EQ) and inequality (NEQ) comparisons between two values of the same type, where that type must support equality testing. Rule T-COMP-ORD handles the ordering operators over numbers, dates, and currency amounts.

$$\frac{\Gamma \vdash e_1 : \tau \qquad \Gamma \vdash e_2 : \tau \qquad \tau \in \{\mathsf{Num}, \mathsf{Bool}, \mathsf{Sym}, \mathsf{Date}\} \cup \{\mathsf{Amt}[c] : c \in \text{ISO-4217}\} \qquad \oplus \in \{\mathrm{EQ}, \mathrm{NEQ}\}}{\Gamma \vdash e_1 \oplus e_2 : \mathsf{Bool}} \text{ T-COMP-EQ}$$

$$\frac{\Gamma \vdash e_1 : \tau \qquad \Gamma \vdash e_2 : \tau \qquad \tau \in \{\mathsf{Num}, \mathsf{Date}\} \cup \{\mathsf{Amt}[c] : c \in \text{ISO-4217}\} \qquad \oplus \in \{\mathrm{GT}, \mathrm{LT}, \mathrm{GTE}, \mathrm{LTE}\}}{\Gamma \vdash e_1 \oplus e_2 : \mathsf{Bool}} \text{ T-COMP-ORD}$$

Rules T-AND and T-OR type the logical connectives over Boolean operands.

$$\frac{\Gamma \vdash \varphi_1 : \mathsf{Bool} \qquad \Gamma \vdash \varphi_2 : \mathsf{Bool}}{\Gamma \vdash \varphi_1 \texttt{ AND } \varphi_2 : \mathsf{Bool}} \text{ T-AND}$$

$$\frac{\Gamma \vdash \varphi_1 : \mathsf{Bool} \qquad \Gamma \vdash \varphi_2 : \mathsf{Bool}}{\Gamma \vdash \varphi_1 \texttt{ OR } \varphi_2 : \mathsf{Bool}} \text{ T-OR}$$

e) Rounding Functions: Rules T-ROUND, T-FLOOR, and T-CEIL accept either Num or any $\mathsf{Amt}[c]$ and preserve the type, ensuring currency information is never stripped.

$$\frac{\Gamma \vdash e : \tau \qquad \tau \in \{\mathsf{Num}\} \cup \{\mathsf{Amt}[c] : c \in \text{ISO-4217}\}}{\Gamma \vdash \texttt{ROUND}(e) : \tau} \text{ T-ROUND}$$

$$\frac{\Gamma \vdash e : \tau \qquad \tau \in \{\mathsf{Num}\} \cup \{\mathsf{Amt}[c] : c \in \text{ISO-4217}\}}{\Gamma \vdash \texttt{FLOOR}(e) : \tau} \text{ T-FLOOR}$$

$$\frac{\Gamma \vdash e : \tau \qquad \tau \in \{\mathsf{Num}\} \cup \{\mathsf{Amt}[c] : c \in \text{ISO-4217}\}}{\Gamma \vdash \texttt{CEIL}(e) : \tau} \text{ T-CEIL}$$

f) Property Accessors: T-COUNT returns the cardinality of a MULTILIST as Num. The temporal accessors compute elapsed time from a Date variable to the current evaluation date, all yielding Num.

$$\frac{\Gamma(x) = \mathsf{SymList}}{\Gamma \vdash x\texttt{!COUNT} : \mathsf{Num}} \text{ T-COUNT}$$

$$\frac{\Gamma(x) = \mathsf{Date}}{\Gamma \vdash x\texttt{!YEARSTONOW} : \mathsf{Num}} \text{ T-YEARSTONOW}$$

$$\frac{\Gamma(x) = \mathsf{Date}}{\Gamma \vdash x\texttt{!MONTHSTONOW} : \mathsf{Num}} \text{ T-MONTHSTONOW}$$

$$\frac{\Gamma(x) = \mathsf{Date}}{\Gamma \vdash x\texttt{!DAYSTONOW} : \mathsf{Num}} \text{ T-DAYSTONOW}$$

$$\frac{\Gamma(x) = \mathsf{Date}}{\Gamma \vdash x\texttt{!MONTHSTONOW\_FROMLASTDAY} : \mathsf{Num}} \text{ T-MTNFROMLASTDAY}$$

g) Set Membership: T-IN and T-NIN test whether a Sym is among the elements of a SymList, returning Bool.

$$\frac{\Gamma(x) = \mathsf{Sym} \qquad \Gamma(y) = \mathsf{SymList}}{\Gamma \vdash x \texttt{ IN } y : \mathsf{Bool}} \text{ T-IN}$$

$$\frac{\Gamma(x) = \mathsf{Sym} \qquad \Gamma(y) = \mathsf{SymList}}{\Gamma \vdash x \texttt{ NIN } y : \mathsf{Bool}} \text{ T-NIN}$$

h) Let Bindings: T-LET introduces local bindings for named intermediate calculations within fee blocks.

$$\frac{\Gamma \vdash e_1 : \tau_1 \qquad \Gamma[x \mapsto \tau_1] \vdash e_2 : \tau_2}{\Gamma \vdash \texttt{LET } x \texttt{ AS } e_1; e_2 : \tau_2} \text{ T-LET}$$

i) Fee Body Typing: We introduce an auxiliary judgment $\Gamma \vdash \mathit{body}$ **yields** τ, meaning "under environment Γ, all

TABLE II
TYPE SYSTEM VALIDITY EXAMPLES

Expression	Valid	Reason
`100<EUR> + 50<EUR>`	YES	Same currency
`100<EUR> * 2`	YES	Scalar multiplication
`2 * 100<EUR>`	YES	Scalar multiplication (commutative)
`100<EUR> / 2`	YES	Scalar division
`100<EUR> + 50<USD>`	NO	Mixed currencies
`100<EUR> + 50`	NO	Cannot add dimensionless to dimensioned
`100<EUR> - 50`	NO	Cannot subtract dimensionless from dimensioned
`CONVERT(x, USD, EUR) + 50<EUR>`	YES	Explicit conversion

YIELD expressions in the fee body have type τ."

$$\frac{\Gamma \vdash e : \tau \qquad \Gamma \vdash \varphi : \mathsf{Bool}}{\Gamma \vdash (\texttt{YIELD}\ e\ \texttt{IF}\ \varphi)\ \mathbf{yields}\ \tau}\ \text{T-YIELD-STMT}$$

$$\frac{\Gamma \vdash e : \tau}{\Gamma \vdash (\texttt{YIELD}\ e)\ \mathbf{yields}\ \tau}\ \text{T-YIELD-UNCOND}$$

$$\frac{\Gamma \vdash \mathit{stmt}\ \mathbf{yields}\ \tau \qquad \Gamma \vdash \mathit{rest}\ \mathbf{yields}\ \tau}{\Gamma \vdash (\mathit{stmt};\ \mathit{rest})\ \mathbf{yields}\ \tau}\ \text{T-YIELD-SEQ}$$

$$\frac{\Gamma \vdash e_1 : \tau_1 \qquad \Gamma[x \mapsto \tau_1] \vdash \mathit{rest}\ \mathbf{yields}\ \tau_2}{\Gamma \vdash (\texttt{LET}\ x\ \texttt{AS}\ e_1;\ \mathit{rest})\ \mathbf{yields}\ \tau_2}\ \text{T-LET-IN-BODY}$$

j) Fee Rules: T-FEE types a fee block without an explicit currency declaration; T-FEE-RETURN handles fees with an explicit RETURN currency.

$$\frac{\Gamma \vdash \mathit{body}\ \mathbf{yields}\ \tau \qquad \tau \in \{\mathsf{Num}\} \cup \{\mathsf{Amt}[c] : c \in \text{ISO-4217}\}}{\Gamma \vdash (\texttt{COMPUTE FEE}\ f\ \mathit{body}\ \texttt{ENDCOMPUTE}) : \tau}\ \text{T-FEE}$$

$$\frac{\Gamma \vdash \mathit{body}\ \mathbf{yields}\ \mathsf{Amt}[c] \qquad c \in \text{ISO-4217}}{\Gamma \vdash (\texttt{COMPUTE FEE}\ f\ \texttt{RETURN}\ c\ \mathit{body}\ \texttt{ENDCOMPUTE}) : \mathsf{Amt}[c]}\ \text{T-FEE-RETURN}$$

k) Polymorphism: T-POLY-FEE types generic fee definitions that abstract over a currency type variable α; T-INST eliminates polymorphism by instantiating a generic fee at a concrete currency.

$$\frac{\alpha \notin \mathrm{dom}(\Gamma) \qquad \Gamma, \alpha : \mathsf{Currency} \vdash \mathit{body}\ \mathbf{yields}\ \mathsf{Amt}[\alpha]}{\Gamma \vdash (\texttt{COMPUTE FEE}\ f\texttt{<}\alpha\texttt{>}\ \texttt{RETURN}\ \alpha\ \mathit{body}\ \texttt{ENDCOMPUTE}) : \forall\alpha.\, \mathsf{Amt}[\alpha]}\ \text{T-POLY-FEE}$$

$$\frac{\Gamma \vdash f : \forall\alpha.\, \mathsf{Amt}[\alpha] \qquad c \in \text{ISO-4217}}{\Gamma \vdash f@c : \mathsf{Amt}[c]}\ \text{T-INST}$$

3) Type System Summary: Dimensionless numbers (Num) cannot be added to or subtracted from currency amounts (Amt[c]). This design decision prevents ambiguous expressions like `100<EUR> + 50` where the intended currency of 50 is unclear. This approach follows established principles from dimensional type systems [11].

4) Type Safety: The type system is designed to enforce the following currency-safety property. We state it as an intended invariant of well-typed programs, validated by the implementation and its test suite rather than mechanically proved, and we describe the guarantee throughout the paper accordingly.

Property 1 (Currency Safety; implementation-validated)**.** *If program P type-checks under environment Γ (written $\Gamma \vdash P : ok$), then during evaluation: (1) every arithmetic expression evaluates without currency mismatch errors; (2) every fee f computes to a value of its declared type; and (3) currency conversions occur only where CONVERT is explicitly used.*

Justification. Currency consistency is enforced node-by-node: T-ADD-AMT and T-SUB-AMT require matching currency tags, and T-CONVERT is the unique rule that changes a tag; the type checker implements these rules via depth-first AST traversal. This argument is corroborated empirically by 269 test cases but does not constitute a formal proof. Establishing the property rigorously, through progress and preservation in the style of Wright and Felleisen [23] over a defined operational semantics, is left to future work; the guarantees reported here are therefore implementation-validated rather than theorem-prover certified.

B. Completeness Analysis

IPFLang supports static analysis to determine whether fee computations produce defined outputs for all valid input combinations.

1) Formal Definitions:

Definition 1 (Input Domain)**.** Each input declaration defines a semantic domain: `NUMBER BETWEEN m AND M` defines $\mathrm{Dom} = \{n \in \mathbb{Z} : m \le n \le M\}$; `BOOLEAN` defines $\mathrm{Dom} = \{\mathrm{TRUE}, \mathrm{FALSE}\}$; `LIST` with choices $\{c_1, \ldots, c_k\}$ defines $\mathrm{Dom} = \{c_1, \ldots, c_k\}$; and `MULTILIST` with choices $\{c_1, \ldots, c_k\}$ defines $\mathrm{Dom} = \mathcal{P}(\{c_1, \ldots, c_k\})$.

Definition 2 (Valuation Space)**.** The valuation space Σ is the Cartesian product of all input domains: $\Sigma = \mathrm{Dom}_1 \times \mathrm{Dom}_2 \times \cdots \times \mathrm{Dom}_m$.

Definition 3 (Coverage)**.** For fee f with conditional yields guarded by conditions $\varphi_1, \ldots, \varphi_n$, the coverage is $\mathrm{Cov}(f) = \{\sigma \in \Sigma : \exists i.\ \sigma \models \varphi_i\}$.

Definition 4 (Completeness)**.** Fee f is complete if $\mathrm{Cov}(f) = \Sigma$, equivalently if $\varphi_1 \vee \cdots \vee \varphi_n$ is valid over Σ.

2) Analysis Algorithm:

3) Complexity Analysis:

Theorem 1 (Verification Complexity)**.** *Let $n = |\Phi|$ (number of yield conditions), $m = |D|$ (number of inputs), k the average condition evaluation cost, and $N = |\Sigma|$ (domain size). Exhaustive verification ($N \le 10^6$) runs in $O(N \cdot n \cdot k)$ time and $O(|gaps|)$ space and provides a formal guarantee. Boundary-based testing ($N > 10^6$) runs in $O(S \cdot n \cdot k)$ time, where $S = O(5^m)$ for boundary sampling, and provides heuristic assurance only.*

Algorithm 1 Completeness Analysis

Require: Fee f with conditions $\Phi = \{\varphi_1, \ldots, \varphi_n\}$; input domains $D = \{D_1, \ldots, D_m\}$
Ensure: $(complete : \text{Bool},\ gaps)$

$\Sigma \leftarrow D_1 \times D_2 \times \cdots \times D_m$ ▷ valuation space
$|\Sigma| \leftarrow \prod_i |D_i|$ ▷ domain size
$gaps \leftarrow \emptyset$
if $|\Sigma| \leq$ EXHAUSTIVETHRESHOLD **then** ▷ default: 10^6
  for each $\sigma \in \Sigma$ **do** ▷ exhaustive: formal guarantee
    if $\neg\exists\varphi_i \in \Phi : \sigma \models \varphi_i$ **then**
      $gaps \leftarrow gaps \cup \{\sigma\}$
      if $|gaps| \geq$ MAXGAPS **then**
        break
      end if
    end if
  end for
else
  $S \leftarrow$ SAMPLEREPRESENTATIVE(D) ▷ boundary: heuristic
  for each $\sigma \in S$ **do**
    if $\neg\exists\varphi_i \in \Phi : \sigma \models \varphi_i$ **then**
      $gaps \leftarrow gaps \cup \{\sigma\}$
    end if
  end for
end if
return $(gaps = \emptyset,\ gaps)$

For MULTILIST inputs, domain size grows exponentially: k choices yield 2^k possible selections.

4) Sampling Strategy: For numeric domains $[min, max]$, representative sampling selects boundary values (min and max), near-boundary values ($min{+}1$ and $max{-}1$), the midpoint, and threshold values extracted from conditions. For MULTILIST domains, sampling includes the empty set, all singleton sets, and the full set.

Proposition 1 (Soundness of Exhaustive Mode)**.** *If the exhaustive algorithm reports complete, the fee is genuinely complete.*

As an example of gap detection, the fee

```
COMPUTE FEE ClaimFee
  YIELD 100<EUR> * (ClaimCount - 20)
    IF EntityType EQ Large AND ClaimCount GT 20
  YIELD 50<EUR> * (ClaimCount - 20)
    IF EntityType EQ Small AND ClaimCount GT 20
ENDCOMPUTE
VERIFY COMPLETE FEE ClaimFee
```

Listing 3. Static completeness gap detection.

produces the output `Gap: {EntityType=Micro}` and `Gap: {ClaimCount <= 20}`.

C. Monotonicity Verification

Fee schedules should exhibit predictable behavior: increasing claim count should never decrease the fee.

1) Formal Definition:

Algorithm 2 Monotonicity Verification

Require: Fee f; numeric input x with domain $D_x = [min, max]$; other domains D_{-x}; expected direction d
Ensure: $(monotonic : \text{Bool},\ violations)$

$violations \leftarrow [\,]$
for each $\sigma_{-x} \in$ SAMPLEREPRESENTATIVE(D_{-x}) **do** ▷ context sampling
  $V \leftarrow$ SAMPLEMONOTONIC$(D_x, 20)$ ▷ 20 ordered values
  $prev\_fee \leftarrow \bot$
  for each $v \in \text{sort}(V)$ **do**
    $curr\_fee \leftarrow$ EVALUATE$(f, \sigma_{-x}[x \mapsto v])$
    if $prev\_fee \neq \bot$ **and** VIOLATES$(prev_fee, curr_fee, d)$ **then**
      append $(\sigma_{-x}, prev\_v, prev\_fee, v, curr\_fee)$ to $violations$
    end if
    $prev\_fee \leftarrow curr\_fee$; $prev\_v \leftarrow v$
  end for
end for
return $(violations = [\,],\ violations)$

Definition 5 (Monotonicity)**.** Let $f : \Sigma \to \mathbb{R}$ be a fee function, $x \in Vars$ a numeric input, and σ_{-x} a partial valuation excluding x. Fee f is non-decreasing with respect to x if

$$\forall\sigma_{-x}.\ \forall v_1, v_2 \in \text{Dom}(x).\ v_1 < v_2 \implies f(\sigma_{-x}[x \mapsto v_1]) \leq f(\sigma_{-x}[x \mapsto v_2]).$$

The non-increasing ($\geq$), strictly increasing ($>$), and strictly decreasing ($<$) variants are defined analogously.

2) Verification Algorithm: Here VIOLATES(p, c, d) returns true iff: $d = NonDecreasing$ and $c < p$; $d = NonIncreasing$ and $c > p$; $d = StrictlyIncreasing$ and $c \leq p$; or $d = StrictlyDecreasing$ and $c \geq p$. Verification directives are written as:

```
VERIFY MONOTONIC FEE ExcessClaimsFee WITH RESPECT
    TO ClaimCount
VERIFY MONOTONIC FEE DiscountFee WITH RESPECT TO
    YearsInProgram
  DIRECTION NonIncreasing
```

V. PROVENANCE AND AUDITABILITY

A. Execution Tracing in IPFLang

Each fee evaluation generates provenance records that comprehensively document the calculation process. These records capture the input parameter values used; all LET variable bindings computed during evaluation; each CASE and YIELD condition evaluated along with its true/false result; the selected YIELD expression with its computed value; and the final fee amount including currency designation. This trace enables practitioners to verify calculations against official schedules and assists in dispute resolution.

B. Auditability for Regulatory Compliance

Auditability extends provenance by structuring trace information for systematic review and verification. While provenance answers "how was this value computed?" auditability

addresses "can an independent reviewer verify computation correctness against authoritative sources?" IPFLang operationalizes auditability through structured audit records that include (1) the complete input parameter set with values and their sources; (2) the fee schedule version identifier linking to authoritative references; (3) each conditional branch evaluated with its Boolean outcome; (4) intermediate variable bindings with their computed values; and (5) the final fee amount with currency designation.

C. Counterfactual Analysis

Counterfactual analysis addresses a fundamental question in regulatory decision-making: what would the outcome have been under different conditions? In fee calculation contexts, this manifests as sensitivity analysis (how much would the total change if we filed as a small entity?), budget planning (what is the fee impact of adding five more claims?), and regulatory impact assessment (how do the new fee schedule rates affect our portfolio costs?). IPFLang's counterfactual engine adopts a simpler but more accessible approach: direct re-evaluation of fee computations with modified inputs, producing concrete alternative scenarios rather than logical proofs.

VI. Reference Implementation

A. Architecture Overview

The IPFLang reference implementation comprises approximately 15,000 lines of C# targeting .NET 10.0, organized into modular subsystems. The *Parser Module* performs lexical analysis, recursive descent parsing, and AST construction with comprehensive error reporting including line numbers and expected tokens. The *Semantic Checker* performs type validation ensuring identifier resolution, type compatibility, constraint validation, and circular dependency detection in LET statements. The *Type System* performs currency-aware type checking with the 161 active ISO 4217 currencies, polymorphic type support, and detailed type error reporting. The *Evaluator* performs depth-first AST traversal with environment binding for expression evaluation. The *Analysis Module* comprises a completeness checker with exhaustive and sampling modes, a monotonicity checker with four direction types, a domain analyzer, and a condition extractor. The *Provenance Module* handles audit trail recording, a counterfactual engine for what-if analysis, and provenance export formatting. The *Versioning Module* covers version metadata management, a diff engine for version comparison, an impact analyzer, and temporal query support. The *Composition Module* implements jurisdiction inheritance with input/fee merging, override detection, and inheritance analysis reporting.

B. Command-Line Interface

The CLI exposes five commands: `parse` validates syntax and types; `run` executes a fee calculation (with optional `--provenance` and `--counterfactuals`); `verify` runs verification directives; `info` displays script metadata; and `compose` combines jurisdictions (with optional `--analysis`).

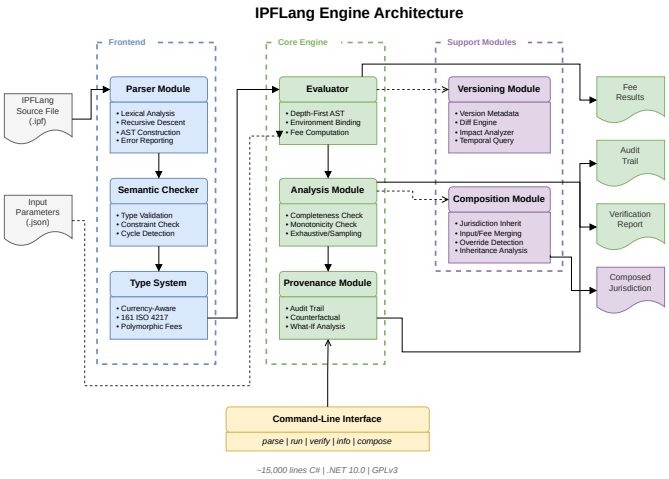

Fig. 2. IPFLang engine architecture showing the modular organization of parser, semantic checker, type system, evaluator, analysis, provenance, versioning, and composition subsystems.

C. Input/Output Formats

Inputs are provided as JSON. For the EPO filing example (`01_epo_filing.ipf`) with `{"EntityType":"LargeEntity", "ClaimCount":25, "SheetCount":35, "ISA":"ISA_NONE", "HasIPRP":false}`, the tool reports the following fees:

```
Filing Fee:          EUR 135.00
Designation Fee:     EUR 660.00
Excess Claims Fee:   EUR 2,650.00
Excess Pages Fee:    EUR 0.00
Search Fee:          EUR 1,460.00
Examination Fee:     EUR 1,840.00
--------------------------------
Total:               EUR 6,745.00
```

Listing 4. Computed fees for the EPO filing example.

D. Source Code Availability

The complete source code is available at https://github.com/vbocan/IPFLang under the GNU General Public License v3.0 (GPLv3). The repository includes the complete DSL engine source (approximately 10,000 lines of C#); a comprehensive test suite (approximately 5,500 lines comprising 269 test methods across 18 test categories); 20 IPFLang example files in the `examples/` directory (approximately 1,700 lines of DSL code); 118 production jurisdiction files plus 4 regional base files in the `jurisdictions/` directory covering PCT national/regional phase entry fees (approximately 21,800 lines of DSL code); and documentation with a syntax reference, a formal EBNF grammar, and a type system manual.

VII. Evaluation

A. Representative Examples

Our implementation includes 20 IPFLang example files demonstrating language expressiveness across diverse fee structure patterns, plus 118 production-ready jurisdiction files covering PCT national/regional phase entry fees for all major

patent offices worldwide. Real-world fee schedules are represented by EPO filing fees with multi-tiered claim pricing and ISA-dependent search fees, as well as a USPTO complete fee calculator with entity-based discounts and excess claim calculations. Feature demonstrations include currency type safety with mixed-currency error detection, entity-based discount patterns, temporal operations for date-dependent fees, nested CASE blocks for complex conditional logic, MULTILIST with `!COUNT` for designation fees, optional fees and versioning, and jurisdiction composition.

B. Jurisdiction File Validation

We validated the 118 production jurisdiction files by combining expert review with an automated consistency check. For fidelity to the authoritative sources, an independent domain expert in intellectual-property fee practice (acknowledged below) compared the fees IPFLang computes for representative applicant scenarios (spanning the entity, claim-count, page-count, and, where applicable, translation parameters that drive each schedule) against the official PCT national- and regional-phase entry fee tables in force during 2023–2024, with any divergence traced to the responsible fee definition and corrected during development. This was structured expert review rather than an automated oracle; the complete set of jurisdiction files is released so that the encoded fees can be independently rechecked against the published schedules. Separately, an automated harness we distribute with the implementation confirms corpus-wide well-formedness: all 118 files parse, pass the currency-aware type and semantic checks, and execute under default inputs to a defined fee total, and the 50 files that extend a regional base also execute when composed with that base. The exhaustive completeness and monotonicity checks of Section IV are applied to individual fee definitions and demonstrated on the example schedules rather than batch-run over the corpus, since for inputs with very large numeric domains the valuation space exceeds the exhaustive threshold and the framework falls back to boundary sampling. The jurisdiction files are organized hierarchically with four regional base files: `base_ep.ipf` (European Patent Convention member states, 38 countries), `base_ea.ipf` (Eurasian Patent Organization member states, 9 countries), `base_ap.ipf` (ARIPO member states, 19 countries), and `base_oa.ipf` (OAPI member states, 17 countries). Individual jurisdiction files either stand alone or inherit from and extend their regional base using the COMPOSE directive.

C. Test Suite

The test suite comprises 269 test methods across 18 test categories, summarized in Table III. All 269 tests pass with 100% success rate. On an AMD Ryzen 9 7900 (12 cores, 63 GB RAM) running Windows 11 Pro with .NET 10.0, test execution completes in sub-millisecond time per test, confirming that DSL interpretation imposes negligible overhead. We isolate this cost with an in-process micro-benchmark on the configuration above: parsing and type-checking a complete schedule takes approximately 0.22 ms (EPO) and 0.28 ms (USPTO), and a single fee evaluation takes approximately 0.10 ms and 0.56 ms respectively, on the order of 10^4 fee evaluations per second for the EPO schedule, confirming that the language imposes no practical performance barrier.

TABLE III
TEST SUITE BREAKDOWN

Category	Tests	Coverage
Calculator operations	11	Basic fee computation
Currency type safety	33	Type checking, 161 currencies
Completeness verification	21	Gap detection, sampling
Monotonicity verification	4	Direction enforcement
Temporal operations	54	Date calculations
Provenance tracking	20	Audit trails
Jurisdiction composition	19	Inheritance, overrides
Versioning	21	Diff, impact analysis
Other	86	Semantics, parsing, logic

D. Comparison with Catala and OpenFisca

To substantiate the qualitative positioning of Table I with evidence rather than description, we re-implemented three representative schedules (the EPO filing fees, the USPTO schedule, and a multi-currency example) in both Catala [7] and OpenFisca [13]. All three engines agree numerically across 16 input vectors spanning the three schedules, which isolates the comparison to language design rather than divergent arithmetic. The exercise confirms the central distinction: neither alternative expresses IPFLang's combination of currency-tagged amounts checked at compile time and static completeness and monotonicity analysis over the input space. In Catala, monetary values inhabit a single `money` type, so cross-currency sums type-check and the currency denomination is external convention; OpenFisca represents fees as floating-point values with no currency notion or static coverage guarantee. Catala, in turn, offers stronger formal foundations (a dependent-type lineage and optional solver-backed proofs), reflecting its focus on formal-methods users rather than the legal practitioners IPFLang targets. The complete re-implementations, container definitions, and validation harness accompany the open-source reference implementation.

E. Threats to Validity

Internal Validity. The test suite validates correctness of the implementation but may not cover all edge cases. The 269 tests focus on feature coverage rather than exhaustive input space exploration.

External Validity. The 118 production jurisdiction files covering PCT national/regional phase entry for major patent offices were validated by a domain expert against official PCT fee schedules. The validation confirmed calculations accurate to the currency unit across all supported jurisdictions.

Construct Validity. Performance measurements reflect test execution time, not isolated component performance. The keyword-based syntax (EQ, GT, AND) was designed for readability based on established DSL design principles [19], [20], but this design hypothesis has not been validated through controlled user studies.

Reliability. Results are reproducible via the open-source implementation. The validation test cases can be independently verified against official fee schedule documents.

VIII. Discussion

A. Advantages of DSL-Based Formal Specification

Transparency and auditability. Unlike black-box proprietary calculators, IPFLang scripts are human-readable specifications that can be audited directly. When fee schedules change, updates are visible diffs that can be reviewed without requiring deep programming expertise, though familiarity with the DSL syntax is necessary.

Formal verification. Static completeness checking (exhaustive mode) and monotonicity checking provide guarantees that are difficult to obtain in imperative implementations without comparable up-front analysis. For domains within the exhaustive threshold, practitioners can be confident that fee definitions cover all cases and behave predictably.

Vendor independence. IPFLang-based calculations are vendor-neutral: any conforming interpreter can execute scripts. Multiple tools can compete on user experience while using a shared, open calculation engine.

Rapid adaptation. Government fee schedules change frequently. IPFLang enables updates through script editing and verification, compared to longer cycles for traditional software development.

B. Cross-Domain Applicability

IPFLang's design patterns address common regulatory calculation structures beyond intellectual property. *Entity-based pricing tiers* apply different fees based on entity characteristics; in IP, the USPTO small- and micro-entity discounts illustrate this pattern: the earlier 50% and 75% reductions were raised to 60% and 80% by the Unleashing American Innovators Act of 2022 (in force from 2023), the schedule our production jurisdiction files encode. This is precisely the kind of statutory change that the version-tracked specifications of Section III are designed to accommodate. Cross-domain applications of the pattern include SME tax rates, tiered professional licensing, and asset-based regulatory fees. *Volume-based progressive pricing* uses marginal pricing where unit cost changes at thresholds; in IP, claim fees charge one rate for claims 1–15, another for 16–50, and a higher rate for 51+, with cross-domain applications including import duties, tiered utility pricing, and bulk discount structures. *Temporal dependencies* involve fees varying based on elapsed time; in IP, maintenance fees are due at specific intervals after grant, with cross-domain applications including late payment penalties, license renewals, and filing deadline calculations. *Multi-component additive fees* calculate totals as sums of independent components; in IP, totals combine filing, search, examination, and claim fees, with cross-domain applications including building permits, vehicle registration, and business licensing with endorsements. A domain suitability assessment suggests high applicability to professional licensing (very high structural similarity), court filing fees (very high), tax calculations for progressive structures (high, requiring negative value support), and customs duties (medium-high, requiring hierarchical types).

C. Limitations

Several limitations constrain the current work. *Implementation scope:* the reference implementation provides only a CLI interface; REST API support would enable broader integration with existing IP management workflows. *Jurisdiction coverage:* while the repository includes 118 production jurisdiction files covering PCT national/regional phase entry for major patent offices, extension to cover additional fee types (annuities, oppositions, appeals) remains important future work. *Type system expressiveness:* the type system does not support dependent types or refinement types that could express additional invariants such as requiring claim counts to be positive. *Empirical validation:* user studies validating syntax readability have not been conducted; design decisions favoring readability are based on DSL design principles rather than empirical evidence. *Boundary-based testing limitations:* for large input domains exceeding the exhaustive verification threshold (10^6 combinations), the boundary-based testing mode may miss gaps between sampled points.

IX. Conclusions and Future Work

This paper introduced IPFLang, a domain-specific language for formally specifying intellectual property fee calculations. The key contributions are: (1) a language specification with declarative syntax designed for readability, featuring keyword-based operators (EQ, GT, AND) rather than symbolic notation, explicit block delimiters, and domain-specific primitives for currency and temporal operations; (2) a currency-aware type system supporting the 161 active ISO 4217 currencies with compile-time prevention of cross-currency arithmetic errors, formalized through typing rules with a type safety argument; (3) static analysis algorithms for completeness checking (exhaustive verification for domains $\leq 10^6$ providing formal guarantees, boundary-based testing for larger domains) and monotonicity verification; (4) provenance tracking with counterfactual analysis supporting auditability requirements for regulatory compliance and dispute resolution; and (5) a reference implementation validated through 269 passing tests, verification against official EPO and USPTO fee schedules, and 118 production jurisdiction files covering PCT national/regional phase entry.

For practitioners, IPFLang offers transparent, verifiable fee calculations with audit trails. For patent offices, the specification gives machine-readable fee schedule definitions enabling third-party tool development. For the research community, IPFLang demonstrates that formal verification techniques can be applied to domain-specific regulatory contexts.

Several directions merit future investigation. On the implementation front, a REST API conforming to OpenAPI standards would enable enterprise integration, while community contributions could expand jurisdiction coverage beyond the current 118 files. Cross-domain pilots, particularly in tax calculations and customs duties, would test IPFLang's broader applicability. From a theoretical perspective, mechanizing the

type soundness argument in Coq or Lean would strengthen formal guarantees through explicit progress and preservation proofs over a defined operational semantics. Perhaps most importantly, user studies with IP practitioners would empirically validate the readability and usability claims that currently rest on design principles rather than measured outcomes.

ACKNOWLEDGMENT

The authors gratefully acknowledge Robert Fichter, Ph.D., of Jet IP for his meticulous verification of each jurisdiction implementation in IPFLang, ensuring calculations accurate to the currency unit across all supported patent offices. Special thanks to Adrian Ivan, M.Sc., of Storya Soft for comprehensive end-to-end application testing, and to Cătălin Bălăşcuţă for his valuable contributions in implementing recurring fee calculation patterns and supporting Fichter's validation work.

DECLARATION OF GENERATIVE AI USE

During the preparation of this work, the authors used Claude (Anthropic) to proofread the manuscript, including checking grammar, spelling, and phrasing consistency. After using this tool, the authors reviewed and edited the content as needed and take full responsibility for the content of the publication.

REFERENCES

[1] United States Patent and Trademark Office, "USPTO Fee Schedule," 2025. [Online]. Available: https://www.uspto.gov/learning-and-resources/fees-and-payment/uspto-fee-schedule

[2] European Patent Office, "EPO Schedule of Fees," 2024. [Online]. Available: https://my.epoline.org/epoline-portal/classic/epoline.Scheduleoffees

[3] Japan Patent Office, "JPO Fee Information," 2024. [Online]. Available: https://www.jpo.go.jp/

[4] World Intellectual Property Organization, "WIPO PCT Fee Tables," 2024. [Online]. Available: https://www.wipo.int/

[5] R. Bekkers and A. Updegrove, "A Study of IPR Policies and Practices of a Representative Group of Standards Setting Organizations Worldwide," in *Proc. National Academies Symp. Intellectual Property Rights*, Washington, DC, USA, 2012.

[6] T. Athan, G. Governatori, M. Palmirani, A. Paschke, and A. Wyner, "LegalRuleML: Design Principles and Foundations," in *Reasoning Web. Web Logic Rules*. Cham, Switzerland: Springer, 2015, pp. 151–188.

[7] D. Merigoux, N. Chataing, and J. Protzenko, "Catala: A Programming Language for the Law," *Proc. ACM Program. Lang.*, vol. 5, no. ICFP, Art. 77, 2021.

[8] World Intellectual Property Organization, "WIPO ST.96—Processing of IP Information using XML," 2023. [Online]. Available: https://www.wipo.int/standards/en/st96.html

[9] European Patent Office, "EPO Open Patent Services (OPS) API," 2024. [Online]. Available: https://www.epo.org/searching-for-patents/data/web-services/ops.html

[10] T. Hvitved, "Contract Formalisation and Modular Implementation of Domain-Specific Languages," Ph.D. dissertation, Univ. Copenhagen, Copenhagen, Denmark, 2011.

[11] A. Kennedy, "Types for Units-of-Measure: Theory and Practice," in *Central European Functional Programming School (CEFP 2009)*. Berlin, Germany: Springer, 2010, pp. 268–305.

[12] D. Orchard, V.-B. Liepelt, and H. Eades III, "Quantitative Program Reasoning with Graded Modal Types," *Proc. ACM Program. Lang.*, vol. 3, no. ICFP, Art. 110, 2019.

[13] OpenFisca Contributors, "OpenFisca: Open-source platform for tax-benefit microsimulation," 2024. [Online]. Available: https://openfisca.org/

[14] Red Hat, "Drools Business Rules Management System," 2024. [Online]. Available: https://www.drools.org/

[15] New Zealand Government, "Better Rules for Government Discovery Report," 2018. [Online]. Available: https://www.digital.govt.nz/dmsdocument/95-better-rules-for-government-discovery-report/html

[16] M. Waddington, "Rules as Code," *Law in Context: A Socio-legal Journal*, vol. 37, no. 1, pp. 179–186, 2021.

[17] J. Mohun and A. Roberts, "Cracking the Code: Rulemaking for Humans and Machines," OECD Working Papers on Public Governance, no. 42, OECD Publishing, Paris, France, 2020.

[18] World Trade Organization and World Economic Forum, "The Promise of TradeTech: Policy Approaches to Harness Trade Digitalization," WTO Publications, Geneva, Switzerland, 2022.

[19] M. Mernik, J. Heering, and A. M. Sloane, "When and How to Develop Domain-Specific Languages," *ACM Comput. Surv.*, vol. 37, no. 4, pp. 316–344, 2005.

[20] M. Fowler, *Domain-Specific Languages*. Boston, MA, USA: Addison-Wesley, 2010.

[21] T. Dardinier, G. Parthasarathy, and P. Müller, "Verification-Preserving Inlining in Automatic Separation Logic Verifiers," *Proc. ACM Program. Lang.*, vol. 7, no. OOPSLA1, Art. 80, 2023.

[22] Java Community Process, "JSR 354: Money and Currency API," Java Specification Request, 2015. [Online]. Available: https://jcp.org/en/jsr/detail?id=354

[23] A. K. Wright and M. Felleisen, "A Syntactic Approach to Type Soundness," *Information and Computation*, vol. 115, no. 1, pp. 38–94, 1994.

Valer Bocan (Senior Member, IEEE) received the Ph.D. degree in computer science from the Politehnica University of Timişoara, Timişoara, Romania. He is currently a Lecturer with the Department of Computer Science, Faculty of Automation and Computers, Politehnica University of Timişoara, and holds the CSSLP credential. His research interests include software engineering, software security, and domain-specific languages. (ORCID: 0009-0006-9084-4064.)

Valentina E. Balas (Senior Member, IEEE) is currently a Professor with the Faculty of Engineering, Aurel Vlaicu University of Arad, Arad, Romania. Her research interests include intelligent systems, fuzzy control, soft computing, and computational intelligence. (ORCID: 0000-0003-0885-1283.)