# iSyn: Semi-automated Smart Contract Synthesis from Legal Financial Agreements

Pengcheng Fang
Department of Computer and Data Sciences
Case Western Reserve University
United States of America
pxf109@case.edu

Zhenhua Zou
Institute for Network Sciences and Cyberspace
Tsinghua University
China
zou-zh21@mails.tsinghua.edu.cn

Xusheng Xiao
School of Computing and Augmented Intelligence
Arizona State University
United States of America
xusheng.xiao@asu.edu

Zhuotao Liu
Institute for Network Sciences and Cyberspace
Tsinghua University
China
zhuotaoliu@tsinghua.edu.cn

## ABSTRACT

Embracing software-driven smart contracts to fulfill legal agreements is a promising direction for digital transformation in the legal sector. Existing solutions mostly consider smart contracts as simple add-ons, without leveraging the programmability of smart contracts to realize complex semantics of legal agreements. In this paper, we propose iSyn, the first end-to-end system that synthesizes smart contracts to fulfill the semantics of financial legal agreements, with minimal human interventions. The design of iSyn centers around a novel intermediate representation (SmartIR) that closes the gap between the natural language sentences and smart contract statements. Specifically, iSyn includes a synergistic pipeline that unifies multiple NLP-techniques to accurately construct SmartIR instances given legal agreements, and performs template-based synthesis based on the SmartIR instances to synthesize smart contracts. We also design a validation framework to verify the correctness and detect known vulnerabilities of the synthesized smart contracts. We evaluate iSyn using legal agreements centering around financial transactions. The results show that iSyn-synthesized smart contracts are syntactically similar and semantically correct (or within a few edits), compared with the "ground truth" smart contracts manually developed by inspecting the legal agreements.

## CCS CONCEPTS

• **Software and its engineering → Source code generation**; • **Computing methodologies → Natural language processing**.

## KEYWORDS

Smart Contracts; Program Synthesis; Natural Language Processing

**ACM Reference Format:**
Pengcheng Fang, Zhenhua Zou, Xusheng Xiao, and Zhuotao Liu. 2023. iSyn: Semi-automated Smart Contract Synthesis from Legal Financial Agreements. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '23), July 17–21, 2023, Seattle, WA, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3597926.3598091

## 1 INTRODUCTION

The ever-growing digital transformation has shaped virtually every type of business, such as online courses, online conferencing, online medical and pharmaceutical systems, remote work forces, and so on. Similarly, the legal sector is experiencing online transformation. For example, DocuSign [32], a U.S. headquartered digital signature company, now has over 85 million users worldwide. Meanwhile, the discussion on whether smart contracts executed on blockchains (or in general software code) can be treated as legally binding contracts began several years ago [24, 25, 38, 54, 69]. Now the answer starts to become clearer as some courts recently confirmed that electronic data stored on blockchains met the authenticity and integrity requirements of electronic evidence [72].

Despite the promise of smart contracts in operating legal agreements, it is challenging for developers to write smart contracts that capture the core logic of legal agreements. Legal agreements are (i) large in size, often consisting of hundreds or even thousands of sentences, and (ii) only a portion of the sentences that describe the core semantics (*e.g.,* financial transactions) are valuable to be recorded or executed on blockchains. Thus, it is non-scalable and error-prone to manually inspect these legal agreements to identify blockchain-friendly sentences and subsequently converting them into smart contract statements.

Existing research that extracts formal representations from software documents [49, 51, 61, 73, 77] are not effective in processing legal agreements because, unlike software documents (*e.g.,* use cases and API documents), legal agreements do not follow very specific styles or presentations. The community also proposes several preliminary protocols [2, 4] to augment traditional legal agreements with smart contracts, by embedding an *existing* smart contract on Ethereum into a specific section of the legal agreement that is written using predefined templates and markup language (for instance, replacing the traditional payment section in an offer letter agreement with a smart contract that periodically pays Ether to the employee). However, these solutions treat smart contracts as *simple and external add-ons*, while the key semantics in legal contracts remain undigitized.

In this work, we push the boundaries of legal agreement digitization by inventing a semi-automated framework iSyn that can synthesize smart contracts to represent the core semantics in legal agreements. To make the problem tangible, we first refine the scope of legal agreements to focus on those centering around *financial transactions*. Our empirical study also confirms that financial agreements account for over 80% of the legal agreements in top-10 most popular categories on Law Insider [33]. Second, because ambiguity does exist in legal agreements (either due to natural languages or even by design), we position iSyn as a *semi-automated* synthesis framework that can reduce the otherwise significant developmental efforts to simple selections.

We recognize several key challenges in realizing iSyn:
*Challenge* ①: due to the semantic gap between the legal agreements written in natural languages and the smart contracts implemented as software code, the search space of mapping hundreds or even thousands of sentences in the legal agreements to various types of statements in smart contracts is enormous. While natural language processing (NLP) techniques [15, 41, 75] could be applied to refine the scope of the sentences to be translated into program statements, it is still far from feasible to solve the sentence-to-statement-mapping problem by directly adopting existing program synthesis techniques [14, 30, 74], which merely handle one or two sentences and synthesize expressions with limited parameters for SQL or other domain specific languages (DSLs).
*Challenge* ②: the second challenge is how to populate variable names and conditions in the statements based on the entities in the sentence and other relevant sentences. While existing named entity recognition (NER) techniques [31, 55, 67, 81] could be applied to extract entities from sentences, they are unable to fill the statements with proper variables or/and conditions given these entities.
*Challenge* ③: the rich scenarios expressed in the legal agreements pose another category of challenges. In particular, a legal agreement describes not only financial transactions that should be executed when certain conditions are met, but also describes various abnormal situations (such as delay of deliveries and violation of commitments) that often result in penalties or liability terminations. Existing program synthesis designs [14, 30, 74] are limited in expressing different contexts for program executions and the statically generated programs cannot represent dynamic scenarios.

To overcome these challenges, iSyn invents three innovative designs. First, we propose a novel intermediate representation (IR) design, SmartIR, to bridge the gap between the natural language sentences and smart contract statements. On the one hand, SmartIR abstracts four types of intents that define the core transaction logic in legal agreements. This abstraction, obtained after we reviewed over 800 legal agreements with 1, 353, 843 sentences, is completely *data-driven*. On the other head, SmartIR is underpinned by formal grammar rules such that one SmartIR item (representing one financial transaction) is mapped to only a fixed set of software statements in smart contracts. *Each SmartIR item also contains one or multiple slots to be filled with the critical terms or actions extracted from legal agreements*, such as payment entities and effective time. Therefore, SmartIR drastically refines the search space for the complicated sentence-to-statement mapping problem (addressing challenge ①).

Second, we design a novel NLP pipeline to populate the SmartIR slots based on the entities and conditions extracted from legal agreements. In particular, we adapt multiple NLP related techniques (including synonym [44], Part-Of-Speech (POS) tag [23], Named Entity Recognition (NER) [31, 55, 67, 81], question-answering [19, 41]) in our synergistic pipeline to perform accurate intent classification and slot filling to instantiate SmartIR instances. Afterwards, we employ a template-driven smart contract synthesis to output concrete, blockchain-deployable smart contracts given these SmartIR instances (addressing challenge ②).

Third, we design a validation framework to verify the correctness of iSyn-synthesized smart contracts in representing various situations defined in the legal agreements. To construct the validation cases, we exercise the conditions expressed by the SmartIR instances, and performs exhaustive enumerations of feasible conditions. By executing the synthesized smart contracts on these test inputs, iSyn confirms that the contracts correctly represent the dynamic semantics originally defined in the legal agreements. The validation framework also includes an oracle contract to connect the iSyn-synthesized smart contracts with key offchain data sources (addressing challenge ③).

**Evaluations.** We conduct extensive evaluations on 86 representative legal agreements (with 129, 907 sentences) chosen from 10 most popular legal agreement categories. Overall, the synthesized smart contracts have on average 179 statements, and these synthesized smart contracts are both syntactically and semantically similar to the ground truth smart contracts developed by directly examining the legal agreements. Our results show that the syntactic similarities between the synthesized and ground truth smart contracts are above 0.99. Meanwhile, the average percentage of semantically correct functions in the synthesized contracts is ~90%, where the incorrectly synthesized smart contracts can be fixed with median two edits. That is, iSyn converts the efforts in writing a 179-LoC smart contract from a legal agreement to merely two edits on the synthesized smart contract and one choice of payment entities to resolve ambiguities.

In summary, this paper makes the following major contributions:

- A novel approach, iSyn, that synthesizes smart contracts from legal agreements written in natural language, with minimum human intervention required to settle the ambiguities existing in legal agreements.
- An empirical study on 8, 029 legal agreements (1.35 million sentences) to motivate the design of iSyn.
- A novel IR, SmartIR, to bridge the gap between the semantics in legal agreements and smart contract statements, and a corresponding NLP pipeline to populate SmartIR slots.
- A novel validation framework to verify the synthesized smart contracts in representing various situations described in legal agreements.
- An evaluation on a diversified set of legal agreements to demonstrate the effectiveness of iSyn. *We release the first large-scale dataset on financial agreements with labels on their core transaction logic in our project repository [3].*

Table 1: Empirical study results of the top-10 most popular legal agreement categories on Law Insider

| Category | Number | Sentences | Intents | | | | Other | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| | | | Entity | Online. | Offline. | Termination | Formats | Disclaimer | Terminology | Misc. |
| A | 224 | 41219 | 10.07% | 4.79% | 7.85% | 2.73% | 21.65% | 14.53% | 5.35% | 33.03% |
| CC | 235 | 164829 | 5.14% | 6.44% | 10.41% | 2.92% | 25.33% | 20.28% | 6.51% | 22.97% |
| EA | 781 | 49096 | 2.59% | 10.75% | 7.50% | 36.43% | 6.57% | 13.55% | 3.90% | 18.73% |
| IC | 581 | 163071 | 4.75% | 5.76% | 12.71% | 2.82% | 26.49% | 18.13% | 5.02% | 24.32% |
| JFA | 1431 | 27103 | 16.73% | 0.68% | 2.88% | 1.74% | 58.60% | 5.70% | 0.60% | 13.06% |
| PMA | 1122 | 318799 | 2.47% | 2.30% | 7.91% | 3.95% | 22.77% | 20.79% | 8.20% | 31.61% |
| RRC | 1299 | 162329 | 5.85% | 2.26% | 11.09% | 2.58% | 24.84% | 18.31% | 7.20% | 27.88% |
| SECPA | 283 | 131331 | 3.46% | 6.80% | 13.34% | 19.68% | 6.05% | 20.09% | 6.29% | 24.29% |
| TA | 578 | 64995 | 5.71% | 6.36% | 11.35% | 3.08% | 22.33% | 17.11% | 6.56% | 27.50% |
| UA | 1495 | 231071 | 23.10% | 23.56% | 28.11% | 5.38% | 4.07% | 4.13% | 9.32% | 2.33% |
| **Total/Avg** | 8029 | 1353843 | 7.99% | 6.97% | 11.32% | 8.13% | 21.87% | 15.26% | 5.89% | 22.57% |

## 2 BACKGROUND AND MOTIVATION STUDY

### 2.1 Blockchain and Smart Contracts

Blockchain [45] lets mutually untrusted parties run a consensus protocol to agree on the trading transactions and maintain a shared ledger of data. Besides enabling cryptocurrencies (*e.g.,* Bitcoin), blockchain supports decentralized execution of general-purpose programs, called smart contracts [6]. These smart contracts are written in turing-complete programming languages such as Solidity [5] in Ethereum [18], and its correct executions are enforced by using the blockchain's consensus protocol. Thus, the programmability and the security of smart contracts powers a wide range of decentralized applications (dApps).

### 2.2 Empirical Study of Legal Agreements

To understand the core semantics logic in legal agreements, we conduct an empirical study on the top-10 most popular categories of legal agreements collected from the Law Insider [33]. Our study subjects include 8029 legal agreements (with over 1.35 million sentences) from the following ten categories: agreement (A), credit contract (CC), employment agreement (EA), indenture contract (IC), joint filling agreement (JFA), plan of merger agreement (PMA), registration right agreement (RRA), security purchase agreement (SECPA), trust agreement (TA) and underwriting agreement (UA).

We find that that 82.17% of these legal agreements (9 out of the 10 studied categories) center around *financial transactions*. The pervasiveness of financial agreements fundamentally motivates us to embrace blockchains (or more precisely smart contracts) in our design, which are experiencing rapid adoption in the finance sector (such as the decentralized finance, DeFi). Further, we identify four types of sentences that are essential to describe financial transactions. Following the NLP community convention, we use four intents, *OnlineStateTransfer*, *Entity*, *OfflineDelivery*, and *Termination*, to represent these four types of sentences. In particular, the core transactional logic can be described via a set of *OnlineStateTransfer* sentences, where one *OnlineStateTransfer* sentence may be associated with certain properties or/and conditions, including the relevant entities (*Entity*), offline deliveries of physical goods (*OfflineDelivery*), and termination conditions (*Termination*).

We train a classification model of these intents using our curated benchmark (§ 4.4) based on over 800 labeled legal agreements. We then apply the model on these legal agreements and compute the sentence distribution of different intents, as shown in Table 1. The results show that on average the sentences classified as one of the four intent types account for a significant portion (35.78%) of sentences in a financial agreement. We also reviewed the remaining sentences (classified as "Other" in Table 1) to understand how they may contribute to financial transactions. We find out that these sentences mainly describe formats (*i.e.,* headers/titles), disclaimers (*e.g.,* "This Agreement will be governed by ... the State of Nevada"), terminologies, and legal miscellaneousness, which are difficult to be represented as software code. *Thus, it is not our goal to completely replace the legal agreements using iSyn.* Instead, iSyn focuses on representing the core transactional logic of the financial agreements as smart contracts, hoping to reduce the commercial frictions and transactional costs while operating these financial agreements.

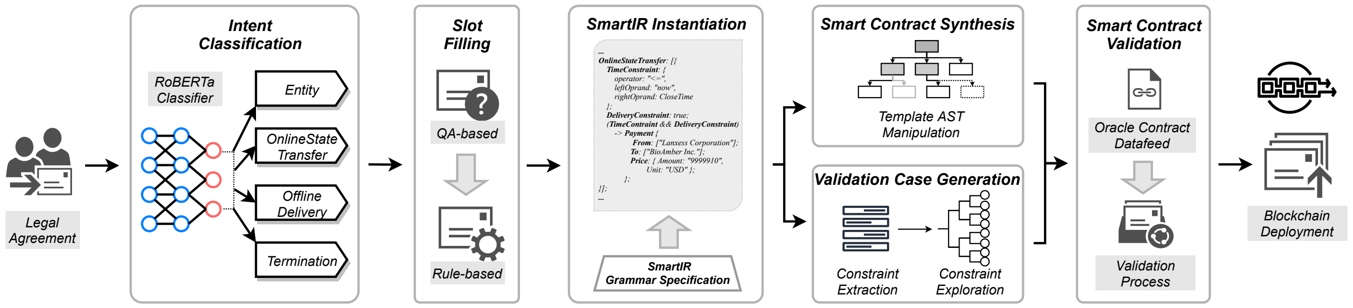
## 3 DESIGN OF ISYN

### 3.1 Overview

Figure 1 shows the overall architecture of iSyn. The design of iSyn centers around SmartIR, a key middleground bridging the semantic gap between legal agreements and smart contracts. Instead of manually examining legal agreements, iSyn constructs SmartIR instances by automatically extracting the core logic from the legal agreements, using a holistic pipeline that consolidates multiple NLP techniques. This synergistic approach achieves accurate intent classification and slot filling for SmartIR instantiation, and outperforms these individual NLP designs when applied alone. Given these SmartIR instances, iSyn proposes a template-driven smart contract synthesis to output *predictable* blockchain-deployable smart contracts. This allows us to build a generic validation framework that can enumerate the branches of the synthesized smart contract with a comprehensive set of validation cases. This not only ensures the correctness of the contract, but also can provide simulated results to showcase various termination cases when necessary.

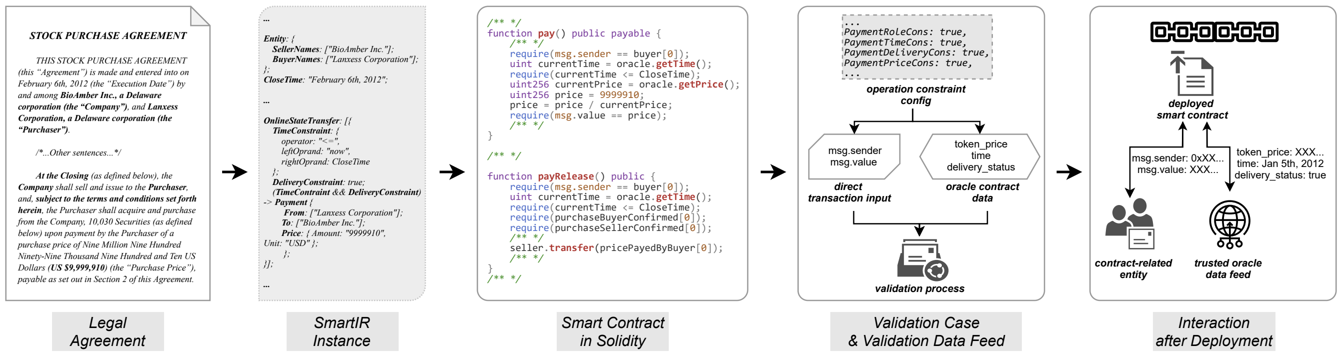
### 3.2 An End-to-End Example

Figure 2 shows the overview of iSyn using an end-to-end example.
**Intent Definition.** Financial agreements written by legal professionals are not originally meant to be translated into software code. Thus, the very first step in iSyn is to decide the proper programming model of legal agreements. We abstract four types of intents in our model to represent the core transaction logic in the legal



```
...
OnlineStateTransfer: [{
  TimeConstraint: {
    operator: "<=",
    leftOprand: "now",
    rightOprand: CloseTime
  };
  DeliveryConstraint: true;
  (TimeContraint && DeliveryConstraint)
    -> Payment {
         From: ["Lanxess Corporation"];
         To: ["BioAmber Inc."];
         Price: { Amount: "9999910",
                  Unit: "USD" };
       };
}];
...
```

**Figure 1: The architecture of iSyn. iSyn first classifies and identifies four types of transaction intents from the legal agreements. To specify these intents, it then uses a dedicated slot filling approach to instantiate SmartIR. Afterwards, iSyn synthesizes blockchain-executable smart contract by tuning the abstract syntax tree (AST) of a smart contract template based on the given SmartIR instance. At the same time, iSyn extract operation constraints from the SmartIR to construct validation cases.**



***STOCK PURCHASE AGREEMENT***

*THIS STOCK PURCHASE AGREEMENT (this "Agreement") is made and entered into on February 6th, 2012 (the "Execution Date") by and among* ***BioAmber Inc., a Delaware corporation (the "Company")****, and* ***Lanxess Corporation, a Delaware corporation (the "Purchaser")****.*

*/*...Other sentences...*/*

***At the Closing*** *(as defined below), the* ***Company*** *shall sell and issue to the* ***Purchaser****, and,* ***subject to the terms and conditions set forth herein****, the Purchaser shall acquire and purchase from the Company, 10,030 Securities (as defined below) upon payment by the Purchaser of a purchase price of Nine Million Nine Hundred Ninety-Nine Thousand Nine Hundred and Ten US Dollars (****US $9,999,910****) (the "Purchase Price"), payable as set out in Section 2 of this Agreement.*

```
...

Entity: {
  SellerNames: ["BioAmber Inc."];
  BuyerNames: ["Lanxess Corporation"];
};
CloseTime: "February 6th, 2012";

...

OnlineStateTransfer: [{
  TimeConstraint: {
    operator: "<=",
    leftOprand: "now",
    rightOprand: CloseTime
  };
  DeliveryConstraint: true;
  (TimeContraint && DeliveryConstraint)
-> Payment {
      From: ["Lanxess Corporation"];
      To: ["BioAmber Inc."];
      Price: { Amount: "9999910",
Unit: "USD" };
    };
}];

...
```

```
/** */
function pay() public payable {
    /** */
    require(msg.sender == buyer[0]);
    uint currentTime = oracle.getTime();
    require(currentTime <= CloseTime);
    uint256 currentPrice = oracle.getPrice();
    uint256 price = 9999910;
    price = price / currentPrice;
    require(msg.value == price);
    /** */
}

/** */

function payRelease() public {
    require(msg.sender == buyer[0]);
    uint currentTime = oracle.getTime();
    require(currentTime <= CloseTime);
    require(purchaseBuyerConfirmed[0]);
    require(purchaseSellerConfirmed[0]);
    /** */
    seller.transfer(pricePayedByBuyer[0]);
    /** */
}
/** */
```

```
...
PaymentRoleCons: true,
PaymentTimeCons: true,
PaymentDeliveryCons: true,
PaymentPriceCons: true,
...
```

**Figure 2: An end-to-end example of smart contract synthesis, validation and deployment using iSyn.**

agreements, as driven by our empirical study. In the example shown in Figure 2, we highlight a payment intent between two entities once certain conditions are satisfied.

**SmartIR Construction.** SmartIR is our critical design to bridge the gap between the financial agreements written in natural languages and smart contracts written in programming languages. On the one hand, the intents expressed in legal agreements are automatically extracted in structured SmartIR instances, underpinned by rigorous grammar rules. On the other hand, the formal specification of SmartIR instances provides a well-defined scope for program synthesis. In this example, the payment intent is translated into a SmartIR item called *OnlineStateTransfer* with two constraints representing the conditions defined in the legal agreement. Internally, the above procedure is accomplished using our synergistic pipeline that unifies multiple NLP techniques.

**SmartIR Ambiguity Resolution with Knobs.** To embrace ambiguities in legal agreements, for each SmartIR item, iSyn provides the top-5 candidates sorted based on the confidence scores, and let stakeholders choose a more appropriate candidate using our knob. Once the SmartIR instance is agreed, the subsequent process requires no human intervention. Note that the top-1 candidate (90%+ change of being correct) is by default presented in the concrete SmartIR instance generated by iSyn.

**Smart Contract Synthesis.** The final blockchain-executable smart contracts are synthesized based on the SmartIR instances, predefined smart contract templates, and optional user inputs. A template essentially defines the "backbone" of the smart contract in form of abstract syntax tree (AST). The synthesis process then modifies, populates, or trims certain nodes of the AST, based on the SmartIR instances, to eventually output the synthesized smart contract. We employ a template-driven contract synthesis for the predictability of the final smart contract. The predictability facilitates the correctness check on the contract.

In this example, the SmartIR items are realized by two functions in Solidity, where the critical constraints are expressed via the require statements. The contract also relies on oracles to obtain authenticated off-chain data, such as the delivery date of physical goods associated with the payment.

**Smart Contract Validation.** We build a generic validation framework to verify the correctness of iSyn-synthesized smart contracts and perform automated vulnerability checks on the synthesized contracts. This framework generates comprehensive validation cases to cover all possible execution branches determined by the operation constraints extracted from the SmartIR instances. In this example, the price value and the payment time are two critical operation constraints validated by the framework.

**Post-deployment Interaction.** The stakeholders of the legal agreements can interact with the post-deployed smart contracts as typical Ethereum smart contracts. In this example, the payer first deposits its payment into the smart contract by constructing a transaction to the pay interface. Once the payment conditions are satisfied, the money is released to the payee by calling the payRelease interface using the data provided by the trusted oracle.

## 3.3 Intent Classification

Based on our empirical study, together with our understanding about smart contracts, we abstract four types of intents to represent the core transaction logic in the financial agreements. In particular, the *Entity* intent defines the participants or stakeholders involved in the legal agreement. The *OnlineStateTransfer* (*OfflineDelivery*) intent covers operations with (without) detailed onchain representations, and *Termination* defines various contract ending conditions. Concretely, our current design of *OnlineStateTransfer* supports *payment* (the most common operations we observe in financial agreements), property-backed Non-Fungible Token transfers (the recent trend of asset management), and using state published by certified organizations (*e.g.,* supporting the Web3.0 innovations [43]). The *OfflineDelivery* now supports oracles [17] for collecting authenticated offchain data. The *Termination* intents have three categories: termination upon offline deliveries, explicit expiry date, and other issues (for instance ending the employment at will).

Given this intent model, we fine-tune a RoBERTa [11] classifier, the state-of-the-art NLP transformer trained using the public reading comprehension dataset like SQuAD2 [19], using own labeled data to identify the sentences related to these transaction intents.

```
⟨SmartIR⟩                 ::= ⟨EntityDef⟩ (⟨TimeDef⟩)+ (⟨OfflineDeliveryDef⟩)?
                              ⟨OnlineStateTransferDef⟩ (⟨TerminationDef⟩)?
⟨ContractCategoryDef⟩     ::= ‘ContractCategory’ ‘:’ ⟨string⟩ + ‘;’

Entity:
⟨EntityDef⟩               ::= ‘Entity:’ + ‘{’ ⟨EntityNamesDefList⟩ ‘}’ + ‘;’
⟨EntityNamesDefList⟩      ::= (⟨EntityNamesDef⟩ ‘;’)+
⟨EntityNamesDef⟩          ::= ⟨EntityType⟩ ‘:’ ‘[’ ⟨EntityName⟩ (‘,’ ⟨EntityName⟩)* ‘]’
⟨EntityType⟩              ::= ⟨string⟩
⟨EntityName⟩              ::= ⟨string⟩

Time Constraint:
⟨TimeDef⟩                 ::= ⟨TimeLabel⟩ ‘:’ ⟨string⟩ ‘;’
⟨TimeLabel⟩               ::= ‘EffectiveTime’ | ‘CloseTime’ | ‘ExpiryTime’
⟨TimeConstraintDef⟩       ::= ‘TimeConstraint’ ‘:’ ⟨BinaryOperation⟩
⟨BinaryOperation⟩         ::= ‘{’ ⟨BinaryOperand⟩ ⟨BinaryOperator⟩ ⟨BinaryOperand⟩ ‘}’
⟨BinaryOperand⟩           ::= ⟨BinaryOperation⟩ | ⟨string⟩
⟨BinaryOperator⟩          ::= ‘>’ | ‘<’ | ‘>=’ | ‘<=’ | ‘==’ | ‘&&’ | ‘||’ | ‘+’ | ‘−’ | ‘*’ | ‘/’

OfflineDelivery:
⟨OfflineDeliveryDef⟩      ::= ‘OfflineDelivery’ ‘:’ ‘{’ ⟨DeliveryConstraintDef⟩ ‘}’
                              ‘;’
⟨DeliveryConstraintDef⟩   ::= ‘DeliveryConstraint’ ‘:’ ⟨bool⟩ | ⟨string⟩ ‘;’

OnlineStateTransfer:
⟨OnlineStateTransferDef⟩  ::= ‘OnlineStateTransfer’ ‘:’ ‘[’ (⟨PaymentDef⟩)+ ‘]’
                              ‘;’
⟨PaymentDef⟩              ::= ‘{’ ⟨PaymentContraintDef⟩ ‘->’ ⟨PaymentFunctionDef⟩ ‘}’
                              ‘;’
⟨PaymentContraintDef⟩     ::= ‘(’ (⟨TimeConstraintDef⟩)? (‘&&’)?
                              (⟨DeliveryConstraintDef⟩)? (‘&&’)? (⟨OtherConstraintDef⟩)?
                              ‘)’
⟨PaymentFunctionDef⟩      ::= ‘Payment’ ‘{’ ⟨SourceDef⟩ ‘,’ ⟨DestinationDef⟩ ‘,’
                              ⟨PriceDef⟩ ‘}’
⟨SourceDef⟩               ::= ‘From’ ‘:’ ⟨EntityName⟩
⟨DestinationDef⟩          ::= ‘To’ ‘:’ ⟨EntityName⟩
⟨PriceDef⟩                ::= ‘Price’ ‘:’ ‘{’ ‘Amount’ ‘:’ ⟨integer⟩ ‘,’ ‘Unit’ ‘:’
                              ⟨string⟩ ‘}’

Termination:
⟨TerminationDef⟩          ::= ‘Termination’ ‘:’ ‘{’ ⟨TerminationConstraintDef⟩ ‘}’ ‘;’
⟨TerminationConstraintDef⟩ ::= (⟨TimeConstraintDef⟩)? (‘||’ ⟨DeliveryConstraintDef⟩)?
                              (‘||’ ⟨OtherConstraintDef⟩)*
⟨OtherConstraintDef⟩      ::= ⟨bool⟩
```

**Grammar 1: BNF grammar of SmartIR**

## 3.4 SmartIR Design

To formally represent the intents in the intent model, we design an intermediate representation (IR), SmartIR. The BNF grammar of SmartIR is shown in Grammar 1. SmartIR mainly consists of four parts that are mapped to the four types of intents: ⟨*EntityDef*⟩, ⟨*OfflineDeliveryDef*⟩, ⟨*OnlineStateTransferDef*⟩, and ⟨*TerminationDef*⟩. Additionally, SmartIR uses ⟨*TimeDef*⟩ to define the effective time, close time, and the expiry time defined in the contract.

As shown in Figure 1, SmartIR is capable of representing complex semantics (such as the constraints in *OnlineStateTransfer* and *Termination* intents), as well as the dependencies among different intents. For example, the entities defined in the ⟨*EntityDef*⟩ intents are later referenced in the ⟨*OnlineStateTransferDef*⟩ intent, the dates defined in the ⟨*TimeDef*⟩ intents are referenced in various ⟨*TimeConstraintDef*⟩ intents, and the boolean values defined in the ⟨*OfflineDeliveryDef*⟩ can be used as the preconditions for the ⟨*OnlineStateTransferDef*⟩ intent. We next explain the SmartIR grammar rules in detail.

**Entity Definitions.** ⟨*EntityDef*⟩ defines all entities extracted from the *Entity* intents, such as purchasers (investors) and companies (sellers) in a typical stock purchasing agreement. Concretely, every ⟨*EntityNamesDef*⟩ declares all the entity names belonging to the same entity type, and puts them in ⟨*EntityNamesDefList*⟩. Entity names are used to uniquely identify entities during the smart contract synthesis process.

**OfflineDelivery Definitions.** The *OfflineDelivery* intents define the operations without detailed onchain representations, such as the delivery of written notice, certificate, receipt or document. For simplicity, ⟨*DeliveryConstraintDef*⟩ allows a boolean variable or a string to indicate whether offline operations are necessary, without further specifying detailed steps and constraints for these operations. In the smart contract synthesis process, we support using either smart contract oracles [78, 79] or file integrity validation functions to fulfill this definition.

**OnlineStateTransfer Definitions.** We use the payment operation as an example of the *OnlineStateTransfer* intents. We allow multiple payments defined in ⟨*OnlineStateTransferDef*⟩ to support the case where a financial agreement involves multiple buyers, sellers or different payment stages. Each payment operation is defined via ⟨*PaymentDef*⟩, which consists of the precondition for the payment (⟨*PaymentConstraintDef*⟩) and the payment operation details (⟨*PaymentFucntionDef*⟩). The payment precondition may consist of a time constraint (*e.g.,* a closing date) or a delivery constraint (*e.g.,* receiving a file), and only when all defined constraints are satisfied the payment will be executed. A ⟨*PaymentFucntionDef*⟩ must be declared with a three-element tuple: the pay action source (⟨*SourceDef*⟩), destination (⟨*DestinationDef*⟩), and the price payed (⟨*PriceDef*⟩). Other types of OnlineStateTransfer functionality defined in our model (*e.g.,* property-backed NFT transfers and using certified offchain states) can be supported similarly by extending the grammar definitions.

**Termination Definitions.** Typical termination conditions of financial agreements fall into three categories: termination triggered by offline delivery, agreement expiration, or other customized conditions (*e.g.,* employment termination at will). SmartIR presents these conditions using three types of constraints: ⟨*TimeConstraintDef*⟩ for representing the constraints related to the dates defined in ⟨*TimeDef*⟩ (*e.g.,* the expiry date), ⟨*DeliveryConstraintDef*⟩ for representing the offline delivery constraint, and ⟨*OtherTerminationDef*⟩

**Table 2: NLP techniques used for each intent type**

| Intent Type | NLP Techniques |
|---|---|
| *Entity* | QA, NER, Rules |
| *OnlineStateTransfer* | QA, NER, Synonym, POS |
| *OfflineDelivery* | Synonym, POS |
| *Termination* | NER, Synonym, POS |

for representing other constraints. If any of the conditions is defined in the *Termination* intents, SmartIR sets the corresponding definition to be true. Eventually, if any of these defined constraints are evaluated to *true*, the contract will be considered as terminated.
**TimeConstraint Definitions.** Time-related information contained in a financial agreement often serves as constraints or triggers for certain operations. For example, a time constraint with the 'CloseTime' label defines the deadline for a *payment* operation and its associated offline deliveries, while time constraints with the 'EffectiveTime' label and the 'ExpiryTime' labels define the life span of an agreement. In addition, time limits are uniformly expressed as nested binary operations to enable time comparison.

## 3.5 SmartIR Instantiation

iSyn instantiates SmartIR instances based on the critical information extracted from the legal agreements using our synergistic NLP pipeline. Based on the intent type, iSyn adopts different combinations of NLP techniques to extract the information from the text, as summarized in Table 2.
**Entity.** To instantiate the SmartIR statements for *Entity* intents, iSyn extracts name entities from the financial agreements. These name entities represent the stakeholders of the agreements, such as parties or organisations, and their corresponding attributes (*i.e.,* whether a party is a payer or payee). iSyn focuses on two types of entities for a legal agreement: the receiver and sender of *OnlineStateTransfer*, which are the major participants of the financial transactions. To detect named entities, iSyn builds a RoBERTa question answering model [19, 41], rather than using Named Entity Recognition (NER) as they still face the challenge of mapping these entities to the slots in the SmartIR statements (*e.g.,* how to distinguish payers and payees). Unlike NER, the question-answering (QA) model provides an answer for a given question, and iSyn can extract entities using the questions related to entity attributes, *e.g.,* "who is seller in this contract?". Particularly, using such kind of questions in a QA model is not sensitive to specific keywords, because the model can identify the synonym group automatically. To capture the different nouns used to represent entities, iSyn trains a QA model for each category of legal agreements. For each category, we fine-tune a model using our constructed benchmark dataset that consists of 100 *Entity* sentences collected from the 100 legal contracts in the same category.

Even so, due to the differences of writing styles, the QA model still face challenges in recognizing some entities. To address this problem, we synergistically combine QA, NER, and semantic rules to improve the accuracy of entity recognition. Specifically, iSyn first filters out sentences that are either too short or too long, as they mostly do not describe the receiver and the sender entities. The length limit is based on the statistical length of sentences in the labeled dataset. iSyn then filters out sentences that are describing legal terminologies rather then entities using keywords such as "mean". Finally, iSyn uses the entities found by the QA model to create the receiver and the sender entities; if the QA model fails to do so, we fall back to the NER results for creating the entities.
**Ambiguity in Entity Recognition.** Legal contracts often contain (even intentionally) ambiguous sentences when defining entities. To work around such ambiguity, iSyn also provides the top-$k$ entities returned by the QA model and corresponding confidence scores. The confidence scores are computed using the sum of probabilities assigned by the QA model for each returned entity. Given the confidence scores of the top-$k$ entities, iSyn automatically adopts the extracted entities if the confidence scores are high while initiating offline manual reviews otherwise [76].
**Offline Delivery.** To instantiate the SmartIR items for *OfflineDelivery* intents, iSyn uses a rule-based approach to identify the *OfflineDelivery* behavior between the signed entities in the legal agreement. If matches are found, a corresponding SmartIR item for the *OfflineDelivery* is generated with the nouns being the name for the ⟨*DeliveryConstraintDef*⟩ statement; otherwise, the *OfflineDelivery* sentence is ignored.
**OnlineStateTransfer and Termination.** To construct SmartIR items for *OnlineStateTransfer* and *Termination* intents, iSyn first extracts the date information that may be used in the preconditions or the definitions of these intents. Towards this end, iSyn applies NER to detect the entities labelled with the DATE tag for date information, and the entities with labelled the MONEY tag for price information. Since the *OfflineDelivery* intents may be used as the preconditions for the *OnlineStateTransfer* and *Termination* intents, iSyn also detects whether offline deliveries appear in these intents using the same rule-based approaches described above. If found, a precondition of the offline deliveries is added. Finally, iSyn uses the sender and the receiver entities extracted from the *Entity* intents as the payer and payee of the *OnlineStateTransfer* intents.

## 3.6 Smart Contract Synthesis

iSyn takes the SmartIR instances as input and employs a template-based synthesis protocol to synthesize to synthesize the smart contract deployable on blockchains. We currently use Solidity (0.5.x) as the target smart contract programming language. Yet, since SmartIR is language-agnostic, our synthesis protocol is applicable to other languages.
**Smart Contract Template Design.** A Solidity smart contract is a collection of functions and state variables (declared using *storage*). In the smart contract template, we program the ⟨*Entity*⟩ and ⟨*TimeConstraintDef*⟩ definitions as state variables and initialize them in the constructor. We convert the date strings in the ⟨*TimeDef*⟩ definitions as *uint*256 Unix timestamps for comparison purpose. We next describe the template design for the following SmartIR definitions: ⟨*PaymentDef*⟩ (the concrete example of ⟨*OnlineStateTransferDef*⟩), ⟨*OfflineDeliveryDef*⟩ and ⟨*TerminationDef*⟩. The complete template is shown in our technical report [3].

Each ⟨*PaymentDef*⟩ operation is mapped to two functions, *pay* and *payRelease.* In the *pay* function, the fund intended to be sent from a payer account to a receiver account is staked temporarily in the contract account and locked. The payer account and the receiver account of the pay function, corresponding to ⟨*SourceDef*⟩ and ⟨*DestinationDef*⟩, are restricted by the *require* statement. The funds currently stored in the contract account are recorded using state variables. The *payRelease* function is invoked by the payer to release the fund staked in the contract to the receiver. For the precondition of the payment operation defined in the ⟨*PaymentConstraintDef*⟩, our smart contract template provides the *payConfirm* function for the payer and the receiver to confirm if the preconditions are satisfied. The confirmation results are recorded as state variables and checked using the *require* statements in *payRelease*. Additionally, the price paid is specified using a local variable, which is consistent with the amount sent by the payer in the *pay* function. The time constraints of a payment operation are also evaluated using the *require* statement.

For ⟨*OfflineDeliveryDef*⟩ definitions, we provide the *uploadFileHash* function to store important string information (such as file content hashes) in a *mapping* state variable. This enables a wide range of dispute resolution mechanisms designed based on the commit-and-reveal primitive.

The template has different types of functions to realize different types of ⟨*TerminationDef*⟩ definitions. In particular, *terminateByDelivery* and *terminateConfirm* are defined for ⟨*DeliveryConstraintDef*⟩. Additionally, the *terminateByDelivery* function is also restricted by time constraints since deliveries are often associated with times in legal agreements. Similarly, the function *terminateByExpiry* designed for ⟨*TimeConstraintDef*⟩ is restricted by comparing the current timestamp with the expiry time, while the function *terminateByOther* designed for ⟨*OtherConstraintDef*⟩ relies on the offchain feeds from the oracles to decide the proper next steps.

The complete mapping between the SmartIR items and Solidity state variables, local variables, and functions in our smart contract template are deferred to the technical report [3].

**Template Instantiation.** To instantiate the template and generate the concrete smart contract, iSyn first uses a Solidity parser built upon ANTLR4 [50] to obtain the abstract syntax tree (AST) of the smart contract template and transform this AST based on the input SmartIR instance to generate a target AST. In particular, based on the SmartIR instance, iSyn decides whether an AST node needs to be modified (if the SmartIR defines a customized logic), or trimmed (if the corresponding boolean in SmartIR is evaluated as false), or retained the same if otherwise. For nodes requiring modifications (usually representing functions), iSyn updates the function body accordingly, for instance, by enforcing time constraints as *require* statements or specifying proper value for a local variable (*e.g.,* the price amount in the *pay* function). We also ensure that the synthesis will add proper security checks to eliminate vulnerabilities, such as integer overflow and reentrancy bugs [1, 22, 68].

The contract synthesis can also take as inputs user specified information that does not originally appear in legal agreements. Typical user inputs include blockchain accounts, the front-end contract address of Oracles [78], and the concrete certified state publishers [43]. The stakeholders, upon mutual agreement, could also overwrite certain SmartIR terms using directly provided inputs.

**Program Predictability.** One of the primary reasons for using predefined templates to drive smart contract synthesis is to ensure the predictability of the final smart contracts. This allows iSyn to design proper oracle interfaces that are compatible with the synthesized contracts Another benefit is that it facilitates the correctness check of the synthesized contracts based on the "ground truth" contracts developed by engineers following a programming style similar to that of the template, as we will discuss in § 4.1.

## 3.7 Smart Contract Validation

To ensure that the functionality of a synthesized smart contract correctly conforms to its SmartIR, we design a contract validation module in iSyn. The validation framework has two components: an *oracle contract* for feeding vital data to the synthesized smart contracts and a *validation case generator* for exploring various operation constraints in the contracts. Furthermore, we apply existing vulnerability detection tools [65, 68] to ensure that known vulnerabilities are eliminated on the synthesized contracts.

The oracle contract provides necessary external data sources for the smart contract, such as the current time, realtime token prices, as well as other external states that dictate whether certain constraints are satisfied or not. We design the oracle interfaces to be compatible with the expected formats of the iSyn-synthesized smart contracts (thanks to the template-driven contract synthesis protocol). These oracle interfaces can be easily extended to include more external data sources based on the needs of the studied legal agreements.

The validation case generator generates validation cases for a smart contract based on its SmartIR. In particular, the generator first parses SmartIR to extract all the operation constraints using an explicit mapping from the SmartIR items to the constraints. Similar to test generation via symbolic execution [26, 36, 59], the generator collects the constraints in a contract and uses the conjunction of these constraints, called path condition, to generate inputs for validating the smart contract. To obtain a new path that represents a different scenario, one of the constraints is negated to create a new path condition. To achieve the complete coverage of all possible scenarios, the generator finds path conditions that cover both the true and the false branches of each constraint. Different from path conditions in testing a program, the constraints in our path conditions are mutually independent, and thus finding a validation input that satisfies the path conditions is straightforward. Our empirical observation also finds that legal agreements rarely have constraints depending on each other. In practice, the validation inputs are crafted as function parameters or oracle data to fulfill certain operation constraints based on the path conditions. Note that we place comprehensive assertions in the validation cases to ensure that expected transaction failures can be captured.

## 4 EVALUATION

We built iSyn in ~6,000 lines of code (Python and Javascript). Our evaluations aim to answer the following research questions:

- **RQ1**: How effective is iSyn in synthesizing smart contracts from legal agreements?
- **RQ2**: How effective is iSyn in translating the core transaction logic in legal agreements into SmartIR instances?
- **RQ3**: How effective is iSyn in classifying intent related sentences?

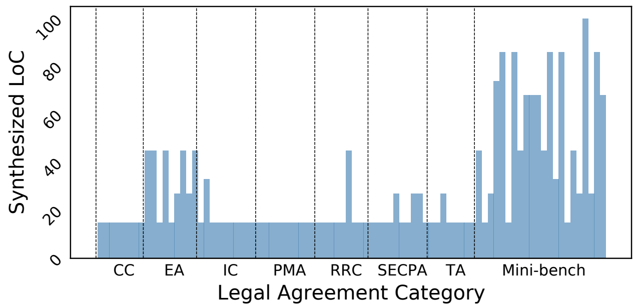


**Figure 3: Synthesized lines of code for each contract in the evaluation set (including inserted, updated and deleted lines based on the smart contract template)**

- **RQ4**: How effective is iSyn in validating various constraints defined in the legal agreements?

## 4.1 Evaluation Setup

**Evaluation Subject.** We use the legal agreements in Law Insider [33] as our evaluation subjects. As shown in Table 1, we first collected 8, 029 legal agreements from the top-10 most popular categories. We excluded the categories of *Joint Filling Agreement*, *Agreement*, *Underwriting Agreement* since they are not centric to financial transactions. Among the remaining 4, 879 contracts, we use the following approach to obtain the evaluation agreements. First, we select the legal agreements with the most *OnlineStateTransfer*-related sentences, and manually review them to exclude those with certain problems (*e.g.,* the contracts appeared in the same download file or the agreements assigned to the wrong category after reviewing the content). In total, we obtain 64 legal agreements from 7 categories. We further include another 22 contracts (referred to as mini-bench) that include sentences for all the four types of intents. In total, the evaluation dataset consists of 86 contracts with 129, 907 sentences, as summarized in Table 3. We applied iSyn to synthesize smart contracts (denoted as $P_s$) from these legal agreements. To evaluate how iSyn minimizes developer efforts, the synthesized smart contracts and their SmartIR used in RQ1 and RQ4 are the top-1 candidates automatically generated by iSyn without human interventions. As shown in Figure 3, the average LoC synthesized by iSyn to fill the slots of the template is 29, and the average total LoC of the synthesized smart contracts is 179 (including the template).

**Ground Truth Smart Contracts.** We manually create the "ground truth" smart contracts (denoted as $P_g$), for all the 86 legal agreements. We first inspected the legal agreements and identified the sentences related with the four types of intents. We then analyzed these intents to construct the core transaction logic, and develop a smart contract to represent the logic following the smart contract templates used in iSyn. We also invite multiple graduate students in our law school to help us build the ground-truth SmartIR dataset. To minimize the manual efforts and avoid tedious negotiation rounds, we did not let them compose the SmartIR from scratch. Instead, they read the legal agreements (each agreement read by two persons) in our evaluation dataset and manually reviewed the automatically generated SmartIR for us. We then further reviewed their comments and correct our ground truth. Typical corrections required by legal experts are about sender and receiver of *OnlineStateTransfer*.

**Ground Truth Validation Cases.** We also construct the "ground truths" validation cases to evaluate the quality of validation cases generated based on SmartIR instances. Towards this end, we, assisted by a group of legal experts, read all the 86 legal contracts in the evaluation set thoroughly and manually identify the operation constraints that should be used to construct the ground-truth validation cases. For consistency, we use the same set of operation constraints in both ground-truth validation cases and SmartIR-generated validation cases.

**Evaluation Metrics.** Due to the inherent ambiguity of legal agreements, $P_s$ will be inevitably different from $P_g$. As long as these differences are trivial (*i.e.,* $P_s \sim P_g$), developers can adopt these smart contracts with minor edits. Thus, we measure the effectiveness of iSyn in minimizing developer efforts by measuring the similarity between $P_s$ and $P_g$ using two metrics.

The first metric measures the structural similarity between $P_s$ and $P_g$. In particular, we use the similarity of program characteristic vectors defined in DECKARD [34] (referred to as *vector similarity*) as our metric. The core task of DECKARD is to construct program characteristic vectors. Towards this end, it first builds ASTs for the compared programs without considering detailed AST node values. Then it records occurrence counts of the *relevant AST nodes* that are essential to the tree structure, such as assignments, increments, arrays and condition expression nodes. These information is then recorded in the ordered dimensions of the characteristic vectors. As such, the similarities between the vectors of $P_s$ and $P_g$ can represent the similarity between $P_s$ and $P_g$ in terms of structure and code-quantity. We use the cosine value for the vectors of $P_s$ and $P_g$ to quantify their similarities.

The second metric measures the semantic correctness of $P_s$ when compared to $P_g$. Unlike traditional program synthesis tasks, in which the target program is as short as a SQL request or a shell script, iSyn synthesizes a holistic smart contract with complex programming logic. Thus, we prefer finer-grained function-level correctness checks when validating the semantic correctness of the synthesized contracts. Additionally, for incorrect functions, we measure how many edits are required to fix them. The definition of an *edit* is inspired by GumTreeDiff [21], *i.e.,* each edit (or edit action) means updating, deleting, adding or moving a single node of the program AST.

## 4.2 RQ1: Overall Effectiveness

We first present the effectiveness of minimizing developer efforts.

**Syntactic Evaluation.** As shown in Figure 4, the vector similarities for all 86 contracts are all above 0.995, demonstrating that $P_s$ are highly similar to $P_g$ in terms of syntactic structures. The reason for the similarity dips in IC, RRC and Mini-bench categories is that some *Termination* related functions are incorrectly included or removed. Fundamentally, this is caused by wrongly assigned SmartIR terms during the slot filling phase. In § 4.3, we will provide the detailed per-category F1 scores for SmartIR instantiation.

**Semantic Evaluation.** We perform function-level correctness verification in semantic evaluation. A function in $P_s$ is incorrectly synthesized if any of its statements is wrong or the entire function is missing. Table 4 present the average percentage of semantically correct functions for each contract category, as well as the average

**Table 3: Statistical information of the evaluation data set**

| Category | CC | EA | IC | PMA | RRC | SECPA | TA | Mini-bench | Total |
|---|---|---|---|---|---|---|---|---|---|
| # of Legal Agreements | 8 | 9 | 10 | 10 | 9 | 10 | 8 | 22 | 86 |
| # of Sentences | 65080 | 1702 | 32924 | 12749 | 3038 | 5427 | 4108 | 4879 | 129907 |

**Table 4: The percentage of semantically-correct functions in the synthesized smart contracts**

| Category | CC | EA | IC | PMA | RRC | SECPA | TA | Mini-bench | Average |
|---|---|---|---|---|---|---|---|---|---|
| Percentage | 91.25% | 91.57% | 87.84% | 88.00% | 92.31% | 95.79% | 83.56% | 88.16% | 89.80% |

**Table 5: Comparison of $F_1$ scores in SmartIR instantiation between iSyn and the other techniques**

| Category | Entity | | | OnlineStateTransfer | | | OfflineDelivery | | | Termination | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | QA | Rule | iSyn | QA | Rule | iSyn | QA | Rule | iSyn | QA | Rule | iSyn |
| **CC** | 0.27 | 0.13 | 0.82 | 1.00 | 1.00 | 1.00 | 1.00 | 1.00 | 1.00 | 0.50 | 1.00 | 1.00 |
| **EA** | 0.49 | 0.51 | 0.81 | 0.22 | 0.61 | 0.88 | 1.00 | 1.00 | 1.00 | 0.50 | 1.00 | 1.00 |
| **IC** | 0.42 | 0.22 | 0.81 | 0.70 | 0.80 | 0.80 | 1.00 | 1.00 | 1.00 | 0.65 | 0.55 | 0.55 |
| **PMA** | 0.34 | 0.18 | 0.72 | 0.60 | 1.00 | 1.00 | 1.00 | 1.00 | 1.00 | 0.50 | 1.00 | 1.00 |
| **RRC** | 0.37 | 0.31 | 0.68 | 0.33 | 0.76 | 0.83 | 0.67 | 1.00 | 1.00 | 0.55 | 0.73 | 0.72 |
| **SECPA** | 0.44 | 0.26 | 0.91 | 0.15 | 0.95 | 0.95 | 1.00 | 1.00 | 1.00 | 0.80 | 0.85 | 1.00 |
| **TA** | 0.21 | 0.14 | 0.63 | 0.63 | 0.63 | 0.63 | 1.00 | 1.00 | 1.00 | 0.63 | 0.93 | 0.94 |
| **Mini-bench** | 0.61 | 0.39 | 0.86 | 0.21 | 0.86 | 0.81 | 0.57 | 0.95 | 1.00 | 0.54 | 0.62 | 0.62 |
| **AVG** | 0.39 | 0.27 | 0.78 | 0.48 | 0.83 | 0.86 | 0.91 | 0.99 | 1.00 | 0.58 | 0.84 | 0.85 |

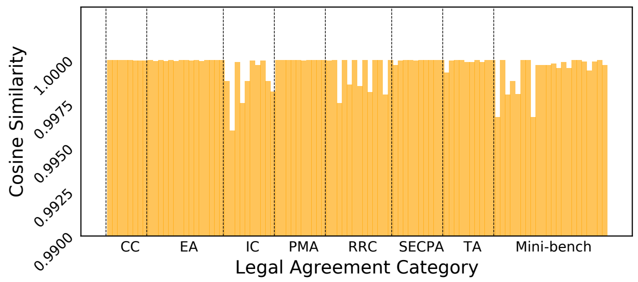


**Figure 4: Deckard characteristic vector cosine similarity between synthesized and ground truth smart contracts**

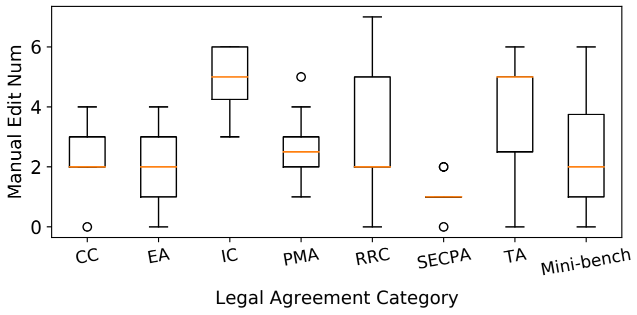


**Figure 5: The number of edits required to semantically correct the synthesized smart contracts (if necessary)**

percentage across all categories. Overall, iSyn achieves roughly 90% accuracy. There are four categories with average correctness rates lower than 90%. A closer inspection reveals that it is caused by the relatively low accuracy for processing *Entity*, *OnlineStateTransfer* or *Termination* intents during SmartIR instantiation. For instance, TA has the lowest accuracy for both *Entity* and *OnlineStateTransfer* in SmartIR instantiation. In § 4.3, we will give the detailed per-category accuracies for each type of intents during SmartIR instantiation. For semantically incorrect functions, we measure the number of edits (as defined in GumTreeDiff [21]) required to correct them. Then we count the total number of edits required for each synthesized smart contract with incorrect functions, and report the final results in Figure 5. Overall, the maximum number of edits to correct a contract is seven. The three contract categories with the highest edit counts are TA, IC and PMA, which is expected based on their function correctness rates in Table 4.

**Summary.** These results demonstrate that $P_s$ are syntactically similar to $P_g$, and it requires only minor edits to make $P_s$ semantically equivalent to $P_g$.

**Table 6: Top-$k$ results for the QA model**

| | Sender | | Receiver | |
|---|---|---|---|---|
| | Conf. | Min. Avg. | Conf. | Min. Avg. |
| **TOP-1** | 0.76 | 0.91 | 0.79 | 0.96 |
| **TOP-2** | 0.85 | 0.58 | 0.93 | 0.70 |
| **TOP-3** | 0.88 | 0.39 | 0.93 | 0.49 |
| **TOP-4** | 0.90 | 0.32 | 0.94 | 0.33 |
| **TOP-5** | 0.90 | 0.25 | 0.97 | 0.25 |

## 4.3 RQ2: SmartIR Correctness

To demonstrate the effectiveness of iSyn in SmartIR instantiation, we compare iSyn with two alternatives: the semantic rule-based technique [52, 56] (referred to as Rule) and the pre-trained RoBERTa QA model (referred to as QA). Rule utilizes the same keyword based method as iSyn to detect *OnlineStateTransfer* and *OfflineDelivery* related sentences, and utilizes the NER technique to extract signed entities, prices, and dates. QA uses the same set of questions as iSyn's QA model to retrieve answers for all the SmartIR items. To conduct fair comparison, both of QA and Rule also adopt the same RoBERTa model for intent classification.

We manually inspect the legal agreements to obtain the expected SmartIR items and compare them with the generated SmartIR items. We measure the true positives ($TP$) that describe the number of as-expected SmartIR items, false positives ($FP$) that describe the number of wrongly-generated SmartIR items, and false negatives ($FN$) that describe that number of missed (expected) SmartIR items. Based on these values, we compute the precision using $\frac{TP}{TP+FP}$, the recall using $\frac{TP}{TP+FN}$, and the $F_1$ score using $2 * \frac{prec*rec}{prec+rec}$. Table 5 shows the $F_1$ scores for the four types of intents. Clearly, iSyn achieves the highest $F_1$ score in all the categories, demonstrating its superiority over Rule and QA. Especially for *Entity* sentences, the $F_1$ improvement of iSyn over QA and Rule are 100.00% and 181.48%, respectively. Due to space limit, we show examples of extracted sentences in our project website [3].

**Ambiguity in QA**. Due to the ambiguity in defining entities using natural language, iSyn also provides the top-$k$ entities returned by the QA model (*i.e.,* RoBERTa). Table 6 shows statistical results about extracted top-$k$ entities for the *OnlineStateTransfer* sender and the *OnlineStateTransfer* receiver of the legal agreements found by the QA model. Column "Min. Avg." shows the minimum average probabilities computed by the QA model for the top-$k$ entities. Column "Conf." shows the confidence scores of the top-$k$ entities, which is defined as: if these $k$ entities contain the entity defined in ground truth, then the confidence is 1.0, otherwise the confidence is 0.0. The results show that with the increases of $k$, the confidence score increases accordingly, and the minimum average probability decreases. For *OnlineStateTransfer* sender and receiver, we notice that the confidence is above 0.9, indicating that the top-$k$ entities identified by iSyn are very likely to be the correct entities defined in the legal agreements.

## 4.4 RQ3: Intent Classification

To classify intent-related sentences from the legal agreements, we can potentially use any state-of-the-art NLP models. To justify our choice, we curate a benchmark dataset of 807 legal agreements and manually add the intent label for over 30,000 sentences in these legal agreements. The labeling process was conducted by 5 computer science graduates reading legal contracts verbatim, aided by 4 law school graduates. The whole process took us 3 months. We then apply multiple state-of-the-art NLP models on this dataset and compare their performances. To measure the effectiveness of intent classification, we utilizes *MCC* (Matthews correlation efficient) [35]. *MCC* is 1.00 if all samples are classified correctly and −1.00 if no samples are correctly classified. Random guess yields a zero *MCC*.

According to the model structure and different pre-trained setup (*e.g.,* number of layer, case sensitive or not, pre-trained data set), we study 10 models: RoBERTa [41], BERT-1 (case insensitive), BERT-2 (case sensitive) [15], DistilBERT-1 (case insensitive) [57], DistialBERT-2 (case sensitive), DistilBERT-3 (different pre-trained data), FlauBERT-1 (small structure and case sensitive) [39], FlauBERT-2 (base structure and case sensitive), FlauBERT-3 (base structure and case insensitive), and Xlnet [75]. All these models are publicly accessible in the Github [20]. We use 80% of the sentences for fine tuning the models,

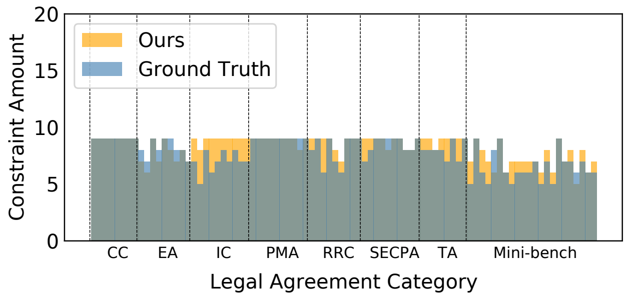


**Figure 6: The number of operation constraints used for generating ground truth validation cases and our validation cases**

and 20% of the sentences for model evaluation. The comparison is conducted using 5-fold validation. Table 8 shows the average *MCC* of each model. The *MCC* values of all models are above 0.849, and RoBERTa achieves the best performance (*MCC* being 0.884). These results demonstrate that the RoBERTa model fine-tuned with our benchmark dataset is highly effective in classifying intent-related sentences from the original legal agreements.

## 4.5 RQ4: Validation Case Quality

In iSyn, the validation cases are generated based on the corresponding operation constraints extracted from the SmartIR instances. Thus, to demonstrate the quality of the validation cases generated by iSyn, for each legal agreement in the evaluation set, we directly compare the operation constraints obtained from SmartIR with the operation constraints manually constructed by the legal experts based on careful examination of the legal agreements. As the set of operation constraints used in both scenarios is consistent, it is sufficient to directly compare the number of operation constraints. We plot the results in Figure 6. We further compute the *recall* and *precision* for each contract category, as shown in Table 7. Both results demonstrate a high recall, *i.e.,* the operation constraints extracted from SmartIR can cover nearly all the operation constraints presented in the original legal agreements. Meanwhile, the overall mean precision for all contracts is as high as 92.26%. The relatively low precision in IC, Mini-bench and TA is caused by the deficiency in processing *OnlineStateTransfer* and *Termination* intents during SmartIR instantiation for these contracts, as we show in Table 5.

## 5 DISCUSSION

**Threats to Validity.** The main internal threat to the validity of iSyn is the possible mistakes while constructing the ground truth smart contracts. To mitigate this threat, we integrate the comments from multiple legal experts when manually validating these ground truth contracts. The major external threat to validity originates from the scope of legal agreements that iSyn can process. To address this concern, we refine our scope to focus on legal agreements centering around financial transactions. This scope refinement is supported by data: through a large scale empirical study (§ 2.2), we find that over 80% of the legal agreements in top-10 most popular categories on Law Insider [33] primarily focus on financial transactions. Meanwhile, smart contracts and blockchains are commonly cited as the key digital enablers for the next generation financial

Table 7: *Recall* and *Precision* of the operation constraints used for generating validation cases

| Category | CC | EA | IC | PMA | RRC | SECPA | TA | Mini-bench | Average |
|---|---|---|---|---|---|---|---|---|---|
| Recall | 100.00% | 93.15% | 100.00% | 98.89% | 98.61% | 98.82% | 100.00% | 96.50% | 98.06% |
| Precision | 100.00% | 100.00% | 78.65% | 100.00% | 91.03% | 96.55% | 90.14% | 87.90% | 92.26% |

Table 8: Comparing NLP models in classifying intents from legal agreements

| Model | Case | MCC AVG | Deviation |
|---|---|---|---|
| **RoBERTa** | insensitive | 0.884 | 0.01 |
| **BERT-1** | insensitive | 0.879 | 0.01 |
| **BERT-2** | sensitive | 0.881 | 0.01 |
| **DistilBERT-1** | insensitive | 0.881 | 0.01 |
| **DistilBERT-2** | sensitive | 0.882 | 0.02 |
| **DistlBERT-3** | insensitive | 0.871 | 0.01 |
| **FlauBERT-1** | sensitive | 0.849 | 0.01 |
| **FlauBERT-2** | sensitive | 0.865 | 0.16 |
| **FlauBERT-3** | insensitive | 0.868 | 0.15 |
| **Xlnet** | sensitive | 0.881 | 0.01 |

systems [16, 58, 71]. Thus, financial transaction based legal agreements are reasonable starting points for iSyn. Further expanding the scope of iSyn is part of our future work.

**Template Mining & Sharing.** To support diverse requirements in legal agreements, iSyn will need to include more types of smart contract templates. We outline two potential approaches. First, we can mine the smart contracts related with the common behavior in legal agreements, and cluster them accordingly. For each cluster, we can infer a representative code sketch and use them as the basis to build additional templates for the specific type of legal agreement. Second, the users of iSyn could reward the template providers to stimulate participation. With increasing template availability, iSyn may further utilize the interactive learning framework [70] or example-based synthesis [7] to generate more templates.

**Supported Intent Types.** Based on our empirical study, iSyn supports four types of intents for representing core operations and conditions for financial legal agreements. To support other types of legal agreements (*e.g.,* voting and service agreements), we can extend our intent model to include additional types to represent their key logic. New intent classification and QA models should be trained accordingly to recognize these new intent types.

**Smart Contract Interoperability.** A line of future work for iSyn is representing the interoperability among multiple smart contracts [42]. For example, the data and execution state of one legal agreement may serve as the constraints and data input for an amendment legal agreement. Currently, the users of iSyn can manually establish this relationship on the synthesized smart contracts by inserting cross-contract function calls.

## 6 RELATED WORK

**Specification-based Program Synthesis.** Specification-based program synthesis [29, 63] uses sketches that express the semantics defined in target programs. Yet, the difficulty of writing complete specifications hinders applications. Another category of program synthesis techniques is driven either by demonstrations (PBD) [13, 37], *i.e.,* a trace of the user-performed behavior, or input-output examples (PBE) [27, 28, 40]. However, these program synthesis techniques essentially perform exhaustive search, and therefore are only capable of generating simple DSL targets such as datasheet scripts or telephone scripts. Given the complex semantics of the legal agreements and the dynamic statements of Turing-complete programming languages, these approaches cannot be directly applied.

**NLP-Based Program Synthesis.** Researchers propose the NLP-Based Program Synthesis to avoid writing specifications [14, 30, 74]. However, the input of these existing works is much smaller than the input of iSyn, which can include dozens of pages and thousands of sentences. Another line of research focuses on translating natural language texts to domain-specific programs [14, 30, 62, 63, 74]. These approaches mainly focus on synthesizing expressions with a few parameters for a DSL and the inputs are often one or two sentences. Thus, none of them have demonstrated the capability of extracting the core transaction logic from thousands of sentences and presenting the logic as a holistic software program with dozens of functions and hundreds of lines of code.

**NLP-based Legal Document Analysis.** With the rapid growth of NLP techniques, AI researchers together with legal experts have carried out significant efforts in automatic legal document analysis, roughly categorized as symbol-based methods [10, 47, 60] and embedding-based methods [8, 12, 46]. These research works aim at undertaking tedious legal jobs, such as retrieving and understanding lengthy legal documents, thus reducing the heavy burden of legal professionals and promoting the efficiency of legal institutions. Though legal document analysis has been widely deployed in areas like judgment prediction [9, 48], similar case matching [64, 66] and text summarization [53, 80], little has been done to enable the extraction of smart contracts from legal agreements.

## 7 CONCLUSION

In this paper, we presented iSyn, the first semi-automated system that synthesizes blockchain-executable smart contracts based on legal financial agreements, with minimal human interventions. iSyn is designed around a novel SmartIR design that bridges the gap between financial agreements and smart contracts: (i) iSyn applies a synergistic NLP pipeline to translate the transaction intents in legal agreements into SmartIR instances defined in rigorous grammar rules; (ii) iSyn applies a template-driven program synthesis to output the smart contracts given these SmartIR instances. We implemented a prototype of iSyn in roughly 6,000 lines of code and perform extensive evaluations to demonstrate its effectiveness.

## ACKNOWLEDGMENTS

We thank the anonymous reviewers for their constructive feedback. Xusheng Xiao's work is partially supported by the National Science Foundation under the grants CCF-2318483 and CNS-2318486. Zhuotao Liu and Xusheng Xiao are corresponding authors.

## REFERENCES

[1] 2021. TheDAO. https://etherscan.io/token/0xbb9bc244d798123fde783fcc1c72d3bb8c189413.

[2] 2023. HyperLedger: Making Legal Contracts Smart. https://www.hyperledger.org/blog/2018/01/11/making-legal-contracts-smart.

[3] 2023. iSyn Project Website. https://github.com/smartcontractsyn/iSyn.

[4] 2023. OpenLaw: Real World Contracts for Ethereum. https://www.openlaw.io.

[5] Solidity Authors. 2023. Solidity. https://docs.soliditylang.org/en/v0.8.4/.

[6] Vitalik Buterin. 2013. Ethereum: a next generation smart contract and decentralized application platform. https://github.com/ethereum/ wiki/wiki/White-Paper.

[7] Pavol Černỳ, Sivakanth Gopi, Thomas A Henzinger, Arjun Radhakrishna, and Nishant Totla. 2012. Synthesis from incompatible specifications. In *Proceedings of the ACM international conference on Embedded software (EMSOFT)*. 53–62. https://doi.org/10.1145/2380356.2380371

[8] Ilias Chalkidis and Dimitrios Kampas. 2019. Deep learning in law: early adaptation and legal word embeddings trained on large corpora. *Artificial Intelligence and Law* 27, 2 (2019), 171–198. https://doi.org/10.1007/s10506-018-9238-9

[9] Huajie Chen, Deng Cai, Wei Dai, Zehui Dai, and Yadong Ding. 2019. Charge-Based Prison Term Prediction with Deep Gating Network. In *Proceedings of the Conference on Empirical Methods in Natural Language Processing and the International Joint Conference on Natural Language Processing (EMNLP-IJCNLP)*. 6362–6367. https://doi.org/10.18653/v1/D19-1667

[10] Fenia Christopoulou, Makoto Miwa, and Sophia Ananiadou. 2018. A Walk-based Model on Entity Graphs for Relation Extraction. In *Proceedings of the Annual Meeting of the Association for Computational Linguistics (Volume 2: Short Papers)*. 81–88. https://doi.org/10.18653/v1/P18-2014

[11] Alexis Conneau, Kartikay Khandelwal, Naman Goyal, Vishrav Chaudhary, Guillaume Wenzek, Francisco Guzmán, Edouard Grave, Myle Ott, Luke Zettlemoyer, and Veselin Stoyanov. 2020. Unsupervised Cross-lingual Representation Learning at Scale. In *Proceedings of the Annual Meeting of the Association for Computational Linguistics (ACL)*. 8440–8451. https://doi.org/10.18653/v1/2020.acl-main.747

[12] František Cvrček, Karel Pala, and Pavel Rychlý. 2012. Legal electronic dictionary for Czech. In *Proceedings of the International Conference on Language Resources and Evaluation (LREC)*. 283–287. http://www.lrec-conf.org/proceedings/lrec2012/pdf/775_Paper.pdf

[13] Allen Cypher and Daniel Conrad Halbert. 1993. *Watch what I do: programming by demonstration.*

[14] Aditya Desai, Sumit Gulwani, Vineet Hingorani, Nidhi Jain, Amey Karkare, Mark Marron, and Subhajit Roy. 2016. Program synthesis using natural language. In *Proceedings of the International Conference on Software Engineering (ICSE)*. 345–356. https://doi.org/10.1145/2884781.2884786

[15] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2018. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805* (2018). https://doi.org/10.48550/arXiv.1810.04805

[16] Faris Elghaish, Sepehr Abrishami, and M Reza Hosseini. 2020. Integrated project delivery with blockchain: An automated financial system. *Automation in construction* 114 (2020), 103182. https://doi.org/10.1016/j.autcon.2020.103182

[17] Ethereum. 2022. ORACLES. https://ethereum.org/en/developers/docs/oracles/.

[18] Ethereum. 2023. Ethereum. https://ethereum.org/en/.

[19] Pranav Rajpurkar etl. 2016. Stanford Question Answering Dataset. https://rajpurkar.github.io/SQuAD-explorer/.

[20] Hugging Face. 2021. Huggingface Transformers. https://github.com/huggingface/transformers.

[21] Jean-Rémy Falleri, Floréal Morandat, Xavier Blanc, Matias Martinez, and Martin Monperrus. 2014. Fine-grained and accurate source code differencing. In *Proceedings of the ACM/IEEE International Conference on Automated Software Engineering (ASE)*. 313–324. https://doi.org/10.1145/2642937.2642982

[22] Josselin Feist, Gustavo Grieco, and Alex Groce. 2019. Slither: a static analysis framework for smart contracts. In *Proceedings of the IEEE/ACM International Workshop on Emerging Trends in Software Engineering for Blockchain (WETSEB)*. IEEE, 8–15. https://doi.org/10.1109/WETSEB.2019.00008

[23] G David Forney. 1973. The viterbi algorithm. *Proceedings of the IEEE* 61, 3 (1973), 268–278. https://doi.org/10.1109/PROC.1973.9030

[24] Norton Rose Fulbright. 2016. Can smart contracts be legally binding contracts. *An R3 and Norton Rose Fulbright White Paper* (2016).

[25] Mark Giancaspro. 2017. Is a 'smart contract' really a smart idea? Insights from a legal perspective. *Computer law & security review* 33, 6 (2017), 825–835. https://doi.org/10.1016/j.clsr.2017.05.007

[26] Patrice Godefroid, Nils Klarlund, and Koushik Sen. 2005. DART: Directed automated random testing. In *Proceedings of the ACM SIGPLAN conference on Programming language design and implementation (PLDI)*. 213–223. https://doi.org/10.1145/1064978.1065036

[27] Sumit Gulwani. 2011. Automating String Processing in Spreadsheets Using Input-Output Examples. In *Proceedings of the Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*. 317–330. https://doi.org/10.1145/1926385.1926423

[28] Sumit Gulwani, William R. Harris, and Rishabh Singh. 2012. Spreadsheet Data Manipulation Using Examples. *Commun. ACM* 55, 8 (aug 2012), 97–105. https://doi.org/10.1145/2240236.2240260

[29] Sumit Gulwani, Susmit Jha, Ashish Tiwari, and Ramarathnam Venkatesan. 2011. Synthesis of Loop-Free Programs. *SIGPLAN Not.* 46, 6 (jun 2011), 62–73. https://doi.org/10.1145/1993316.1993506

[30] Jiaqi Guo, Zecheng Zhan, Yan Gao, Yan Xiao, Jian-Guang Lou, Ting Liu, and Dongmei Zhang. 2019. Towards Complex Text-to-SQL in Cross-Domain Database with Intermediate Representation. In *Proceedings of the Annual Meeting of the Association for Computational Linguistics (ACL)*. 4524–4535. https://doi.org/10.18653/v1/P19-1444

[31] Maryam Habibi, Leon Weber, Mariana Neves, David Luis Wiegandt, and Ulf Leser. 2017. Deep learning with word embeddings improves biomedical named entity recognition. *Bioinformatics* 33, 14 (2017), i37–i48. https://doi.org/10.1093/bioinformatics/btx228

[32] DocuSign Inc. 2021. DocuSign. https://www.docusign.com/.

[33] Law Insider. 2022. Law Insider. https://www.lawinsider.com/.

[34] Lingxiao Jiang, Ghassan Misherghi, Zhendong Su, and Stephane Glondu. 2007. Deckard: Scalable and accurate tree-based detection of code clones. In *Proceedings of the International Conference on Software Engineering (ICSE)*. IEEE, 96–105. https://doi.org/10.1109/ICSE.2007.30

[35] Giuseppe Jurman, Samantha Riccadonna, and Cesare Furlanello. 2012. A comparison of MCC and CEN error measures in multi-class prediction. *PloS one* 7, 8 (2012), e41882. https://doi.org/10.1371/journal.pone.0041882

[36] James C King. 1976. Symbolic execution and program testing. *Commun. ACM* 19, 7 (1976), 385–394. https://doi.org/10.1145/360248.360252

[37] Tessa Lau, Steven A Wolfman, Pedro Domingos, and Daniel S Weld. 2003. Programming by demonstration using version space algebra. *Machine Learning* 53, 1 (2003), 111–156. https://doi.org/10.1023/A:1025671410623

[38] LESSIG Lawrence. 2000. Code is law. On Liberty in Cyberspace. *Harvard magazine* (2000).

[39] Hang Le, Loïc Vial, Jibril Frej, Vincent Segonne, Maximin Coavoux, Benjamin Lecouteux, Alexandre Allauzen, Benoit Crabbé, Laurent Besacier, and Didier Schwab. 2020. FlauBERT: Unsupervised Language Model Pre-training for French. In *Proceedings of the Language Resources and Evaluation Conference (LREC)*. 2479–2490. https://aclanthology.org/2020.lrec-1.302

[40] Henry Lieberman. 2001. *Your wish is my command: Programming by example.*

[41] Yinhan Liu, Myle Ott, Naman Goyal, Jingfei Du, Mandar Joshi, Danqi Chen, Omer Levy, Mike Lewis, Luke Zettlemoyer, and Veselin Stoyanov. 2019. Roberta: A robustly optimized bert pretraining approach. *arXiv preprint arXiv:1907.11692* (2019). https://doi.org/10.48550/arXiv.1907.11692

[42] Zhuotao Liu, Yangxi Xiang, Jian Shi, Peng Gao, Haoyu Wang, Xusheng Xiao, Bihan Wen, and Yih-Chun Hu. 2019. Hyperservice: Interoperability and programmability across heterogeneous blockchains. In *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security (CCS)*. 549–566. https://doi.org/10.1145/3319535.3355503

[43] Zhuotao Liu, Yangxi Xiang, Jian Shi, Peng Gao, Haoyu Wang, Xusheng Xiao, Bihan Wen, Qi Li, and Yih-Chun Hu. 2022. Make Web3.0 Connected. *IEEE Transactions on Dependable and Secure Computing (TDSC)* (2022). https://doi.org/10.1109/TDSC.2021.3079315

[44] George A Miller. 1995. WordNet: a lexical database for English. *Commun. ACM* 38, 11 (1995), 39–41. https://doi.org/10.1145/219717.219748

[45] Satoshi Nakamoto. 2009. Bitcoin: A Peer-to-Peer Electronic Cash System. (2009).

[46] John J. Nay. 2016. Gov2Vec: Learning Distributed Representations of Institutions and Their Legal Text. In *Proceedings of the First Workshop on NLP and Computational Social Science (NLP+CSS)*. 49–54. https://doi.org/10.18653/v1/W16-5607

[47] Thien Huu Nguyen and Ralph Grishman. 2018. Graph Convolutional Networks with Argument-Aware Pooling for Event Detection. In *Proceedings of the AAAI Conference on Artificial Intelligence and Innovative Applications of Artificial Intelligence Conference and AAAI Symposium on Educational Advances in Artificial Intelligence (AAAI/IAAI/EAAI)*. Article 724, 8 pages. https://doi.org/10.1609/aaai.v32i1.12039

[48] Sicheng Pan, Tun Lu, Ning Gu, Huajuan Zhang, and Chunlin Xu. 2019. Charge prediction for multi-defendant cases with multi-scale attention. In *Computer Supported Cooperative Work and Social Computing (ChineseCSCW)*. Springer, 766–777. https://doi.org/10.1007/978-981-15-1377-0_59

[49] Rahul Pandita, Xusheng Xiao, Hao Zhong, Tao Xie, Stephen Oney, and Amit Paradkar. 2012. Inferring method specifications from natural language API descriptions. In *international conference on software engineering (ICSE)*. IEEE, 815–825. https://doi.org/10.1109/ICSE.2012.6227137

[50] Terence Parr. 2014. ANTLR (ANother Tool for Language Recognition). https://www.antlr.org/.

[51] Gayane Petrosyan, Martin P Robillard, and Renato De Mori. 2015. Discovering information explaining API types using text classification. In *IEEE International Conference on Software Engineering (ICSE)*, Vol. 1. IEEE, 869–879. https://doi.org/10.1109/ICSE.2015.97

[52] Guang Qiu, Bing Liu, Jiajun Bu, and Chun Chen. 2011. Opinion word expansion and target extraction through double propagation. *Computational linguistics* 37,

1 (2011), 9–27.

[53] Pranav Rajpurkar, Robin Jia, and Percy Liang. 2018. Know What You Don't Know: Unanswerable Questions for SQuAD. In *Proceedings of the Annual Meeting of the Association for Computational Linguistics (Volume 2: Short Papers)*. 784–789. https://doi.org/10.18653/v1/P18-2124

[54] Max Raskin. 2016. The law and legality of smart contracts. (2016).

[55] Lisa F Rau. 1991. Extracting company names from text. In *Proceedings of the IEEE Conference on Artificial Intelligence Application (CAIA)*. IEEE Computer Society, 29–30. https://doi.org/10.1109/CAIA.1991.120841

[56] Alan Ritter, Sam Clark, Oren Etzioni, et al. 2011. Named entity recognition in tweets: an experimental study. In *Proceedings of the Conference on Empirical Methods in Natural Language Processing (EMNLP)*. 1524–1534.

[57] Victor Sanh, Lysandre Debut, Julien Chaumond, and Thomas Wolf. 2019. DistilBERT, a distilled version of BERT: smaller, faster, cheaper and lighter. *arXiv preprint arXiv:1910.01108* (2019). https://doi.org/10.48550/arXiv.1910.01108

[58] Fabian Schär. 2021. Decentralized finance: On blockchain-and smart contract-based financial markets. *FRB of St. Louis Review* (2021). https://doi.org/10.20955/r.103.153-74

[59] Koushik Sen, Darko Marinov, and Gul Agha. 2005. CUTE: A concolic unit testing engine for C. *ACM SIGSOFT Software Engineering Notes* 30, 5 (2005), 263–272. https://doi.org/10.1145/1095430.1081750

[60] Yi Shu, Yao Zhao, Xianghui Zeng, and Qingli Ma. 2019. *Cail2019-fe*. Technical report, Gridsum.

[61] John Slankas, Xusheng Xiao, Laurie Williams, and Tao Xie. 2014. Relation extraction for inferring access control rules from natural language artifacts. In *Proceedings of the Annual Computer Security Applications Conference (ACSAC)*. 366–375. https://doi.org/10.1145/2664243.2664280

[62] Armando Solar-Lezama. 2008. *Program Synthesis by Sketching*. Ph. D. Dissertation. Advisor(s) Bodik, Rastislav.

[63] Armando Solar-Lezama, Liviu Tancau, Rastislav Bodik, Sanjit Seshia, and Vijay Saraswat. 2006. Combinatorial sketching for finite programs. In *Proceedings of the international conference on Architectural support for programming languages and operating systems (ASPLOS)*. 404–415. https://doi.org/10.1145/1168857.1168907

[64] Keet Sugathadasa, Buddhi Ayesha, Nisansa de Silva, Amal Shehan Perera, Vindula Jayawardana, Dimuthu Lakmal, and Madhavi Perera. 2019. Legal document retrieval using document vector embeddings and deep learning. In *Intelligent Computing: Proceedings the Computing Conference, Volume 2*. Springer, 160–175. https://doi.org/10.1007/978-3-030-01177-2_12

[65] Sergei Tikhomirov, Ekaterina Voskresenskaya, Ivan Ivanitskiy, Ramil Takhaviev, Evgeny Marchenko, and Yaroslav Alexandrov. 2018. Smartcheck: Static analysis of ethereum smart contracts. In *Proceedings of the International Workshop on Emerging Trends in Software Engineering for Blockchain (WETSEB)*. 9–16. https://doi.org/10.1145/3194113.3194115

[66] Vu Tran, Minh Le Nguyen, and Ken Satoh. 2019. Building legal case retrieval systems with lexical matching and summarization using a pre-trained phrase scoring model. In *Proceedings of the International Conference on Artificial Intelligence and Law (ICAIL)*. 275–282. https://doi.org/10.1145/3322640.3326740

[67] Van Cuong Tran, Ngoc Thanh Nguyen, Hamido Fujita, Dinh Tuyen Hoang, and Dosam Hwang. 2017. A combination of active learning and self-learning for named entity recognition on twitter using conditional random fields. *Knowledge-Based Systems* 132 (2017), 179–187. https://doi.org/10.1016/j.knosys.2017.06.023

[68] Petar Tsankov, Andrei Dan, Dana Drachsler-Cohen, Arthur Gervais, Florian Buenzli, and Martin Vechev. 2018. Securify: Practical security analysis of smart contracts. In *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security (CCS)*. ACM, 67–82. https://doi.org/10.1145/3243734.3243780

[69] Tanash Utamchandani Tulsidas. 2018. Smart contracts from a legal perspective. (2018).

[70] Arie Van Deursen, Paul Klint, and Joost Visser. 2000. Domain-specific languages: An annotated bibliography. *ACM Sigplan Notices* 35, 6 (2000), 26–36. https://doi.org/10.1145/352029.352035

[71] Hao Wang, Chaonian Guo, and Shuhan Cheng. 2019. LoC—A new financial loan management system based on smart contracts. *Future Generation Computer Systems* 100 (2019), 648–655. https://doi.org/10.1016/j.future.2019.05.040

[72] Hong Wu and Guan Zheng. 2020. Electronic evidence in the blockchain era: New rules on authenticity and integrity. *Computer Law & Security Review* 36 (2020), 105401. https://doi.org/10.1016/j.clsr.2020.105401

[73] Xusheng Xiao, Amit Paradkar, Suresh Thummalapenta, and Tao Xie. 2012. Automated extraction of security policies from natural-language software documents. In *Proceedings of the ACM SIGSOFT International Symposium on the Foundations of Software Engineering (FSE)*. 1–11. https://doi.org/10.1145/2393596.2393608

[74] Navid Yaghmazadeh, Yuepeng Wang, Isil Dillig, and Thomas Dillig. 2017. SQLizer: query synthesis from natural language. *Proceedings of the ACM on Programming Languages* 1, OOPSLA (2017), 1–26. https://doi.org/10.1145/3133887

[75] Zhilin Yang, Zihang Dai, Yiming Yang, Jaime Carbonell, Ruslan Salakhutdinov, and Quoc V Le. 2019. XLNet: Generalized Autoregressive Pretraining for Language Understanding. *arXiv preprint arXiv:1906.08237* (2019). https://doi.org/10.48550/arXiv.1906.08237

[76] Jane Yen, Tamás Lévai, Qinyuan Ye, Xiang Ren, Ramesh Govindan, and Barath Raghavan. 2021. Semi-automated protocol disambiguation and code generation. In *Proceedings of the ACM SIGCOMM Conference (SIGCOMM)*. 272–286. https://doi.org/10.1145/3452296.3472910

[77] Le Yu, Tao Zhang, Xiapu Luo, and Lei Xue. 2015. AutoPPG: Towards automatic generation of privacy policy for android applications. In *Proceedings of the Annual ACM CCS Workshop on Security and Privacy in Smartphones and Mobile Devices (SPSM)*. 39–50. https://doi.org/10.1145/2808117.2808125

[78] Fan Zhang, Ethan Cecchetti, Kyle Croman, Ari Juels, and Elaine Shi. 2016. Town Crier: An authenticated data feed for smart contracts. In *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security (CCS)*. 270–282. https://doi.org/10.1145/2976749.2978326

[79] Fan Zhang, Deepak Maram, Harjasleen Malvai, Steven Goldfeder, and Ari Juels. 2020. DECO: Liberating web data using decentralized oracles for TLS. In *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security (CCS)*. 1919–1938. https://doi.org/10.1145/3372297.3417239

[80] Haoxi Zhong, Yuzhong Wang, Cunchao Tu, Tianyang Zhang, Zhiyuan Liu, and Maosong Sun. 2020. Iteratively questioning and answering for interpretable legal judgment prediction. In *Proceedings of the AAAI Conference on Artificial Intelligence (AAAI)*, Vol. 34. 1250–1257. https://doi.org/10.1609/aaai.v34i01.5479

[81] Xiaohua Zhou, Xiaodan Zhang, and Xiaohua Hu. 2006. MaxMatcher: Biological concept extraction using approximate dictionary lookup. In *Proceedings of the Pacific Rim International Conference on Artificial Intelligence (PRICAI)*. Springer, 1145–1149. https://doi.org/10.1007/978-3-540-36668-3_150

Received 2023-02-16; accepted 2023-05-03