electronics

MDPI

Article

Transformation of Real-World Contracts to Smart Contracts for Blockchain Applications

Cecilia E. Chen [1,2], Xuanyu Liu [1], Limin Jia [3], Bo Liang [4,*], Yan Zhu [5] and Tong Wu [1,6]

[1] School of Computer & Communication Engineering, University of Science and Technology Beijing, Beijing 100083, China; chene@ustb.edu.cn (C.E.C.); tongw@ustb.edu.cn (T.W.)
[2] School of Computer & Communication Engineering, Shunde Innovation School, University of Science and Technology Beijing, Foshan 528300, China
[3] Zhejiang Neptune Technology Co., Ltd., Hangzhou 310022, China; 13575772566@126.com
[4] CCRC of State Administration for Market Regulation, Beiijng 100045, China
[5] National Center for Materials Service Safety, University of Science and Technology Beijing, Beijing 102206, China; zhuyan@ustb.edu.cn
[6] Guangdong Provincial Key Laboratory of Novel Security Intelligence Technologies, Shenzhen 518055, China
* Correspondence: liangbo@samr.gov.cn

Abstract

The widespread adoption of smart contracts, self-executing agreements on the blockchain, is hindered by the complexity of translating real-world contracts, often written in multiple languages, into their digital counterparts. This paper addresses this challenge by introducing an innovative approach based on Contract Text Markup Language (CTML), an extensible markup language specifically designed to facilitate the automatic generation of smart contracts from multilingual contracts. CTML overcomes traditional method limitations by employing a two-stage transformation process: (1) Contract Abstraction and Markup: CTML redefines grammar rules and incorporates encoding extensions to transform multilingual contracts into structured, marked-up contracts. This process effectively abstracts the essential details of the original contract, enabling language-agnostic interpretation. (2) Domain-Specific Language (DSL) Translation and Smart Contract Code Generation: The marked-up contract is then seamlessly translated into a DSL program, capturing the legal concepts in a machine-readable format. Finally, the DSL program is automatically compiled into executable smart contract code, ready for deployment on the blockchain. The effectiveness of the proposed approach is demonstrated using a legal contract in both English and Chinese. Therefore, the CTML-based approach can automatically generate smart contracts from multilingual contracts, enabling a more inclusive and accessible smart contract ecosystem.

Keywords: smart contracts; real-world contracts; automatic transformation; blockchain; Contract Text Markup Language

Check for updates

Academic Editor: Fabio Grandi

Received: 3 February 2026
Revised: 10 March 2026
Accepted: 30 March 2026
Published: 3 April 2026

Copyright: © 2025 by the authors. Licensee MDPI, Basel, Switzerland. This article is an open access article distributed under the terms and conditions of the Creative Commons Attribution (CC BY) license.

1. Introduction

A smart contract is essentially a computer program or protocol stored on a blockchain that automatically executes the terms of an agreement. Today, smart contracts have been a vital cornerstone for the development of blockchain applications. They provide developers with various programming languages, such as Solidity, Vyper, Yul, and Rust, which are utilized in the blockchain space for different applications. Moreover, by building a specific engine or execution environment, they can automatically execute contract terms without manual intervention, improving transaction efficiency and reducing transaction

costs. Therefore, smart contracts offer endless possibilities for blockchain application development, capable of fostering new business models and application scenarios and building a brand-new blockchain ecosystem.

A real-world contract represents a binding agreement between two or more parties, and each entity must fulfill its obligations. Smart contracts, in contrast, function as containers of code that encode and automate the execution of specific agreements within the blockchain network. They can be automatically executed when certain predetermined conditions are met. Smart contracts represent the terms and conditions of a real-world contract in a digital format, encoded in executable code. Smart contracts are not static digital replicas of real-world contracts; they are dynamic and can perform actions when certain conditions are met. This aligns with the essence of digital twins [1], as they are a digital model, a dynamic representation of an entity that simulates its real-world behavior based on data, and interacts with their physical counterparts [2].

Digital twins have emerged as digital virtual representations of actual physical assets, while smart worlds are virtual environments that simulate and optimize the performance of physical assets, systems, and processes in the real world [3]. The creation of digital twins requires reliable management and data monitoring, including appropriate secure policies for storage and analysis. Blockchain can provide several security functions for digital twins, such as traceability, tamper resistance, and encryption services implemented through smart contract interactions. Then, the cross-validated data can be stored on the blockchain to maintain its integrity and credibility. The product lifecycle events can be input into data-driven systems for process monitoring, diagnosis, and optimization control. Subsequently, digital twins can conclude the data to identify faults and recommend preventive measures before major events occur. The above process uses smart contracts as a connecting adhesive and places it in a narrow area between real (non-digital) and virtual (digital) fields, through which Artificial Intelligence (AI) and IoT can play a greater role in controlling economic activities in organizations.

The real-world contracts serve as agreements among involved parties to monitor, execute, and comply with civil or legal agreements and play a crucial role in maintaining order and the legitimate rights and interests of involved parties. These traditional contracts can be enforced through organizations concentrated in the law, but smart contracts can replace trusted third parties for specific, pre-defined tasks. Advances in computing and networking technologies and AI have paved the way toward the automation of establishing, monitoring, executing, and complying with real-world contracts [4].

Since real-world contract texts are written in natural language, they can be vague and subject to various interpretations [5]; therefore, monitoring and enforcing the execution of real-world contracts is a challenge. As an automated program on the blockchain platform, smart contracts can establish a trusted environment among multiple parties. Several advanced programming languages, such as Solidity and Serpent, can be used to write Ethereum smart contracts. The contract code is compiled into Ethereum Virtual Machine (EVM) bytecode and deployed for execution on the blockchain. Therefore, smart contracts improve existing contract management practices in real-world projects by replacing traditional contracts.

While the potential of smart contracts is undeniable currently, translating traditional contracts into computer code presents a significant current hurdle. This challenge stems from several factors:

- Ambiguity and Interpretation: Contracts, written in natural language, are inherently open to interpretation. Different parties and developers may have varying understandings of the contract's meaning, making it difficult to translate these nuances into the precise and unambiguous language of smart contracts.

- Limited Expressive Power: Current smart contract languages might lack the sophistication to fully capture the complexities of collaborative terms within contracts [6]. These terms often involve ongoing interactions and shared decision-making, which can be challenging to represent directly in code.
- Accessibility for Non-Programmers: Smart contracts are written in code, which can be a barrier for participants who lack programming knowledge. This creates difficulties for those who need to interact with the decentralized applications (dApps) built on these contracts.

Overall , these challenges make transforming contracts into a reliable and user-friendly computer-readable format a major obstacle that needs to be addressed.

Recent studies in high-impact journals have explored comprehensive frameworks for contract lifecycle management [7] and investigated advanced security-by-design patterns for decentralized applications [8]. While these works provide essential theoretical foundations for formal verification and reliability, the challenge of maintaining semantic consistency across multilingual real-world contracts remains underexplored.

Fan and Chen [9] proposed a technology for automatically generating smart contracts from real-world contracts, which involves automatically identifying and parsing contract content and transforming it into code or data structures for smart contracts. The key is to automate the process and try to reduce manual intervention. However, a real-world contract can be written in any language, including natural language, legal text, programming language, etc. The transformation process refers to transforming these different forms of contract content, whether natural language descriptions or formal code, into smart contracts. This may involve various technologies and tools that can handle input in different formats and languages.

In addition, there are other works on converting smart contracts [10–12]. Still, there is no method to support the automatic generation of smart contracts from real-world contracts written in different natural languages. However, research and tools that attempt to transform real-world contracts written in different languages into smart contracts are being developed. Here are some possible methods:

(1) Automated Conversion Tool: Develop automated tools to analyze the syntax and logic of real-world contracts and transform them into smart contract codes. This method requires a deep understanding of the syntax and semantics of different languages and the design of corresponding transformation rules.
(2) Manual conversion: Professional personnel manually transform the logic and conditions of real-world contracts into smart contract codes. This requires professional knowledge and skills that consume much time and effort.
(3) Intermediate representation language: Using intermediate representation language as a bridge to transform real-world contracts written in different languages into smart contracts. Intermediate representation languages can provide a unified syntax and logical structure, making transformations simpler and standardized.
(4) Deep learning technology: Utilizing deep learning technology to train models to learn transformation rules between different languages. This requires large data and computing resources and may have certain accuracy and applicability limitations.

Despite the advancements in existing contract modeling, annotation, and transformation approaches, a focused research gap remains: current methods lack a standardized, language-agnostic intermediate representation to fully automate the transition from multilingual natural language texts to executable code without heavy manual intervention. Transforming real-world contracts, written in various languages, into robust smart contracts remains a complex and ongoing challenge. To bridge this gap, several key steps are needed:

(1) Identify obstructive clauses: We must pinpoint specific clauses in real-world contracts that hinder a smooth translation to smart contracts.
(2) Multilingual transformation method: A method that facilitates the conversion of real-world contracts across different natural languages into smart contracts needs to be developed.
(3) Automated generation: Ultimately, the goal is to enable automated processes to generate smart contracts directly from real-world contracts.

This paper introduces an approach that supports the automatic transformation of real-world contracts written in any natural language into smart contracts. Specifically, we go beyond existing methods by offering the following concrete and verifiable contributions:

- Language-Agnostic Contract Abstraction via CTML: We use a human-readable language called CTML to automatically abstract and express the details of real-world contracts in marked-up contracts, overcoming the language dependence of traditional modeling methods.
- Automated Smart Contract Generation Pipeline: The CTML compiler can automatically compile executable code for generating smart contracts. This two-stage approach reduces manual intervention and automatically generates smart contracts from real-world contracts in various fields written in any universal natural language.
- Cross-Language Verification and Consistency: To validate our approach, we provide a detailed explanation using representative legal factoring contracts as an example. Through comparative analysis, we demonstrate the feasibility and consistency of this approach in transforming real-world contracts written in different natural languages (such as English and Chinese) into executable smart contracts.

To ensure clarity, we distinguish between smart legal contracts and executable contract codes. A smart legal contract refers to a legally binding agreement that incorporates natural language and computational logic. It emphasizes its legal attributes and business logic. It is a product of CTML annotating. Although it contains code logic, it is still considered a "contract." Executable contract codes refer to machine-readable code (such as Solidity Bytecode or compiled bytecode) that is ultimately deployed and runs on the blockchain. It is the final product of Stage 4.

Organization: The remainder of this paper is organized as follows. Section 2 provides a review of the current state of the art, comparing formal methods, template-based approaches, and NLP/LLM-based approaches with our proposed approach. Section 3 details the overall system framework. Section 4 focuses on the abstraction of real-world contracts, including the definition of CTML syntax and semantic markup rules. Section 5 describes the transformation of marked-up contracts into Domain Specific Language (DSL) and the subsequent generation of executable smart contract code. Then, Section 6 presents the experimental validation and case studies, and Section 7 describes the advantages and limitations of our approach. Finally, Section 8 concludes the paper.

2. Current State of the Art

There are currently methods to transform real-world contracts into smart contracts. Still, there are only a few methods to transform real-world contracts written in different languages into smart contracts. This is because there are significant differences in grammar and logical structures between different languages, requiring complex semantic parsing and transformation. Only certain aspects of the complete automatic generation process have been addressed. Existing methodologies can be categorized into three primary approaches: formal methods, template-based approaches, and Natural Language Processing (NLP)/Large Language Models (LLMs) approaches.

2.1. Formal Methods

Early research focused on using intermediate graphical or symbolic representations. Hamdaqa et al. [10] proposed a reference model for multi-platform deployment, but the language supports only basic graphical symbols, and it is difficult to be used for representing the details of real-world contracts. Frantz and Nowostawski [11] presented a modeling approach to support a semi-automated translation of human-readable contract representations into computational equivalents, but it is difficult to use the language grammar to express the complex semantics of the real-world contract, and it is difficult to automate the process of generating smart contracts.

Choudhury and Das [12] discussed enabling a seamless translation of constraints encoded in a knowledge representation to blockchain requirements. Their approach converts trading business rules into smart contracts, but it does not have a common contractual ontology definition and results in a semi-automatic generation process. Jurgelaitis and Butkiene [13] introduced UML class and state machine diagrams to model the structure and behavior of smart contracts. Their work emphasizes model-to-model transformations targeting the Solidity platform, yet it does not address the initial translation from natural language specifications to UML models.

2.2. Template-Based Approaches

Another significant research stream focuses on template-based approaches. Tateshi et al. [14] presented an integrated generation technique from contract documents to smart contracts based on templates and a controlled natural language (CNL). Their technique for generating templates is not automated. Also, real-world contracts need to be manually pre-modified to match the template or develop a new template to generate executable smart contracts.

Addressing this limitation, Chen et al. [15] presented the SPESC Translator that provides an effective method of understanding the content and meaning of real-world contracts through the development of a Smart Legal Contract Language (SLCL) [16]. The tool [17], developed by this method, can automatically compile SPESC contracts into executable code of the target platform. This approach, however, still relies on legal experts and programmers to work together and manually generate SPESC contracts based on the terms of real-world contracts.

While some template-based methods, such as SolcTrans Shi et al. [18], utilize Abstract Syntax Trees (AST) and predefined templates to facilitate the translation of smart contracts, their focus is on reverse engineering (translating code back to natural language). This differs fundamentally from our goal of forward transformation.

2.3. NLP/LLM-Based Approaches

Recent advancements leverage NLP and LLMs to automate the extraction of legal intent. Aejas et al. [19] utilized Named Entity Recognition (NER) and Relation Extraction (RE) to automate contract generation. However, the current implementation is limited to the NER phase, leaving the complete transformation pipeline unfinished. Fang et al. [20] introduced iSyn, a framework utilizing SmartIR (an intermediate representation) and NLP to synthesize smart contracts via template-based logic. While it narrows the semantic gap, it remains limited to mapping natural language sentences to predefined contract statements. Ahmed et al. [21] utilized NLP and blockchain for smart contract generation from legal documents. However, their approach is restricted to single-language contexts and lacks a deterministic, extensible markup framework to adapt to arbitrary natural languages.

Tong et al. [22] proposed an AI-assisted Smart Contract Generation (AIASCG) framework, which utilizes an AI-based segmentation technique (called SpIn) to facilitate multi-

lingual contract negotiation; however, it focuses on reducing manual drafting workload rather than the full transformation into executable code. With the emergence of Generative AI, recent studies have explored LLM-driven pipelines for direct text-to-code translation. Napoli et al. [23] introduced an LLM-based pipeline for generating smart contracts from text. However, its reliability depends on the model's inherent capabilities rather than a deterministic framework, making it prone to syntax inconsistencies and vulnerabilities in complex, multilingual scenarios. Similarly, Santos et al. [24] performed a comparative evaluation of LLMs for translating legal texts into smart contracts, focusing on model efficiency and vulnerability risks. However, their study assesses existing AI tools rather than proposing a universal, extensible methodology that ensures structural consistency across multiple languages. Barbàra et al. [25] highlighted the failure of LLMs in smart contract generation due to stochastic behaviors and hallucinations. Their study underscores the necessity of a standardized intermediate representation to reliably bridge the semantic gap in multilingual contexts.

2.4. Comparison with Our Works

While existing methods have addressed specific aspects of the generation process, a significant gap remains in achieving high automation, semantic fidelity, and multilingual adaptability simultaneously. Table 1 provides a structured comparative analysis of the current state-of-the-art across four dimensions: automation level, semantic fidelity, verifiability, and multilingual support.

As shown in Table 1, NLP/LLM-based approaches [19–25] prioritize high automation but suffer from "semantic black-boxes" and probabilistic outputs, which fail to meet the deterministic execution requirements of legal contracts. Conversely, formal methods [10–13] ensure maximum rigor yet impose a prohibitive technical barrier on legal practitioners. While template-based approaches [14–17] attempt to bridge this gap, they often necessitate a mandatory re-encoding of the original prose into machine-specific syntax.

Table 1. Structured comparative analysis of smart legal contract transformation approaches.

Approach	Recent Refs	Automation	Semantic Fidelity	Verifiability	Multilingual
Formal Methods	[10–13]	Low (Manual)	Very High	Strong (Proof)	Low
Template-based	[14–17]	Medium	Medium	Medium	Low
NLP/LLM-Based	[19–25]	High (Auto)	Low (Probabilistic)	Weak (Black-box)	High
Our Approach	this paper	High (Markup)	High (Deterministic)	Strong (Compiler)	High

In contrast, our proposed CTML framework occupies a strategic middle ground. By combining a deterministic markup grammar with a compiler-driven pipeline, CTML achieves high automation and semantic integrity without compromising the legal completeness of the original multilingual text. This positioning effectively addresses the long-standing tension between technical automation and legal precision.

Furthermore, Fan and Chen et al. [9] proposed an approach to automatically generate smart contracts from real-world contracts. However, a real-world contract can be written in any language. Therefore, we will continue to explore how to transform real-world contracts in any language into smart contracts. Fan's approach emphasizes the automation of the generation process, with the goal of minimizing reliance on human intervention. This paper emphasizes flexibility and adaptability, which can handle input from multiple languages and transform them into smart contracts.

3. Our Overall Approach

For a legal contract, textual annotation of unstructured linguistic information is essential to the "knowledge bottleneck" in legal information processing and more advanced

applications. In this paper, we establish a method to normalize the content and meaning of legal contracts by designing a CTML similar to HTML, XML, and SGML. The CTML will help overcome the bottleneck, and the extracted information can be submitted for further processing of smart legal contracts. To be precise, this language is defined as follows:

Definition 1 (Contract Text Makeup Language, CTML)**.** *Contract Text Markup Language is a text encoding system consisting of symbols or tags inserted into a contract document to express its structure, format, meaning, or relationships between its parts.*

Before we present our overall approach, we need to discuss our framework for the content abstraction of real-world contracts in a natural language using the CTML. This framework will facilitate processing unstructured linguistic texts and extracting information in a structured language. In our approach, automatic smart contract generation involves marking up a natural language contract with CTML, converting it into a DSL-driven smart contract, and generating executable smart contract code, ensuring the same meaning. CTML extracts contract elements accurately by marking up contract grammar, structure, and vocabulary. CTML translates real-world contracts to smart contracts:

- Extracting contract structure, key elements (e.g., involved parties, subject matter, price, time, place of performance, liabilities, dispute resolution methods), and their attributes, forming a foundation for contract processing and application.
- Ensuring consistency between the original contract and the resulting smart contract while automatically transforming the text using the contract template's regular grammar and vocabulary, enhancing efficiency and accuracy in smart contract development.

3.1. Establishment Principles

CTML enables the formal representation of contract elements at varying granularity. Designing DSL-driven smart contracts should adhere to the following principles:

(1) Use marks or tags to extract complete meaning and establish the mapping relationship between contracts and DSL-driven smart contracts.
(2) Standardize DSL-driven smart contract programming using CTML transition rules to enhance the efficiency of transforming contract text to executable code.
(3) Create a systematic framework to support "templates" that can be customized using CTML variables.

Consequently, CTML contracts comprise three elements: parameters (variables), program code (marks/tags and transition rules), and contract text, adhering to the principles. Our approach will focus in-depth on the legal contracts in English as an example to illustrate the automatic generation of smart contracts and the use of CTML. The Exchange Markup Data (EMD) is a data table containing key-value pairs describing metadata markup tokens in a CTML contract, which aids in confirming, specifying, and reducing client interactions with these tokens. The EMD also supports the idea of a contract template in three ways:

(1) It supports the man-machine interface based on an offer-acceptance system at the contract level, providing a clear binding framework for the formation of contracts;
(2) It supports automated data exchange at the user level, enabling user customization to suit the specific needs of the parties involved;
(3) It achieves separation of code (representing contract semantics) and data (contract instances) at the technical level, ensuring that the code can be developed and deployed in advance and has broader application.

As discussed in the approach, our focus is on designing a grammatical specification to establish semantic relationships between real-world contracts and smart contracts, which can be divided into two stages:

- Exact expression of contracts' meaning: By adding contract element symbols and attributes from contract text, contract elements, and content interpretation can be accurately and unambiguously determined.
- Computerized understanding of contracts: By defining semantic transition and data processing methods on extracted contract elements, marked contracts can be transformed into DSL-driven contracts and finally into smart contracts.

3.2. System Framework

In our approach, CTML supports extensions to apply to any natural language. For real-world contracts in any natural language, the transformation and automatic generation from real-world contracts to smart contract programs is illustrated in Figure 1. This process begins with the syntactic analysis of the natural language in which the contract is written, extracting the contract into a markup language, and then translating the markup language into a domain-specific language (DSL). The DSL program is compiled into executable code. Our approach consists of the following stages:

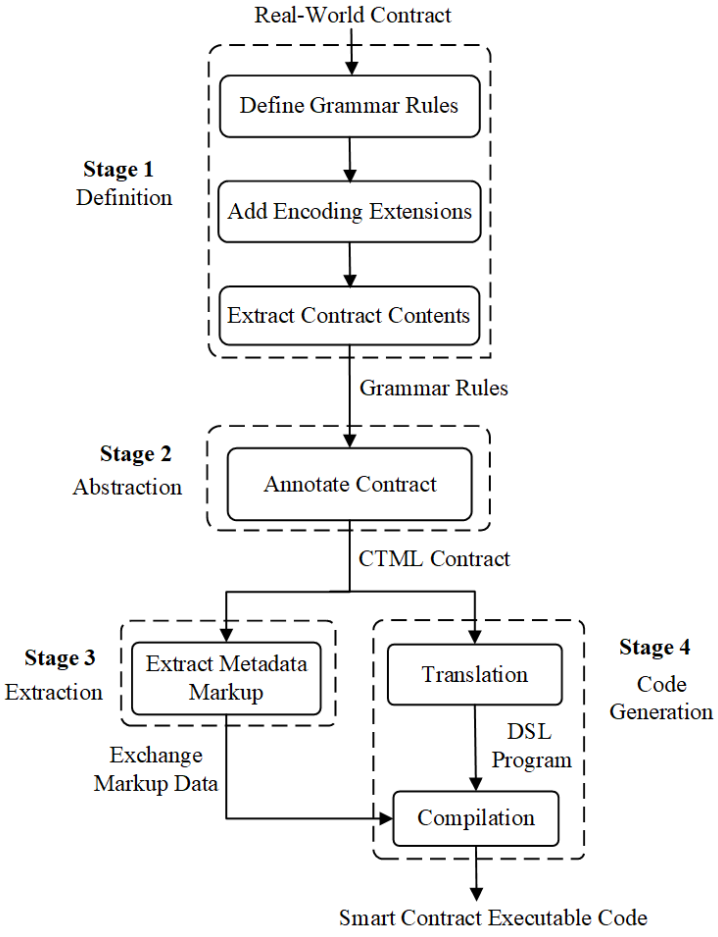


Figure 1. The system framework of the CTML-based transformation approach.

(1) **Stage 1. Definition of grammar rules:** Perform the grammar analysis on the natural language used to write real-world contracts, and the process includes:

- Define Grammar Rules: Identify the natural language of real-world contracts and define relevant grammar rules in CTML.

- Add Encoding Extensions: Add encoding extensions for a specific language to existing rules.
- Extract Contract Contents: Understand and extract the corresponding contract contents based on the definitions of different languages and output grammar rules.

(2) **Stage 2. Markup of Real-world Contract:** The process includes: Transforming CTML contracts involves annotating the contracts for automated processing and understanding the contract regarding the grammar rules.

(3) **Stage 3. Extract Metadata Markup:** Metadata Markup from these CTML contracts is extracted into EMD, which provides structured and specific contract information.

(4) **Stage 4. Code Generation:** Complete the mapping from CTML to smart legal contract language through model transformation. The process includes the following:

- Translation: Translating CTML contracts generated in Stage 2 to DSL using lexical mapping and transformation rules to generate DSL programs.
- Compilation: Compiling the DSL program and linking the program with EMD generated in Stage 3 to generate the smart contract executable code and build a human-machine interface of metadata.

In summary, the main process of our approach can be simplified as follows: Stage 1 is to define the grammar rules of real-world contracts, followed by Stage 2 to abstract the real-world contracts, and Stage 3 to extract the metadata markup into the EMD. Finally, Stage 4 uses a code generator to sequentially complete the translation and compilation, thereby generating smart contract executable codes. However, it's important to note that while our method theoretically supports any natural language, this support is built upon the scalability of the CTML framework. Specifically, for a new natural language, a customized Stage 1 is required, namely, defining the language's syntax rules and adding encoding extensions using tools such as Xtext. Once the contract content is mapped to structured CTML markup, the subsequent DSL conversion and code generation stages enter a language-independent automated pipeline, thus ensuring a robust conversion from multilingual legal texts to consistent smart contract code.

Building upon the automation framework in [9], this paper shifts the focus toward multilingual adaptability, ensuring that real-world contracts can be transformed into smart contracts regardless of natural language differences. While [9] prioritized minimizing human intervention, our approach emphasizes a flexible, universal transformation methodology. Technically, the original three-step process is expanded into a four-stage pipeline: Stage 1 handles initial language abstraction, while the refined Stages 2, 3, and 4 address the core semantic and generation challenges. These stages are detailed in Section 4 (Stages 1–3) and Section 5 (Stage 4).

To differentiate the manual, semi-automatic, and fully automatic steps required for each transformation stage, we qualitatively define the nature of the work in each stage in Table 2. Stage 1 is manual. Manually defining the grammar requires repeated communication between legal experts and programmers. Stage 2 is semi-automatic. While the identification of legal clauses requires human expertise, the CTML editor provides real-time syntactic validation and auto-completion based on the predefined grammar. This significantly reduces human error compared to manual coding. Stages 3 and 4 are fully automatic. Once the marked-up contract is fed into the CTML compiler, the translation to the DSL and the subsequent generation of Solidity code are performed by the system without any human intervention (Zero-human-in-the-loop). This separation ensures that legal precision is maintained by human experts while execution logic is reliably handled by automated generation.

Table 2. Automation level of our approach.

Step	Automation Level	Required Manual/Computing Power
Grammar Definition in Stage 1	Manual	One-time Developer Definition (Xtext)
Markup/Annotation in Stage 2	Semi-automatic	Manual Identification Logic + CTML Plugin-Assisted Filling
DSL Instantiation in Stages 3 and 4	Fully automatic	Millisecond-level Compiler Processing
Code Generation in Stage 4	Fully automatic	Millisecond-level Code Mapping

4. Abstraction of Real-World Contracts

The syntax rules of CTML are implemented using Xtext [26], an open-source software framework for developing domain-specific languages (DSL) [27] based on the Extended Backus-Naur Form (EBNF) paradigm rules [28]. Xtext generates the class model of Abstract Syntax Tree (AST) corresponding to this language model [29], which can effectively identify the elemental hierarchy of instances and facilitate the architectural analysis of instances. Xtext's "Validation Package" module [30] allows for the specification of validation rules for a specific domain language, implementing constraints on it. This module offers features like auto-completion, syntax coloring, renaming, refactoring, bracket matching, auto-editing, outline view, and code formatting to ensure proper indentation and organization of documents.

The CTML syntax design is divided into three main parts:

(1) Document type description: This declares the document type (DOCTYPE) using a short token string and is only used for schema selection.
(2) Document header: Contains essential information for CTML documents, such as text language.
(3) Document Body: The main content is marked with CTML tags based on the original contract. It's divided into two parts:
    - Semantic Expression: Indicates elements and meanings in contracts, providing a basis for smart contracts through semantic markup.
    - Metadata Expression: Marks contract templates or incomplete contracts to identify and locate the parts to be filled automatically.

CTML syntax follows nested structures and common notation rules, with syntax rules for factors, properties, components, and metadata provided in Table 3. The CTML syntax rules are defined as "CTML Annotation Guideline." As the introduction mentions, we will illustrate an example using a legal contract. The syntax is designed with the nature of real-world legal contracts kept in mind.

Nested structures refer to the identifications or names composed of element identifiers, attributes, and components through hierarchical relationship connectors "." to identify the inclusion relationship between elements. Due to the disorder of contract text, it usually does not have the general clear structure of a metamodel, resulting in the heterogeneity of legal elements and legal attributes related to legal elements. Therefore, grammar rules allow legal attributes to indicate the corresponding element identifier through the hierarchical relation connector "." when using this annotation form. Also, it can exist at the same level as legal elements at the annotation level.

To ensure the rigor of legal logic transformation, a standardized annotation protocol was established: the annotation team consists of researchers with legal backgrounds and blockchain developers, who divide tasks according to a predefined formal CTML Annotation Guideline. The workflow follows "Independent Annotation" → "Consistency Check" → "Expert Review". The annotation process follows the principle of "independent annotation by two or more people", meaning that for each contract case, two or more annotators

independently perform CTML tag mapping. Also, we conducted a cross-verification of the markup results to evaluate reliability. For example, we require the inter-annotator reliability to reach a score of 0.88 in the annotation of core legal elements (such as the parties, payment triggering conditions, and liability for breach of contract). It demonstrates that the annotation results possess high repeatability and reliability."

Table 3. The syntax definition of CTML (CTML Annotation Guideline).

Labeling Module		Syntax Definition
Document Format		`<!DOCTYPE ctml><ctml lang=zh_cn>«factor title@printerDeal»text«/factor»</ctml>`
Semantic Markup	Law Factor	`factorExpression ::= «factor factorSet@factorID(#attribute=value)+»text«/factor» factorSet ::= {title, party|group, asset, genTerm|breTerm|arbiTerm, conclusion, addition}`
	Law Property	`propertyExpression ::= «property[factorID.]propertySet[@propertyID] (#attribute=value)+»text«/property» propertySet ::= {info, right, action, preCondition, adjCondition, postCondition, against, controversy, institution, signature}`
	Law Component	`componentExpression ::= «component(#attribute=value)+»text«/component» component ::= {actionTime, timePredicate, rangePredicate, deposit|withdraw|transfer, assetExpression}`
	Field	`fieldExpression ::= «field [factorID.propertySet]@fieldID [%type] [#quantity=value]»text«/field»`
Metadata Markup		`metadataExpression ::= <{[factorID]@exchangedDataID [%type] (#option=value)+}>`

4.1. CTML Semantic Markup

Semantic markup uses a hierarchical markup structure to markup contract text. The hierarchical markup structure divides the text into different levels based on the "inclusion relationship" between the textual information of the contract, thus allowing the user to refine the text layer-by-layer in a "from large to small" and "from coarse to fine" order.

Considering the high universality of natural language, the connection from the contract instances to the target smart legal contract language involves the interpretation of certain elements. Therefore, the concept of "attribute" has been added to the tag as a parameterlist within the tag to provide expansion space. The definition is as follows:

Definition 2 (Attribute)**.** *An attribute of a tag is called an attribute. It allows content to be defined and given meaning as an "attribute" that cannot be marked in natural language alone.*

Semantic markup provides information about the content, allowing CTML to create contract templates with a clear hierarchical structure. These templates can be transformed into smart legal contracts. Semantic markup uses double-pointed brackets and defines various markup functions:

(1) Law Factor Mark (LFM): A first-level function that identifies and represents legal factor elements.
(2) Law Property Mark (LPM): A secondary function that extracts legal properties under the legal factor.

(3) Law Component Mark (LCM): A third-level function that identifies and represents law components in a nested manner.

Definition 3 (Semantic Markup)**.** *Semantic markup is used to annotate law factors, law attributes, law components, and domains in a document using the following format:*

$$\begin{aligned} SemanticExpression ::= &\ll \textbf{\textit{factor}} \mid \textbf{\textit{property}} \mid \textbf{\textit{component}} \\ &\mid \textbf{\textit{field}} \quad parameterList \gg text \ll \textbf{\textit{factor}} \\ &\mid \textbf{\textit{property}} \mid \textbf{\textit{component}} \mid \textbf{\textit{field}} \gg \end{aligned}$$

where ***factor, property, component,*** *and* ***field*** *are the reserved words for legal elements, properties, components, and fields, respectively; parameterList denotes the list of parameters, which has different definitions depending on the type of tag; and text denotes the marked text.*

Semantic markup defines a generalized format for CTML markup or tags that are not directly used for text annotation. Additionally, specific text markup processes use auxiliary marks like Field Mark (FM). Maintaining a hierarchical method when using these markup functions is essential to define the relationships between LFM, LPM, LCM, and FM. When using these four types of markup, a hierarchical markup approach should be used to specify the relationship between the four types of markup (i.e., LFM, LPM, LCM, and FM).

To implement generic rules in CTML, the following scenarios need to be discussed, as shown in Figure 2.

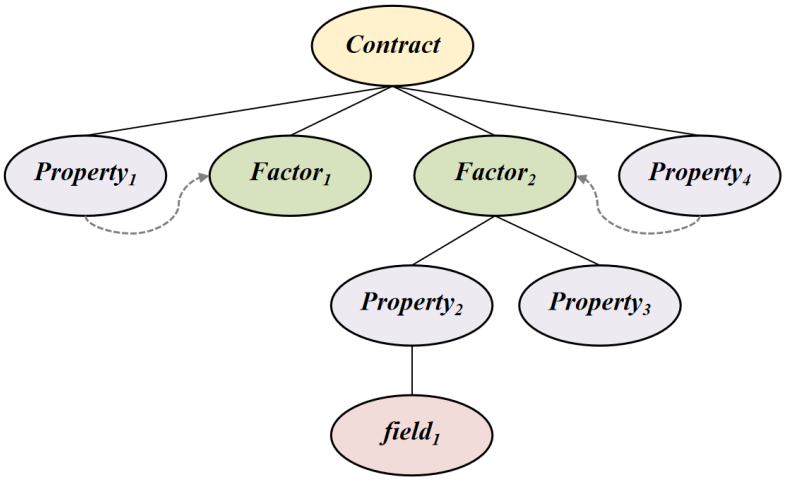


Figure 2. Diagram of a CTML syntax tree.

(1) In the syntax tree, the domain $field_1$ is a child node of $Property_2$. If $field_1$ requires an element from a property identified by nesting rules, the cross-reference can only retrieve the relevant instances on the branch by default. This means that only $Property_2$ and $Property_3$ rather than $Property_1$ or $Property_4$ can be retrieved to satisfy the required syntax rules of CTML implementation. The non-nested forms of legal properties are referred to as $Property_1$ and $Property_4$.

(2) $Factor_1$ includes $Property_1$. If the domain $field_1$ is included in $Factor_2$, the cross-reference of $field_1$ should not retrieve $Property_1$ because it does not have a containing relationship with $Factor_2$.

Therefore, based on the above description, it is necessary to redefine the cross-reference scope to improve and limit the retrieved results of Property [31]. This re-defined cross-reference scope should undergo the following process:

(1) Obtain the referenced Factor instance;

(2) Iterate through the CTML's AST to extract the legal attributes that point to the Factor entity, including attributes of nested forms within element entities and non-nested forms pointing to element instances with unique identifier names;
(3) Return the attribute set that meets the specified condition.

According to the above process, the optional range of *Property* is transformed into the properties contained in the element instance based on the selected element instance. The pseudo-code is shown as Algorithm 1.

Algorithm 1 getScope

Input: field, context, reference
Output: A list of Property

```
function GETSCOPE(field, context, reference)
    factor ← getFactor(field.factorID)
    if factor is null or factor.eIsProxy is null then
        return super.getScope(context, reference)
    else
        propertyList ← extractProperty(context, factor.name)
        for property ∈ getProperties(context) do
            if property.name is not null then
                if property.factorID == factor.name then
                    addElement(property, propertyList)
                end if
            end if
        end for
        return propertyList
    end if
end function
```

It should be noted that the definition of grammar rules and encoding extensions is a pre-configuration phase. Although this requires domain expertise, it is a one-time investment. Once the CTML for a specific domain (e.g., factoring) is established, it serves as a standardized template that supports the automatic processing of all subsequent contracts within that domain.

4.2. CTML Metadata Markup

Metadata markup labels textual elements refer to the exchanged data within the contract template. In CTML, the metadata often includes the basic, customized, and natural vocabulary that the contracting parties use to declare, fill in, or select real-world contracts.

After marking the CTML document body, the metadata markup can be extracted to create an Exchange Markup Data (EMD), which is a data table in the form of a "key-value" pair. The EMD can be used to implement the principle of "separating code and data" proposed in our approach.

Definition 4 (Metadata Markup)**.** *The metadata markup is defined in the following format:*

$$\begin{aligned} \textit{metadataExpression} ::= <\{&[\text{factorID}]@\text{exchangedDataID} \\ &[\%\text{type}]\ (\#\text{option}=\text{value})+\} > \end{aligned}$$

where, factorID *denotes the outermost factor identifier in the hierarchy to which the interaction data belongs;* exchangedDataID *denotes the unique identifier of the interaction data;* value *indicates the alternative data or value taken here for the contract text;* type *indicates the type of the interaction*

data, which can be divided into dataType and rightType; option *indicates the selection method of the interaction data, which is represented by*

$$optionSet ::= \{\text{singleOption}|\text{multiOption}, \text{import}, \text{trigger}, \text{allocate}\}$$

where, import *means that it can receive user incoming data,* trigger *means that it can receive external events, and* allocate *means that it can receive complex data types.*

The annotation process of Metadata markups has the following steps:

(1) Select the texts of interaction data and specify the data types in the entire contract.
(2) Obtain the interactive data identifier of Metadata containing the element identifier factorID for use or revision in the follow-up operations.
(3) Configure the options of the Metadata's interactive operations for the human-machine interface based on the data type, where the option includes *singleOption, multiOption, external import, trigger*, and *allocate* to normalize the operation behavior.
(4) Content is stored based on the specified data identifier, data type, and option of interactive operations into the user-specified location or replacing the selected text with the Metadata.

5. Code Generation

The marked CTML contracts need to be mapped to the DSL-driven contracts. Model transformation can achieve this goal by transforming CTML entities into predefined DSL objects, resulting in well-defined, easy-to-understand, and enforceable DSL-driven contracts. A code generator, which will be referred to as a Generator, typically manages the above model transformation process for a given DSL language. In our research, the SPESC built on Xtext serves as the target language for code generation. The SPESC language syntax is defined in Table 4.

The SPESC has defined a series of objects, including parties, assets, fields, additional information, terms, and signatures. Importantly, each of them has the same corresponding elements in CTML. The Generator maps the elements of the CTML contract's AST and their related information into the objects of the SPESC contract according to the element types and then writes these maps into the generated SPESC contract. The Generator's workflow involves the following steps:

(1) Pre-processing: Extracting domain marks and non-nested properties from the AST and restructuring the AST to avoid repeated content and omitted subjects.
(2) Recursion: Filtering out information on the specific Element entities based on the AST.
(3) Generating factors: Creating all factor entities in the SPESC contract based on AST's mapping relationship.
(4) Generating property, component, or field: Making a pass through all factors to generate properties, components, or fields based on the mapping relationship in every iteration.

Before the generator maps the AST, it must preprocess the AST to transform the entities of the CTML semantic model into the SPESC contract elements. It is necessary because the AST does not follow the hierarchical structure among elements in the SPESC contract. Preprocessing classifies and extracts all markups and entities not nested in the AST to ensure all elements are preserved after mapping.

The preprocessing of field markups involves extracting the semantic relationships from the CTML contract text. In this process, the function *getAllContentsOfType()* [31] (the Java class of the Xtext) is used to retrieve all elements of a specific type in the tree by starting at the AST root node. In Algorithm 2, we demonstrate the pseudocode of the above

field processing for removing nested field markups to prevent duplication. Algorithm 2 transforms field, component, and property from CTML contract to DSL program (Stage 4).

Table 4. Contract syntax for SPESC.

Contract Mode		Grammar
Architecture		**Contracts** ::= **Title**{ **Parties**+ **Assets**+ **Terms**+ **Additions**+ **Signs**+ }
Title		**Title** ::= **contract** Cname (: **serial number** Chash)?
Party		**Parties** ::= **party group**? Pname {field+ }
Asset Expression		**Assets** ::= **asset** Aname{ **info**{ field+ } **right**{ field+ } }
Asset Operation	Deposit	**Deposits** ::= **deposit** (**value** RelationOperator)? **AssetExpression**
	Withdraw	**Withdraws** ::= **withdraw AssetExpression**
	Transfer	**Transfers** ::= **transfer AssetExpression** to target
Time Expression	ActionEnforcedTimeQuery	**ActionEnforcedTimeQuery** ::= (**all** | **some** | **this**)? party **did** action
	TimePredicate	**TimePredicate** ::= (targetTime)? (**is** | **isn't**) (**before** | **after**) baseTime
	BoundedTimePredicate	**BoundedTimePredicate** ::= (**within**)? boundary (**before** | **after**) baseTime
Term	General Term	**GeneralTerms** ::= **term** Tname: Pname (**must** | **can** | **cannot**) action(field+) (**when** preCondition)? (**while transactions**+)? (**where** postCondition)?
	Breach Term	**BreachTerms** ::= **breach term** Bname (**against** Tname+)? : Pname (**must** | **can**) action(field+) (**when** preCondition)? (**while** transactions+)? (**where** postCondition)?
	Arbitration Term	**ArbitrationTerms** ::= **arbitration term** : (The statement of any controversy)? administered by **institution** : instName
Signature		**Signatures** ::= **Signature of party** Pname : { printedName:string, signature: string, Date: date }
Addition Information		**Additions** ::= field +
field		attribute : (constant | type)

Similarly, the non-nested CTML properties can be extracted from the AST root node. Since the property is an abstract entity, the property type must be determined. A list of different property entities is acquired using the above extraction method.

Algorithm 2 fieldProcess

Input: contract, EcoreUtil2
Output: A list of field

```
function FIELDPROCESS(contract, EcoreUtil2)
    root ← EcoreUtil2.getRootContainer(contract)
    fieldList ← EcoreUtil2.getAllContentsOfType(root, Field)
    for field ∈ fieldList do
        if field.factorID is null then
            if field.propertyID is null then
                fieldList.remove(field)
            end if
        end if
    end for
    return fieldList

end function
```

According to the preprocessing results, the Generator consists of the following two parts:

- Generate the corresponding factor entities/objects according to the mapping relationships.

- Iterate through the information contained in the entities and generate the corresponding properties, components, or field entities in the SPESC contract.

Explaining the Title transformation processing as an example, the syntax in the SPESC language is as follows:

$$\textbf{Title} :: = \text{contract Cname } (: \textbf{serialNumber} \text{ Chash})?$$

where the Cname and the Chash correspond to the contract's titleID and the serial number serialNo, respectively. The property value of the terms can be extracted from the CTML markups. Therefore, the SPESC contract code is generated as "**contract** titleID: **serialNumber** serialNo."

The mapping of the field information into the target contract as mentioned in (4) of the CTML Generator's workflow. We get the field entity in terms of the following field syntax from the SLCL:

$$\boldsymbol{field} ::= \text{attribute} : (\text{constant} \mid \text{type})$$

According to this syntax, the corresponding words of "attribute" and "constant| type" are filled into the syntax as the SPESC contract code. In CTML, the fieldID from the field markup is used as an attribute in SPESC. Similarly, the value or type from the field markup is as "constant" or "type" in the above syntax. In addition, the factorID and propertySet from the field markup are used to locate the place on which the SPESC contract code "fieldID : (value|type)" will be written.

6. An Illustrative Example

In the realm of smart contracts, the common legal textual contracts include sales contracts, lease contracts, etc. According to Article 761 of "The Civil Code of the People's Republic of China", a factoring contract refers to a contract where an account receivable creditor transfers existing or future accounts receivable to a factoring party who provides services such as fund financing, account receivable management or collection, and payment guarantee for account receivable debtors. Its textual elements include contract name, parties involved, subject statement, and clauses that can fully express the requirements for different levels of CTML markup.

Considering that research on language for annotating contract texts is still in its early stages, smart legal contracts are mainly applied in economic transaction scenarios, with most designs based on such application scenarios. Therefore, this section will primarily demonstrate the practical effects of semantic markup and code generation using CTML in well-known factoring contracts from selected samples recommended by industry associations.

Our approach can transform a real-world contract in any natural language into a smart legal contract, which means CTML supports extensions applied to any natural language. This extension requires redefining the grammar rules related to texts in CTML and adding encoding extensions for specific languages to existing rules. Subsequently, the CTML compiler can understand and extract the corresponding contract contents based on these text definitions. However, the corresponding code generators generated using our approach are also different for real-world contracts in different natural languages.

We will use a factoring contract written in different natural languages (such as Chinese and English) as an example to demonstrate our approach. Moreover, after automatically transforming the real-world contract annotated with CTML into a smart legal contract through a code generator, the extraction of contract text is clear, the conversion is standardized, and the arrangement of legal contract elements is more organized.

6.1. An Example Contract Written in English

To facilitate the understanding and readability of readers from different countries, according to the universality and universal applicability of natural language types, we will use the factoring contract in English [32] shown in Figure 3a (translated from the appendix of T/CATIS 003-2020 [33]) to illustrate our approach.

(1) Stage 1: Analyze the grammar in English in the factoring contract.

- Identify the natural language for writing contracts as English and define grammar rules related to the text in CTML.
- Add encoding extensions for English to existing rules.
- Based on the above text definition, extract the corresponding contract contents and define "< ctml lang=zh_en > " in the document type description (Table 3).

(2) Stage 2: Following CTML grammar rules and annotation methods, the factoring contract is transformed into a CTML contract. Figure 3b shows the marked-up CTML contract. Semantic extraction annotates a basic factoring contract divided into six modules: contract title, parties, additional information, assets, terms, and contract conclusion.

(3) Stage 3: The key-pair values for EMD from the CTML contract are extracted. That is, the hierarchical markup method is applied to mark the semantic elements of the contract, and metadata markup is used to generate an EMD for data interaction.

(4) Stage 4: Automatically transform a real-world contract annotated by CTML into a smart legal contract through a Generator.

- To generate the smart legal contract using CTML, the SPESC code for the CTML contract can be automatically generated using the Generator, resulting in a smart legal contract.
- After SLC is generated using CTML, the code can be linked with the EMD, compiled to generate a smart contract executable code, and sent to the Smart Contract Library for contract deployment.

Notice of Assignment of Accounts Receivable

No.:

To: _____(hereinafter referred to as "you")

From: _____(hereinafter referred to as "we")

For the needs of our business development,we have applied to transfer the following accounts receivable and all associated rights and interests to _____(hereinafter referred to as the "commercial factoring company"). We hereby notify you:

1.The following item(s)of accounts receivable and all associated rights and interests have been transferred to the commercial factoring company:

1) All accounts receivable between us and you under _____.

2) All accounts receivable listed below from you to us:

Item Number	Debtor of Accounts Receivable	Basic Education Contract and Number	Document Name	Document Number	Amount of Accounts Receivable	Due Date of Accounts Receivable

2.Please pay the full amount of the above accounts receivable to the following account on the due date(Account Holder:_____;Bank: _____; Account Number:_____).We confirm that you should only pay the above accounts receivable to the designated account,and only by paying the above

(**a**)

Figure 3. *Cont.*

```
<!DOCTYPE ctml>
    <ctml lang=zh_cn,link="exchangesheet.emd">>
    <<factor title@assignNotification>>Notice of Assignment of Accounts Receivable<</factor>>
        <<field@contractNumber>>No.:<@contractNumber}><</field>>
To:<<factor party@partyA>><<field partyA@name>><{partyA@name}><</field>><</factor>>(hereinafter referred to
as"you")From:<<factor party@partyB>><<field partyB@name>><partyB@name><</field>><</factor>>(hereinafter referred to as"we")
    For the needs of our business development,we have applied to transfer the following accounts receivable and all associated rights and
interests to<<field@factoringName>>,<{@factoringName}><</field>>(hereinafter referred to as the"commercial factoring
company").We hereby notify you:
1.  The following item(s)of accounts receivable and all associated rights and interests have been transferred to the commercial factoring
    company:
    1)All accounts receivable between us and you under<<field@assetRange>><{@aasetRange}><</field>>.
    2)All accounts receivable listed below from you to us:
    <<factor asset@relatedInformation>>
        <<property info>>
            <<field@assetNumber>>Item Number:<@assetNumber}><</field>>
            <<field@assetBuyer>>Debtor of Accounts Receivable:<{@assetBuyer}><</field>>
            <<field@assetBaseContract>>Basic Education Contract and Number:<@assetBaseContract}><</field>>
            <<field@documentName>>Document Name:<{@documentNamel><</field>>
            <<field@accountAmount%Money>>Amount of Accounts Receivable:<@accountAmount}><</field>>
            <<field@accountDue%Date>>Due Date of Accounts Receivable:<{@accountDue}><</field>>
        <</property>>
    <</factor>>
2.  Please pay the full amount of the above accounts receivable to the following account on the due date(
    <<field@aacountHolder>>Account Holder:<{@accountHolder}><</field>>;
    <<field@accountBank>>Bank:<{@accountBank}><</field>>>;
    <<field@accountAccount>>Account Number:<{@accountAccount}><</field>>).
    We confirm that you should only pay the above accounts receivable to the designated account,and only by paying the above accounts
    receivable to the designated account can you be released from your payment obligation.Without the written notification from the
    commercial factoring company(whether unilateral or jointly with us),the receiving account cannot be changed or revoked.
```

(**b**)

Figure 3. The illustrative example in English: (**a**) the factoring contract in English [33]; (**b**) the corresponding marked factoring contract with CTML.

6.2. An Example Contract Written in Chinese

We chose a factoring contract in Chinese from selected samples recommended by the China Service Trade Association. For example, the marking process of a Chinese factoring contract using CTML is similar to that of an English contract.

(1) Stage 1: Analyze Chinese grammar written in the factoring contract.

- Identify the natural language for writing contracts as Chinese and define grammar rules related to the text in CTML.
- Add encoding extensions for Chinese to existing rules.
- Extract the corresponding contract contents and define "< ctml lang=zh_cn > " in the document type description (See Table 3).

(2) Stage 2: According to CTML syntax rules and annotation methods, a module analysis is conducted on the contract text for semantic extraction, including six modules: contract title, parties, additional information, assets, terms, and contract formation. The annotation of the factoring contract in Chinese (shown in Figure 4a) is represented in Figure 4b.

(3) Stage 3: The hierarchical annotation method annotates the semantic part of the example layer by layer. The EMD containing the key-pair values is generated by marking the data negotiated in the contract with metadata.

(4) Stage 4: CTML contract automatically generates executable code after being mapped by a code generator.

- The SPESC code for the CTML contract is automatically generated by the code generator.
- Link and compile the code with EMD to generate a smart contract executable code and send it to the Smart Contract Library for contract deployment.

The experimental results show that our approach can automatically transform the same real-world contract written in different natural languages into a smart contract. This means that regardless of the language used to write the contract, our approach can be used to transform it into a smart contract. However, the code generators generated during the execution process are different. This is also very important for promoting standardization and interoperability of the contract.

《应收账款转让通知书》

编号：

致：_____（下称“贵方”）

自：_____（下称“我方”）

我方因经营发展的需要，已申请将下述应收账款及所从属的一切从权利和权益转让给_____（下称“商业保理人”），兹此通知贵方：

1、如下第____项应收账款及从属的一切从权利和权益转已转让给商业保理人：

(1) 我方与贵方之间_____项下的全部应收账款。

(2) 我方对贵方下表所列的全部应收账款：

序号	应收账款债务人	基础教育合同及编号	单据名称	单据号码	应收账款金额	应收账款到期日

2、 请贵方于应收账款到期日将上述应收账款金额足额支付到如下账户(开户人：_____;开户银行：_____; 账号：_____)。我方确认，贵方仅应向该指定账户支付上述应收账款、且只有向该指定账户支付上述应收账款才能解除贵方的付款义务。除商业保理人书面通知（无论单方还是联合我方共同书面通知）贵方外，该收款账户不得变更或撤销。

(**a**)

```
<!DOCTYPE ctml>
    <ctml lang=zh_cn,link="exchangesheet.emd">
    <<factor title@assignNotification>>应收账款转让通知书<</factor>>
        <<field@contractNumber>>编号：<{@contractNumber}><</field>>
    致：<<factor party@partyA>><<field partyA@name>><{partyA@name}><</field>>
    <</factor>>(下称“贵方”)
    自：<<factor party@partyB>><<field partyB@name>><{partyB@name}><</field>>
    <</factor>>(下称“我方”)
        我方因经营发展的需要，已申请将下述应收账款及所从属的一切从权利和权益转让给
        <<field@factoringName>><{@factoringName}>(下称“商业保理人”)，<</field>>
    兹此通知贵方：
1、如下应收账款及从属的一切从权利和权益转已转让给商业保理人：
    (1)我方与<{partyB@name}>之间<<field@assetRange>><{@assetRange}><</field>>项下的的全部应收账款以及该应收账款上所从属的一切从权利和权益。
    (2)我方对贵方下表所列的全部应收账款：
    <<factor asset@relatedInformation>>
    <<property info>>
        <<field@assetNumber>>序号：<{@assetNumber}><</field>>
        <<field@assetBuyer>>应收账款债务人：<{@assetBuyer}><</field>>
        <<field@assetBaseContract>>基础交易合同及编号：<{@assetBaseContract}><</field>>
        <<field@documentName>>单据名称：<{@documentName}><</field>>
        <<field@documentNumber>>单据号码：<{@documentNumber}><</field>>
        <<field@accountAmount%Money>>应收账款金额：<{@accountAmount}><</field>>
        <<field@accountDue%Date>>应收账款到期日：<{@accountDue}><</field>>
    <</property>>
    <</factor>>
2、请贵方于应收账款到期日将上述应收账款金额足额支付到如下账户(
        <<field@accountHolder>>开户人：<{@accountHolder}>;<</field>>
        <<field@accountBank>>开户银行：<{@accountBank}>;<</field>>
        <<field@accountAccount>>账号：<{@accountAccount}><</field>>)。我方确认，贵方仅应向该指定账户支付上述应收账款、且只有向该指定账户支付上述应收账款才能解除贵方的付款义务。除商业保理人书面通知(无论单方还是联合我方共同书面通知)贵方外，该收款账户不得变更或撤销。
    <<factor genTerm@affordTerm>>
```

(**b**)

Figure 4. The illustrative example in Chinese: (**a**) the factoring contract in Chinese [33]; (**b**) the corresponding marked factoring contract with CTML.

6.3. Comparative Analysis

This approach minimizes semantic loss and ensures logical equivalence through a two-layer semantic anchoring mechanism:

- In the contract abstraction, the approach does not perform a black-box, fully automated translation. Instead, users with legal backgrounds explicitly anchor core clauses (such as payment conditions and default triggers) using CTML. This "human-in-the-loop" model ensures the accurate extraction of key legal intent from natural language.
- In the code generation, CTML employs a deterministic grammar mapping based on Xtext to transform the marked legal logic 1:1 into DSL, thus technically avoiding semantic drift caused by probabilistic models. Developers conduct pre-deployment audits of the generated code. Although the legal enforceability of smart contracts ultimately depends on the interpretation of a specific jurisdiction, CTML provides a robust structural guarantee for the semantic equivalence between legal text and executable code by generating logically consistent code.

CTML's design logic is analogous to HTML or XML. While different natural languages (such as Chinese, English, and French) require different syntax parsing rules (Stage 1), this belongs to the "configuration layer" rather than the "architectural layer". Once the tagging is complete, the core transformation logic (Stage 2) is completely universal. Our claim of "supporting any language" refers to the extensibility of its architecture, meaning adaptation can be achieved by adding language-specific configuration files.

Currently, the main boundary condition lies in the ambiguity of legal semantics (such as reasonable compensation). We use manual annotation to lock these ambiguities within specific CTML tags. While this increases the annotation burden, it ensures the absolute rigor of the legal logic. Future automation (such as the introduction of LLM) will be based on this structured foundation, thus establishing an evolutionary path from manual to fully automated.

Although this paper selects a limited number of cases, we have deliberately chosen complex contract types that cover nested logic, multi-party games, and temporal constraints. These cases are representative in terms of logical topology. Furthermore, since CTML employs a rule-based linear parsing algorithm with a computational complexity of $O(n)$, this means that the system's performance will remain linearly stable when processing longer and more contracts, demonstrating theoretical scalability. In addition, we verified its preliminary cross-language feasibility through bilingual (Chinese and English) cases. While the illustrative case studies in English and Chinese validate the feasibility of the CTML-based transformation, we acknowledge the limitations in terms of sectoral diversity. However, the choice of these two distinct languages demonstrates the framework's capability to handle significant linguistic variances. The modular architecture of CTML allows for the integration of domain-specific ontologies, providing a foundation for future scalability. Subsequent research will focus on stress-testing the framework with a broader corpus of complex industrial contracts to further refine its general applicability.

Furthermore, this approach fully considers the underlying constraints of blockchain execution when designing the CTML: Regarding gas costs and performance, CTML only transforms the execution logic in the contract (such as payment triggering and conditional judgments) into code, while keeping non-executable clauses off-chain, thus ensuring that the generated Solidity code has extremely high computational efficiency and low gas consumption. In terms of security, since the code is automatically generated based on predefined syntax rules, it effectively avoids common syntax-level vulnerabilities such as overflows and reentrancy encountered during manual coding. And to address blockchain-specific limitations, the system has built-in adapters for mainstream platforms such as

Ethereum, ensuring that the logic meets the deterministic requirements of on-chain execution through "compile-time checks."

7. Advantages and Limitations

7.1. Advantages of Our Approach

Legal contracts have undergone a significant transformation, transitioning from manual processing to smart contracts and, more recently, smart legal contracts. This evolution aims to improve efficiency and accuracy. Here's how CTML contributes to this process:

- Manual processing stage: Traditionally, legal contracts were painstakingly converted by hand into executable programs. This approach lacked verification tools, making it difficult to ensure the program accurately reflected the original contract's meaning.
- Smart contract stage: The introduction of smart contracts marked a turning point, enabling automated contract execution. However, challenges such as poor readability, slow development, and inefficient conversion between legal and smart contracts limited their wider adoption in legal settings.
- Smart legal contract stage: Smart legal contracts, with their improved readability, addressed the learning curve associated with traditional smart contracts and bridged the gap between legal and programmatic language. However, the disconnect between smart legal contracts and their natural language counterparts hindered further development.
- Legal code stage (or code-legalization stage): This stage introduces CTML. Here's how it works:
  - Legal contracts in natural language are tagged with CTML, essentially adding structure and instructions.
  - These CTML contracts are then transformed into smart legal contracts.
  - Finally, these smart legal contracts are compiled into executable code.

  This process allows for a complete transformation from a natural language contract to executable code, ensuring the generated program aligns perfectly with the legal intent of the marked-up contract.

In essence, CTML bridges the gap between legal contracts and smart contracts, fostering a more efficient, accurate, and legally-sound approach to contract management. After generating the smart contract executable code, we integrate the CTML contracts, EMD, and DSL programs into the blockchain smart contract library to enable real-time interaction between contract parties. We present an online smart contract library integrating CTML auxiliary annotation and SLC contract generation functions. The specific related research about the library is in the previous paper [9].

We illustrate the functional performance of the four smart contract transform approaches in terms of legal applicability, as shown in Table 5 below. Analysis shows that while formal and template-based methods offer high accuracy, their coverage is limited by predefined templates or symbols, making them ill-suited for the complexities of multilingual contracts. NLP/LLM-based approaches, while capable of handling multiple languages, exhibit instability in transparency and error rates. In contrast, the CTML framework proposed in this paper, by introducing deterministic semantic markup, significantly improves the controllability and transparency of the transformation process while maintaining high coverage, effectively addressing the shortcomings of existing technologies.

Table 5. Qualitative functional comparative analysis of transformation approaches.

Metrics	Formal Methods	Template-Based	NLP/LLM-Based	Our Approach
Accuracy	High (Limited to predefined models)	High (Strict adherence to rules)	Variable (Susceptible to hallucinations)	High (Deterministic mapping)
Transparency	High (Visual logical flow)	High (Explicit transformation rules)	Low (Opaque "black-box" processing)	High (1:1 semantic-to-code traceability)
Coverage	Low (Limited by graphical symbols)	Medium (Dependent on template library)	High (Broad natural language support)	High (Extensible markup grammar)
Error Rates	Low (Primarily manual modeling errors)	Low (Errors due to template mismatch)	High/Unpredictable (Stochastic outputs)	Minimal (Controlled via syntax validation)

Automatically transforming real-world contracts into smart contracts has the following benefits:

- Accuracy: The execution of smart contracts is based on predefined logic and conditions, thus ensuring the accuracy of the contract. This reduces the possibility of misunderstandings and disputes.
- Decentralization: The execution of smart contracts does not rely on centralized institutions or third-party trust. This increases the security and stability of the contract.
- Transparency and verifiability: The execution process of smart contracts is public, and all participants can check and verify the contract results. This provides higher transparency and trust.
- Tampability: Smart contracts cannot be modified or deleted once deployed on the blockchain. This ensures the integrity and immutability of the contract.
- Reducing intermediary costs: The automated execution of smart contracts reduces the involvement of intermediary agencies, thereby lowering transaction and contract execution costs.

In summary, transforming real-world contracts in any natural language into smart contracts can bring higher efficiency, accuracy, transparency, and security while reducing intermediary costs. This makes contract execution more reliable and efficient and expands the application areas of contracts. While the overall framework is general, the underlying implementation components differ: (1) General parts: CTML's syntax design principles (such as nested structures and semantic markup formats) and metadata extraction logic (EMD data table generation) are universal. (2) Custom parts: The compilers and code generators generated for different languages are different.

Our approach can automatically transform real-world contracts written in different natural languages into smart contracts. This approach has enormous potential for application, especially in cross-border transactions and supply chain management. For example, different countries may use different languages in international trade to write real-world contracts. By using our approach to transform these contracts into smart contracts, we can achieve global contract execution and payment and ensure timely and accurate transfer of funds.

However, it should be noted that the effectiveness and reliability of this approach still require more extensive experimentation and validation. The differences between different natural languages are very complex, involving some factors such as grammar, vocabulary, and culture. Therefore, automatically transforming real-world contracts in different natural languages into smart contracts is challenging. Further research and improvement methods are needed to enhance the accuracy and applicability of the transformation.

7.2. Limitations and Future Work

To address the limitations of this approach, the focus should be on transforming current challenges (such as labor costs, grammatical complexity) into future research directions (such as automation, generality). We summarize these as follows:

(1) Annotation burden: Currently, CTML relies on manual or semi-manual labeling (Stage 1), which presents a steep learning curve for non-professionals. While CTML provides a structured way to bridge natural language and code, the annotation burden remains a significant boundary condition. Currently, the precision of the generated smart contract heavily relies on the quality of manual markup. This process requires users to have a deep understanding of both legal logic and CTML syntax. To address this, future research will focus on integrating Large Language Models (LLMs) to automate the initial markup phase, transforming raw contract text into CTML-ready formats with minimal human intervention.

(2) Limitations of domain-specific syntax: Contracts across different industries (such as insurance, supply chain, and intellectual property) have vastly different syntax structures, and current syntax rules may not be sufficient to cover all vertical sectors. Our current framework is optimized for standard commercial agreements. However, contracts in specialized fields—such as insurance derivatives or intellectual property licensing—often contain unique logic structures that may require custom grammar extensions or specialized generators. Future iterations of CTML aim to develop a modular grammar library, allowing users to plug in domain-specific rule sets (e.g., a Supply Chain Module) to enhance the framework's versatility across different industrial sectors.

(3) Semantic complexity and boundary conditions: Ambiguous terms in legal texts (such as "within a reasonable time") are difficult to directly translate into rigid code logic. CTML encounters boundary conditions when dealing with semantic ambiguity inherent in natural language (e.g., reasonable efforts). These terms are difficult to formalize into deterministic smart contract code. We plan to explore hybrid execution models where CTML-generated contracts can interact with decentralized oracles or human-in-the-loop consensus mechanisms to resolve subjective legal predicates.

(4) Robustness: For the ambiguity clauses, CTML does not rely on probabilistic automatic parsing but instead uses human experts to lock nondeterministic semantics into deterministic logical labels. For exceptional conditions and complex legal structures, the approach utilizes nested hierarchical syntax rules to fully capture logical branches. For incomplete contracts, the built-in compilation and verification engine can effectively identify missing key elements and trigger semantic error warnings, thereby forcibly avoiding potential logical risks in smart contract execution at the architectural level. While the current framework ensures robustness through expert-led markup and strict grammar validation, future iterations could integrate Formal Verification methods to mathematically prove the consistency between the marked-up contract and the generated bytecode, especially for mission-critical legal constructs.

In summary, the transition from manual, domain-specific intervention to a fully autonomous and generalized framework represents the core trajectory of our future work. By addressing the current annotation burden and expanding our grammar modules, we aim to make CTML a universal standard for legal-to-code transformation.

8. Conclusions

This study addresses the challenges of linguistic structural differences, complex semantic understanding, and low automation in the transformation of real-world contracts

to blockchain smart contracts. It proposes an innovative transformation framework based on Contract Text Markup Language (CTML). This approach can automatically transform smart contracts from real-world contracts in any natural language. Through a four-stage transformation process (definition of grammar rules, contract abstract markup, extract metadata markup and automated code generation), CTML overcomes the dependence of traditional methods on specific languages and the logical inconsistencies inherent in pure AI generation. It achieves a universal transformation mechanism that combines deterministic semantics, high automation, and multilingual applicability, providing a standardized intermediate abstraction layer for the reliable mapping of cross-language legal agreements to smart contracts. Our research demonstrates that the proposed approach reliably facilitates the transformation of complex multilingual contracts into high-fidelity Solidity code, providing a scalable and verifiable bridge between legal intent and blockchain execution.

Author Contributions: Conceptualization, C.E.C.; Methodology, C.E.C., B.L. and Y.Z.; Writing—original draft, C.E.C. and X.L.; Writing—review & editing, C.E.C., X.L., L.J., and T.W.; Validation, X.L., L.J. and B.L.; Supervision, B.L. and Y.Z.; Funding acquisition, C.E.C. and T.W. All authors have read and agreed to the published version of the manuscript.

Funding: This work was supported in part by Advanced Materials-National Science and Technology Major Project (Grant No. 2025ZD0620000), Guangdong Basic and Applied Basic Research Foundation (Grant No. 2023A1515110083), Fundamental Research Funds for the Central Universities (Grant No. FRF-BD-25-021), Key Laboratory of Industrial Data Security and Protection Testing, Ministry of Industry and Information Technology Open Research Project (Grant No. IDPSA2025-004), and partially sponsored by CAAI-CANN Open Fund, developed on OpenI Community.

Data Availability Statement: Data are contained within the article.

Conflicts of Interest: Author Limin Jia was employed by the company Zhejiang Neptune Technology Co., Ltd. The remaining authors declare that the research was conducted in the absence of any commercial or financial relationships that could be construed as a potential conflict of interest.

References

1. Yaqoob, I.; Salah, K.; Uddin, M.; Jayaraman, R.; Omar, M.; Imran, M. Blockchain for digital twins: Recent advances and future research challenges. *IEEE Netw.* **2020**, *34*, 290–298. [CrossRef]
2. Hunhevicz, J.J.; Motie, M.; Hall, D.M. Digital building twins and blockchain for performance-based (smart) contracts. *Autom. Constr.* **2022**, *133*, 103981. [CrossRef]
3. Ma, J.; Huang, R. Digital explosions and digital clones. In Proceedings of the 2015 IEEE 12th Intl Conf on Ubiquitous Intelligence and Computing, Beijing, China, 10–14 August 2015; pp. 1133–1138.
4. Savelyev, A. Contract law 2.0: 'Smart' contracts as the beginning of the end of classic contract law. *Inf. Commun. Technol. Law* **2017**, *26*, 116–134. [CrossRef]
5. Chowdhary, K.R. Natural language processing. In *Fundamentals of Artificial Intelligence*; Springer: New Delhi, India, 2020; pp. 603–649.
6. Dixit, A.; Deval, V.; Dwivedi, V.; Norta, A.; Draheim, D. Towards user-centered and legally relevant smart-contract development: A systematic literature review. *J. Ind. Inf. Integr.* **2022**, *26*, 100314. [CrossRef]
7. Rasti, A.; Anda, A.A.; Alfuhaid, S.; Parvizimosaed, A.; Amyot, D.; Roveri, M.; Logrippo, L.; Mylopoulos, J. Automated generation of smart contract code from legal contract specifications with Symboleo2sc. *Softw. Syst. Model.* **2025**, *24*, 1127–1156. [CrossRef]
8. Ait Hsain, Y.; Laaz, N.; Mbarki, S. SCEditor-Web: Bridging Model-Driven engineering and generative AI for smart contract development. *Information* **2025**, *16*, 870. [CrossRef]
9. Fan, Y.; Chen, E.; Zhu, Y.; He, X.; Yau, S.S.; Pandya, K. Automatic Generation of Smart Contracts from Real-World Contracts in Natural Language. In Proceedings of 2023 IEEE Smart World Congress (IEEE SWC 2023), Portsmouth, UK, 28–31 August 2023; pp. 1–8.
10. Hamdaqa, M.; Met, L.A.P.; Qasse, I. iContractML 2.0: A domain-specific language for modeling and deploying smart contracts onto multiple blockchain platforms. *Inf. Softw. Technol.* **2022**, *144*, 106762. [CrossRef]

11. Frantz, C.K.; Nowostawski, M. From institutions to code: Towards automated generation of smart contracts. In Proceedings of the 2016 IEEE 1st International Workshops on Foundations and Applications of Self* Systems (FAS* W), Augsburg, Germany, 12–16 September 2016; pp. 210–215.
12. Choudhury, O.; Rudolph, N.; Sylla, I.; Fairoza, N.; Das, A. Auto-generation of smart contracts from domain-specific ontologies and semantic rules. In Proceedings of the 2018 IEEE International Conference on Internet of Things (iThings), Halifax, NS, Canada, 30 July–3 August 2018; pp. 963–970.
13. Jurgelaitis, M.; Butkienė, R. Solidity code generation from UML state machines in model-driven smart contract development. *IEEE Access* **2022**, *10*, 33465–33481. [CrossRef]
14. Tateishi, T.; Yoshihama, S.; Sato, N.; Saito, S. Automatic smart contract generation using controlled natural language and template. *IBM J. Res. Dev.* **2019**, *63*, 6:1–6:12. [CrossRef]
15. Chen, E.; Qin, B.; Zhu, Y.; Song, W.; Wang, S.; Chu, C.C.W.; Yau, S.S. SPESC-Translator: Towards automatically smart legal contract conversion for blockchain-based auction services. *IEEE Trans. Serv. Comput.* **2021**, *15*, 3061–3076. [CrossRef]
16. He, X.; Qin, B.; Zhu, Y.; Chen, X.; Liu, Y. SPESC: A specification language for smart contracts. In Proceedings of 2018 IEEE 42nd Annual Computer Software and Applications Conference (COMPSAC), Tokyo, Japan, 23–27 July 2018; Volume 1, pp. 132–137.
17. SPESC. A Specification Language for Smart Contracts. Available online: https://github.com/USTB-InternetSecurityLab/SPESC (accessed on 31 January 2023).
18. Shi, C.; Xiang, Y.; Yu, J.; Sood, K.; Gao, L. Machine translation-based fine-grained comments generation for solidity smart contracts. *Inf. Softw. Technol.* **2023**, *153*, 107065. [CrossRef]
19. Aejas, B.; Belhi, A.; Bouras, A. Smart Contracts Auto-generation for Supply Chain Contexts. In *Proceedings of IFIP International Conference on Product Lifecycle Management*; Springer: Cham, Switzerland, 2022; pp. 347–357.
20. Fang, P.; Zou, Z.; Xiao, X.; Liu, Z. isyn: Semi-automated smart contract synthesis from legal financial agreements. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*; Association for Computing Machinery: New York, NY, USA, 2023; pp. 727–739.
21. Ahmed, S.U.; Danish, A.; Ahmad, N.; Ahmad, T. Smart contract generation through NLP and blockchain for legal documents. *Procedia Comput. Sci.* **2024**, *235*, 2529–2537. [CrossRef]
22. Tong, Y.; Tan, W.; Guo, J.; Shen, B.; Qin, P.; Zhuo, S. Smart contract generation assisted by AI-based word segmentation. *Appl. Sci.* **2022**, *12*, 4773. [CrossRef]
23. Napoli, E.A.; Barbàra, F.; Gatteschi, V.; Schifanella, C. Leveraging Large Language Models for Automatic Smart Contract Generation. In Proceedings of the 2024 IEEE 48th Annual Computers, Software, and Applications Conference (COMPSAC), Osaka, Japan, 2–4 July 2024; pp. 701–710.
24. Santos, H.V.B.; Januario, R.R.S.; Zanatta, R.C.; Matos, S.N.; Ueyama, J. Comparative Analysis of Smart Contract Generation Using Large Language Models. In *Anais do XLIII Simpósio Brasileiro de Redes de Computadores e Sistemas Distribuídos (SBRC)*; Sociedade Brasileira de Computação (SBC): Porto Alegre, Brazil, 2025; pp. 112–125.
25. Barbàra, F.; Napoli, E.A.; Gatteschi, V.; Schifanella, C. Automatic Smart Contract Generation Through LLMs: When The Stochastic Parrot Fails. In Proceedings of the 6th Distributed Ledger Technology Workshop (DLT), Turin, Italy, 24–25 May 2024.
26. Eysholdt, M.; Behrens, H. Xtext: Implement your language faster than the quick and dirty way. In Proceedings of the ACM International Conference Companion on Object Oriented Programming Systems Languages and Applications Companion, Reno/Tahoe, NV, USA, 17–20 October 2010; pp. 307–309.
27. Mernik, M.; Heering, J.; Sloane, A.M. When and how to develop domain-specific languages. *ACM Comput. Surv. (Csur)* **2005**, *37*, 316–344. [CrossRef]
28. *ISO/IEC 14977:1996*; Information Technology-Syntactic Metalanguage-Extended BNF. ISO/IEC Internationl Standard: Geneva, Switzerland, 1996.
29. Wile, D.S. Supporting the DSL spectrum. *J. Comput. Inf. Technol.* **2001**, *9*, 263–287. [CrossRef]
30. Xtext. Available online: https://www.eclipse.org/Xtext/ (accessed on 22 November 2022).
31. Bettini, L. *Implementing Domain Specific Languages with Xtext and Xtend*, 2nd ed.; Packt Publishing: Birmingham, UK, 2016.
32. Parekh, S. Understanding Factoring Contracts and How They Work. Available online: https://www.dripcapital.com/en-in/resources/finance-guides/factoring-agreement (accessed on 22 November 2022).
33. Standard. Commercial Factoring Contract Guidelines (Appendix). T/CATIS 003-2020. Available online: http://cfi.cfiservice.com/view.php?aid=2174 (accessed on 1 January 2021).

Disclaimer/Publisher's Note: The statements, opinions and data contained in all publications are solely those of the individual author(s) and contributor(s) and not of MDPI and/or the editor(s). MDPI and/or the editor(s) disclaim responsibility for any injury to people or property resulting from any ideas, methods, instructions or products referred to in the content.