SPESC-Translator: Towards Automatically Smart Legal Contract Conversion for Blockchain-based Auction Services

E Chen, Bohan Qin, Yan Zhu, *Member, IEEE,* Weijing Song, Shengdian Wang, Cheng-Chung William Chu, *Senior Member, IEEE,* Stephen S. Yau, *Fellow, IEEE*

Abstract—In recent years, advanced smart contract languages (ASCLs) have been proposed to solve the problem of difficult reading, comprehension, and collaboration when writing smart legal contracts among people in different fields. However, this kind of languages are still hard to put into practice due to the lack of an effective conversion method from the ASCLs to executable smart contract programs. Aiming at this problem, we take SPESC as example to explore how to design conversion rules from the contract in it to the target programming language in Solidity, and to propose a three-layer smart contract framework, including advanced smart-contract layer, general smart-contract layer, and executable machine-code layer. These rules provide an approach to convert the definition of SPESC contracting parties into party-contracts on target language, as well as to produce SPESC contract terms into main-contract on target language. Moreover, the proposed framework specifies not only program architecture and storage structure on general smart-contract layer, but also important mechanisms, including personnel management, timing control, exception handling, etc., which can assist programmers to write smart contract programs. Furthermore, taking four SPESC contracts as testing objects, we provide the whole process of converting from SPESC contracts to Solidity programs by the SPESC-Translator, and verify the efficiency and security of the conversion process, including coding, deploying, running, and testing through Ethereum. The instance results show that the conversion rules and the three-layer framework can simplify the writing of smart contracts, standardize the program structure, and help programmers to verify the correctness of the contract programs.

Index Terms—Smart contract, blockchain, advanced smart contract languages, smart legal contract, SPESC, automated generation.

1 Introduction

As second-generation blockchain technology, smart contracts have greatly enriched the functional expression of blockchain to make application development more convenient. The term "smart contract" was originally coined by Nick Szabo in 1995 [1]. Broadly speaking, smart contract is a set of digital executable protocols [2] intended to make contractual clauses partially or fully self-executing, self-enforcing, or both. This means that smart contracts can essentially be programmed to implement a wide variety of actions, so as to provide an entire platform for new applications designed to solve many real-world problems. Therefore, it has attracted widespread attentions from academic and industrial fields in recent years.

In a narrow sense, a smart contract is a computer program deployed and run on a blockchain, and automatically executes when predetermined terms and conditions are met. Exactly, all related data of smart contract, including program codes, intermediate states, and executed results, would be stored in the blockchain to ensure that these data are not tampered with. Moreover, the blockchain's consensus protocols also verify the correctness of running process by executing the smart contract with the same input at all nodes. Therefore, the blockchain's security mechanism with tamper-proof and traceable features makes smart contracts possible to be recognized by law.

- *E Chen, B. Qin, Y. Zhu, W. Song and S. Wang were with the School of Computer and Communication Engineering, University of Science and Technology Beijing, Beijing, 10083 China.*
- *C.-C. W. Chu was with the Computer Science Department, Tunghai University, Taichung City, 40730 Taiwan.*
- *S. S. Yau was with the School of Computing, Informatics, and Decision Systems Engineering, Arizona State University, TEMPE, AZ 85281.*
- *Corresponding author: Yan Zhu (E-mail: zhuyan@ustb.edu.cn)*

Manuscript received April 19, 2005; revised August 26, 2015.

Compared with Bitcoin's script system, the existing blockchain's smart contracts can process more complex business logic and have more flexibility to adopt blockchain for storing various data (including contract intermediate states). Therefore, almost of blockchain platforms and manufacturers (such as Ethereum and Hyperledger Fabric) have developed smart contract mechanisms to improve the usability of their products. For example, the Ethereum platform currently supports two programming languages, Serpent and Solidity, where the former is similar to Python and the latter is to JavaScript. As another example, Hyperledger supports traditional languages such as Go and Java. In addition, other platforms also provide some development tools based on traditional programming languages (such as C, C ++, Java) for smart contracts.

Motivation. In summary, the current smart contracts could be divided into three categories in terms of language types and operating environments, as follows:

- **Script-based smart contract**, which supports simple calculation and condition control through script instructions and Forth-like stack-based language defined in the blockchain, such as the Bitcoin script system;
- **General-purpose smart contract**, which directly adopts traditional programming language deployed in a virtual machine or docker to interact with the blockchain through a prescribed interface. For example, the chaincode in the Hyperledger platform uses the languages such as Java and Go. The Neo platform provides the compilation of multiple languages (such as C#, Java, and Python) into the instruction set supported by NeoVM;
- **Specific-purpose smart contract**, which uses new domain-specified languages or adds special elements to traditional programming languages for interacting with blockchain,

such as Ethereum's Solidity language with special mechanisms (e.g., gas billing).

From the above categories, it is easy to see that smart contract platforms and languages have become increasingly mature and complete in recent years. However, they are dominated by the general-purpose smart contracts at present, and the specific-purpose smart contracts are still in the development stage. Since smart contracts usually involve cooperation in the different fields, such as IT, law, and finance, the current smart contract programming languages remain some problems to be solved, such as being unfriendly to people in the non-computer field and difficult to understand for those who have not studied programming on blockchain. Specifically, these languages have the following disadvantages:

- Smart contract language obeys the grammar of computer languages, and is far from the form of real-world legal contracts;
- Smart contract language requires great knowledge and experience in computer, and is difficult for users and legal personnel to understand;
- There is a lack of approach for direct generation from real-world legal contract to executable smart contract.

These disadvantages make it very difficult to write a smart contract, and hit an obstacle in communication between people in different fields. Therefore, they restrict the efficient development of smart contracts and the degree of public acceptance.

To solve these problems, advanced smart contract language (ASCL) has been proposed to build a bridge between real-world legal contracts and blockchain-based smart contracts. Through this bridge, ASCL helps people in different fields to communicate through an easy-to-read and standardized syntax, and it can (semi-)automatically realize the conversion to smart contract language on blockchain, so as to assist programmers in writing smart contracts.

Related work. Since the concept of smart contracts based on blockchain was proposed, the scholars have done a lot of researches in many aspects, including law, program design, platform construction, etc., especially some new languages. Many of these languages were to verify the correctness of the contracts through formal methods, such as, the logic-based languages [3] and the generic description language [4]. The Flint language [5] was designed for writing robust smart contracts with the following features: caller capabilities, safe asset transfer operations, and default immutability, so as to enforce the writing of safe and predictable code. The Obsidian language [6] with components, including typestate-oriented programming and resource type integrated into OO-style language, makes it easier for programmers to write correct program. The Scilla [7] is an intermediate level language to provide a clean separation between the communication aspect of smart contracts on blockchain and a programming component. The former allows for the rich interaction patterns, and the latter enjoys principled semantics and is amenable to formal verification.

In contrast with the above work, it is still relatively less for the research on the conversion from real-world contracts to programming language by establishing high-level or advanced smart contract language models. Several researches related to ASCL will be presented as follows. O'Connor [8] proposed a functional language called Simplicity, which provides static analysis to derive upper bounds on the computational space and time resources needed in Bitcoin Script and Ethereum's EVM prior to execution. This work is conducive to solving the prepayment problem of smart contract.

Regnath et al. [9] adopted natural language to propose a new kind of smart contract language called SmaCoNat in 2018. Some additional mechanisms, e.g., limiting the use of user-defined variable names and nested structures, sectioning the code, expanding the basic data types, defining natural language syntax, and unifying identity representation, were used to significantly enhance the readability and security of the smart contracts. Choudhury et al. [10] provided a new framework of automatic generating smart contracts. This framework adopted several semantic rules to encode knowledge in a specific domain, and then used an abstract syntax tree to incorporate the required constraints. Finally, the syntax of the constraints was encoded as a smart contract on blockchain. Biryukov et al. [11] introduced a new smart contract language called Findel, which mainly describes the action of currency transfers and the expression of multiplication, logic and time series from the perspective of financial engineering. However, Findel only contains two basic actions (Zero and One), two multiplication expressions (Scale and ScaleObs), three logical expressions (And, Or and If), and one time expression (Timebound), so its functionality is slightly single for various contracts.

Another exploratory research was a legal-oriented advanced smart contract language, called SPESC [12], that is used to solve the problem of hard-to-understanding for non-computer personnel. The SPESC language was similar to natural language with syntax elements in the real-world legal contract, as the first smart legal contract language. Moreover, the SPESC contract structure consists of four parts: contract name, parties, terms, and additional information, so as to make it closer to a real-world contract. Meanwhile, the additional information was employed to record the important data and the change of contract states by writing them into the tamper-proof blockchain.

In comparison with other languages, SPESC has an easy-to-read syntax, a clear structure, and a complete definition of syntax model, so that it might conform to the development direction of the future smart legal contracts. Moreover, the SPESC language is closer to the requirements for advanced smart contract languages in this paper. However, the issue of no generator to translate the SPESC instance into an executable smart contract code will directly influence its practical application, so that we will continue to study this issue on SPESC.

We next turn our attention to the work on generation of smart contract codes. Due to the fact that the researches of smart contracts are currently in the stage of language exploration, there is little work on automatic conversion or generation of contract codes. Then, two recent researches on this field will be introduced as follows.

Hamdaqa et al. [13] proposed a reference model (iContractML) and its modeling language for building smart contracts in a graphical way. Similar to Unified Modeling Language (UML), the iContractML can support five graphical objects, including participants, assets, transactions, conditions, and relationship. Moreover, they provided a conversion method from the reference model to three major platforms, i.e., Ethehereum, Microsoft Azure, and Hyperledger Composer. This kind of conversion is also relatively straightforward and simple since the five graphical objects are close to the objects in the programming language. In addition, this model and its modeling language is used as a UML-like model for smart contract design, but the model does not comply with the form of legal contract to write a text recognized by law.

Wöhrer et al. [14] proposed a Contract Modeling Language (CML), which supports a syntactic structure, similar to SPESC, with common contract-specific concepts, e.g., *party, asset, transaction, event, function,* and *clause constraints*. Also, the coarse conversion method was given from CML to Solidity, including that *party, asset* and *transaction* are converted into structure in Solidity, *event, function* and *clause constraints* are mapped to function, especially *clause constraints* are mapped to function modifier with conditional checks. However, this paper merely draws simple diagrams to illustrate conversion, and does not give specific conversion rules. The proposed method in SPESC-Translator is also suitable for CML language due to the similar structure between them. In addition, the mechanism, dealing with PullPayment pattern and supporting floating-point implementation, can be referred to as a supplement to our work.

In summary, the closer the grammar of smart contract language is to natural language in legal contract, the richer the expression of intention, and the more complex it is to convert into executable codes. On the contrary, the closer the smart contract language is to the programming language, the simpler the conversion, but the more difficult it is for non-computer personnel to understand.

Contribution. Aiming at the problem of lack of conversion methods from ASCL to executable smart contract language, in this paper we design an executable code generator for SPESC language through the instantiation of common contracts. Taking Solidity as a target programming language, we establish the conversion relationship from SPESC to executable program. The presented generator will be able to simplify the writing of smart contracts, standardize the program structure of smart contracts, and assist programmers to verify the correctness of code. The main work is listed as follows:

1) We propose a three-layer smart contract framework, which consists of advanced smart contract, general smart contract, and executable machine code. The general features of real-word legal contract, e.g., *party description, right and obligation, contract terms with time limit and manner on enforcement, contract conclusion*, are introduced to formal grammar of advanced smart contract layer. Moreover, benefiting from asset transaction model built on $deposit, transfer, withdraw$ from contracting accounts, the three-layer framework provides a legitimate approach to circulate digital assets by advanced smart contract languages in accordance with real-world legal contracts.
2) We design a series of conversion rules and program structure from SPESC to target programming language, e.g., Solidity. These rules provide an approach to generate party sub-contracts of target language according to the definition of SPESC contracting parties and their actions, as well as to produce main sub-contract of target language according to the terms of SPESC contract. In addition, several important mechanisms, e.g., party personnel management, program timing control, exception detection, etc., are integrated into the target program during the conversion from SPESC to Solidity. These mechanisms can assist programmers in semi-automatically writing smart contract programs.

Taking four SPESC contracts as our test objects, we provide the whole process (including coding, deploying, running and testing) of the conversion rules from SPESC contracts to executable programs (average 85% conversion rate and 5.9 times production ratio) in Solidity. Moreover, the converted contracts went straight through several vulnerability detection tools for smart contracts, except for three repairable ones.

Organization. The smart contract framework containing ASCL is introduced in Section 2. The syntax of SPESC is described in Section 3. The auction process is analyzed in Section 4. We show how to write an auction contract through SPESC in Section 5. Next, the SPESC compilation rules is presented on the target language Solidity. In Sections 7 and 8, we deploy and test the generated Solidity smart contracts, and then make a conclusion.

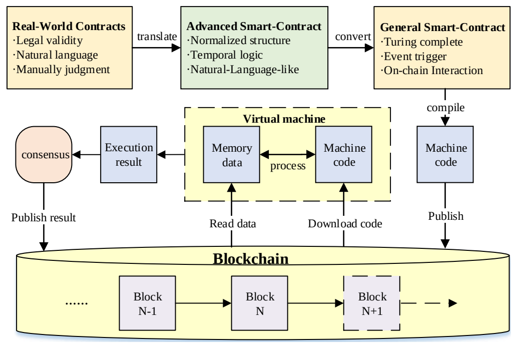


Fig. 1: Advanced smart contract framework.

2 Framework of Smart Contract System

2.1 System Goals

As smart contract is considered as a self-executing contract, blockchain-based smart contract system generally refers to a system that supports the contract's programming, compiling and automatic execution. The existing smart contract systems generally have two layers: general smart-contract layer (GSCL) for contract's programming and compiling, and executable machine-code layer (EMCL) for contract's automatic execution.

In this paper, a new advanced smart-contract layer is placed on the top of these two layers, so that a three-layer smart contract framework is proposed as follows:

- *advanced smart-contract layer* enhances the normalization and legibility of contracts.
- *general smart-contract layer* helps generate the executable contracts by using the existing smart contract language, like traditional programming language.
- *executable machine-code layer* ensures the automatic execution of these contracts.

With the help of the advanced smart-contract language, this new layer provides a higher contract-oriented encapsulation than the general smart contract language. The relationship between these two languages is similar to that between high-level and low-level programming languages, such as SQL and C++, or Rython and Java. Exactly, the following requirements should be fulfilled on this framework:

- The contract should be easier to read and understand on advanced smart contract language;
- Advanced smart contract language should simplify and standardize the programming of smart legal contracts;
- A translator should be developed to generate the program framework of smart contracts on target language.

2.2 Framework

In Fig. 1, we illustrate a common workflow by which contracting parties are allowed to develop smart contracts on the proposed three-layer framework. In this framework, the workflow can be described as follows:

1) A user starts with an instance (advanced smart contract or smart legal contract) of ASCL in terms of real-world contract or intention of contracting parties.
2) The instance can be translated into a program (general smart contract) written in the existing smart contract language through advanced smart contract language translator (in short translator), so that the program is compiled in-

to the target code executed on blockchain's virtual machine through traditional programming language compiler.

3) When an external event or internal transaction meets the logical conditions of contract terms, both the target code and running state are loaded into the virtual machine and then triggered, and the final results (including value and state) will be written back into the blockchain.

Furthermore, we illustrate the main entities and properties involved in this workflow as follows.

2.2.1 Real-world contract

Real-world contracts are legally valid after they are established and signed in compliance with law. The written form of real-world contract is relatively free, and there is no strictly prescribed format. Even if there are ambiguities or defaults in contracts, the justice institutions can make a legal decision from true intentions of contracting parties. In comparison with current contracts, real-world contracts have the following features:

- They are legally valid;
- Using natural language and free format;
- Once a dispute arises, judge can reach a legal decision.

2.2.2 Advanced smart contract

Derived from real-world contract, the advanced smart contract tends to integrate the features of computer programs, laws and finance together. It can express contract intentions in a way that is easier to understand than the programming language and more standardized than natural language. This kind of contract adheres to the grammatical rules in the advanced smart contract language, e.g., SPESC, which has the following features:

- Refer to the structure of real-world contract;
- Express sequential relationship among terms through temporal logic;
- Provide the natural-language-like grammar.

There have existed many discussions on smart contracts from a legal perspective. Kasprzyk [15] discussed the present legislation with regard to the legal definition of smart contract, especially actual and potential conflicts between smart contract and established principles of contract law. Goldenfein et al. [16] discussed the emergence of smart contracts as legal conduct around blockchain platforms and automated transactions. Allen et al. [17] analyzed smart contracts from a natural language perspective. In addition, Gomes [18] applied the theoretical and methodological framework of the Economic Analysis of Law to investigate the characteristics and possibilities of adopting smart contracts under the perspective of the Brazilian legal order and its insertion in the Creative Economy.

2.2.3 Smart contract

The existing smart contract language is similar to the traditional programming language, but special elements or operations related to the blockchain are designed or predefined on it. Moreover, event-triggered mechanism is adopted to help smart contract express the interaction process among multiple parties. So, unlike traditional agreement mechanism that rely on human intervention or approval to execute the intended function of the contract, the deployed code of smart contract has no choice but to execute the triggered functions automatically. In brief, the current mainstream smart contract languages, such as Solidity, Java, GO, etc. have the following features:

- Constructed on Turing-complete language with powerful expression ability;
- Triggered by external events;
- Predefine special elements or operations related to the blockchain.

2.2.4 Machine code and execution

After a smart contract is compiled to generate machine code, it may be deployed to the blockchain. And then, the deployed code will be run in a virtual machine or a container (docker) once it is triggered by external events or internal transactions. After execution, the transaction data and program states are written back to the blockchain, and the blockchain's consensus protocol is able to ensure that they are always consistent across all nodes. In short, the deployment and execution of the machine code are mainly divided into the following steps:

- **Deploy smart contracts:** publish the compiled machine code to the blockchain, so that other nodes participating in the consensus can obtain them for verification;
- **Execute smart contract:** restore the contract state by downloading code and data stored in blockchain into memory, and then run them in a local virtual machine or container;
- **Publish the execution results:** after running the contract to response external events or internal transactions, the execution result will be agreed with other consensus nodes, and finally upload to the blockchain.

3 SPESC SYNTAX

In this section, we turn our attention to the syntax of SPESC language. The SPESC refers to the common structure of real-world contract, which consists of four parts: title, parties, terms and additional informations, as follows:

Definition 1 (Contract) *Contract in SPESC is defined as:*

Contract::= *Title*{***Parties+ Terms+ Additional+***}

where + denotes multiple entries are allowed.

Generally, a contracting party is one who holds the obligations and receives the benefits of a legally binding agreement. There are at least two parties involved in a contract, and sometimes a third party beneficiary may be named. Also, the parties may be individuals, organizations, or groups.

The main attributes of parties should be recorded in the $Parties$ of contract, as follows:

Definition 2 (Parties) *Parties in SPESC is defined as:*

Parties::= ***party group***? *PartyName* {*Field+ Action+*}

where ? denotes the keyword is optional.

In the above definition, the optional keyword "group" indicates the parties are a group. The entry $Field$ denotes a list of attributes of parties, and the entry $Action$ states the parties' rights and obligations in the contract.

Single party refers to individual who has certain rights or obligations in a contract, such as buyer or seller. Party group refers to multiple individuals with the same rights and obligations in a contract, such as voters, bidders, etc. In addition, party group can not only be appointed in advance before the execution of contract, but also be joined or withdrawn dynamically during the execution of contract.

The definition of parties is to make convenient for processing and recording the parties' information. Each individual has a corresponding account address in blockchain, so that the address is included into the attributes of party by default. Moreover, the address can not only be set to specify the party's identity when signing the SPESC contract, but also be changed based on the contract terms according to runtime conditions.

The contract terms are divided into two types: right terms and obligation terms. Exactly, a right term indicate an action that can be performed under certain conditions, and a obligation term indicate an action that must be completed under certain conditions. Actions, not written in the contract, represent that they will be forbidden. In SPESC, the contract terms are defined formally as:

Definition 3 (Terms) *Terms in SPESC is defined as:*
Terms::= ***term*** *tname: PartyName* (***shall*** | ***can***) *action*
(***when*** *PreCondition*)?
(***while*** *AssetTransaction+*)?
(***where*** *PostCondition*)?.

In the above definition, $PartyName$ indicates the party name defined in $Parties$, and $action$ refers to the action to be performed. In addition, $PreCondition$ represents the preconditions that should be satisfied before the execution of term, $AssetTransaction$ indicates the asset transactions accompanying the execution of term, and $PostCondition$ represents the post-conditions that should be satisfied after the execution of term. The basic manner of distinguishing pre-condition from post-condition is that the business logic of contract can be expressed by pre-condition and time expression, but be restricted by post-condition in case of unexpected situation.

3.1 Asset Transaction

In the contract term, the asset transaction, shortened to $AssetTransaction$, proclaims the special operation used to transfer an asset. In order to inspect and trace the process of smart contract, all assets must be transfered through the account associated with the contract, called **contracting accounts**. For example, an asset transaction from A to B requires that the asset is transfered from A to the contracting account, and then transfered from the contracting account to B. Through the above method, the contract is convenient for checking the transfer conditions and the amount according to the term's $AssetTransaction$, and the transfer information is recorded during the execution of contract. Therefore, the asset transactions should be divided into two types: deposit user's asset to contracting account, and transfer the asset in contracting account to user's account. as shown in Fig. 2 (a).

For the sake of asset's security, the operation of transferring assets from user's account to contracting account can only be performed on his own initiative rather than compulsory enforcement. Thus, it should be necessary to distinguish the executor account from other accounts as shown in Fig. 2 (b). The operations of transferring assets in $AssetTransaction$ can be divided into three parts:

- **deposit**: the assets are deposited actively from the executor into the contract;
- **withdraw**: the assets are withdrew from the contract to their owner according to the contract terms;
- **transfer**: the assets are transfered from the contract to another account according to the contract terms.

Specifically, the $AssetTransaction$ is defined as:

Definition 4 (AssetTransaction) *Asset transactions in the SPESC is defined as:*
AssetTransaction::=
{*Deposit*} ***deposit*** (***value*** *ROP*)? *AssetExp*
|{*Withdraw*} ***withdraw*** *AssetExp*
|{*Transfer*} ***transfer*** *AssetExp* ***to*** *Target*

In the above definition, ROP represents the relational operation, including $>$, $<$, $=$, $>=$, and $<=$, e.g., deposit value $>=$ \$10. $AssetExp$ represents the asset expression, which is used to describe the transferred asset, and $Target$ represents the target account for transferring asset, e.g., transfer \$10 to Buyer.

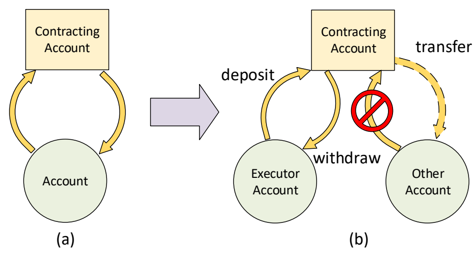


Fig. 2: Different operations in asset transaction.

In three types of asset operations, the deposit operation is different from withdraw and transfer operations because the latter must have been agreed in advance. This also means the withdraw and transfer operations need to be accurately described by using constants or variables in contract. However, it is not necessary for the deposit operation to directly stipulate the asset under the contract considering that the user must performs the deposition actively. For example, the contract described in Section 5 can limit to the current highest price rather than directly specify the price that the bidders deposit.

As far as implementation is concerned, the asset operations must depend on smart contract platform and target language, e.g., there is no restriction on the account's type in the Ethereum platform which may be either a contracting account or a user's account. Moreover, considering that the contract in Solidity can define a default function to deal with the received assets (Ether), it is necessary to prevent the security risks from transferring assets to unknown accounts. Therefore, the contracting parties would better replace the transfer operation by actively withdrawing asset. For example, assume A intend to transfer a pledged loan from contracting account to B, it is always secure to let the recipient B withdraw his loan himself.

4 Auction

In this section, we will illustrate the SPESC generator through an auction contract instance, so as to confirm the generated contract, further to affirm the availability of three-layer smart contract framework. Auction is a kind of spot trading mode where the institution plays an important role to accept the entrustment of owner. The institution also displays the auctioned goods to buyer at the specified time according to certain rules, makes public bidding, and finally sells the goods to the buyer who conforms to rules. This paper mainly addresses the auction contracts with the highest price. In tis paper, the smart contract is referred to as the institution specialized in auction business.

The auction contract involves two parties: auctioneer who is an enterprise legal person engaged in auction activities, and bidders who are some citizens, legal persons or organizations to participate in bidding for auction targets. As shown in Fig. 3, the highest bidding process is described as the following steps.

- The auctioneer starts the auction system after setting a reserve price and an auction end time, and waits for the auction to end;
- The bidders can bid anytime during the system. If the bid is greater than the current highest price, the system records it as new highest price, puts the bid into the fund pool, and returns the bid paid by the previous highest bidder; otherwise, the bidder fails and the bid is returned;
- After the auction time is over, the auctioneer can collect the highest bid from the fund pool.

5 SPESC-based Auction Contract

According to the above-mentioned auction process, the SPESC-based contract mainly consists of three parts: parties, terms, and additional information, which are described as below.

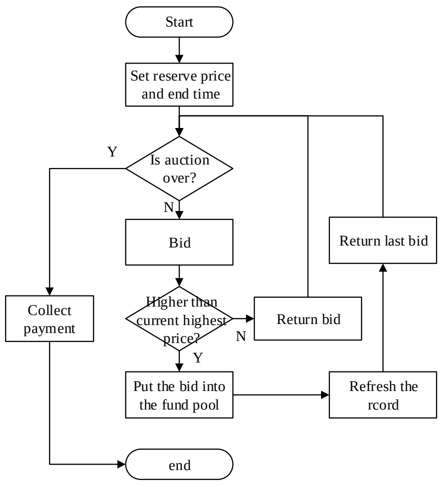

Start
Set reserve price and end time
Is auction over?
Y
N
Bid
Higher than current highest price?
N
Y
Return bid
Return last bid
Collect payment
Put the bid into the fund pool
Refresh the rcord
end

Fig. 3: The auction process.

5.1 Parties

As mentioned in Section 4, the auction contracts include two parties: the bidders and the auctioneer. At first, the statement of bidders is defined as follows:

```
party group bidders{
  amount : Money
  Bid()
  WithdrawBid()
}
```

The bidders is a group and contains an attribute $amount$ with Money type. This attribute is used to record the bidder's fund pool. The $Bid()$ operation declares that the bidder has right to deposit money and bid for the goods. Moreover, the $WithdrawBid()$ operation is to refund invalid bids.

```
party auctioneer{
  StartBidding(reservePrice : Money,
    auctionDuration : Date)
  CollectPayment()
}
```

The auctioneer indicates an individual who is the owner of the goods. There are two operations, $StartBidding()$ and $CollectPayment()$, that the auctioneer can perform. Exactly, when performing the $StartBidding()$, the auctioneer need to enter two parameters: reserve price and auction end time.

5.2 Additional Information

There are three variables defined in this contract:

```
highestPrice : Money
highestBidder : biddersBiddingStop
BiddingStopTime : Date
```

The first variable $highestPrice$ is the Money type which records the current highest price, the second variable $highestBidder$ is the bidders type to record the current highest bidder, and the third variable $BiddingStopTime$ is the Date type to record the end time of the auction.

The additional information will be recorded in blockchain to ensure that the blockchain records not only the current state of contract, but also the history of the step-by-step execution process. Since blockchain stores data in a sequential and unchangeable way, it guarantees the immutability and traceability of execution states in smart contract. Therefore, it is necessary to define the key variables in the additional information.

5.3 Contract Terms

In Fig. 4, we show a SPESC-based aucton contract that includes five terms to specify the auction rules. In terms of the procedure in Section 4, there are three actions, start auction, bid, and collect payment, that must be triggered actively by the participants. These actions will be converted into five terms.

```
contract SimpleAuction{
    party group bidders{
        amount : Money
        Bid()
        WithdrawBid()
    }
    party auctioneer{
        StartBidding(reservePrice : Money,acutionDuration: Date)
        CollectPayment()
    }
    highestPrice : Money
    highestBidder : biddersBiddingStop
    BiddingStopTime : Date

    term no1 : auctioneer can StartBidding,
        when before auctioneer did StartBidding
        where highestPrice = reservePrice and BiddingStopTime =
                acutionDuration + now.

    term no2 : bidders can Bid,
    when after auctioneer did StartBidding and before BiddingStopTime
        while deposit $ value > highestPrice
        where highestPrice = value and highestBidder = this bidder and
           this bidder : : amount = this bidder: :0ri amount + value .

    term no3_1 : bidders can WithdrawBid,
        when this bidder isn't highestBidder and this bidder: :amount >0
        while withdraw $ This bidder : : amount
        where this bidder : : amount = 0.

    term no3_2 : bidders can WithdrawBid,
        when this bidder is highestBidder and this bidder : : amount >
                  highestPrice
        while withdraw $ this bidder : : amount - highestPrice
        where this bidder : : amount = highestPrice.

    term no4 : auctioneer can CollectPayment,
    when after BiddingStopTime and before auctioneer did
              CollectPayment
        while withdraw $highestPrice.
       }
```

Fig. 4: SPESC-based auction contract.

As shown in Fig. 5, a Petri nets is used to illustrate the state transition diagram of the auction process. From this figure, it is easy to see that the contract consists of four states (represented by a circle) and five actions (by a segment). The states includes: Active, Bidding, BiddingEnd, ContractEnd, and the actions include StartBidding, Bid, WithdrawBid, TimeOut, and CollectPayment. Moreover, the action Timeout will be triggered automatically by the contract system, and four remaining actions are triggered by extern events derived from the user.

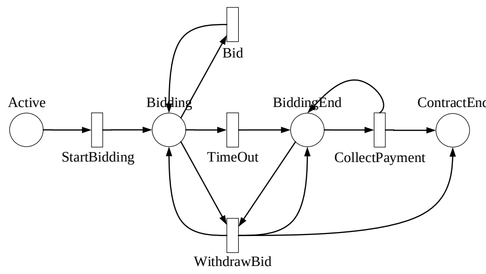


Fig. 5: State transition of auction on Petri net.

In terms of four above-mentioned actions, the contract terms are written as follows.

Term 1. The term corresponding to the action StartBidding is used to initial the contract, which is defined as follows:

```
term no1 : auctioneer can StartBidding,
    when before auctioneer did StartBidding
  where highestPrice = reservePrice and
    BiddingStopTime = auctionDuration + now.
```

This term indicates that only $auctioneer$ has the right to trigger the action for initiating the auction ($StartBidding$). After the action is executed, the current highest price ($highestPrice$) shall be set to the reserve price ($reservePrice$) entered by auctioneer, and the end time ($BiddingStopTime$) shall be the current time (now) plus the auction duration ($auctionDuation$).

Term 2. The term corresponding to the action Bid is defined as follows:

```
term no2 : bidders can Bid,
  when after auctioneer did StartBidding and
    before BiddingStopTime
  while deposit $ value > highestPrice
  where highestPrice = value and
    highestBidder = this bidder and
    this bidder::amount = this bidder::Origin amount +
     value.
```

The bidders can bid to the contract by executing this term (Bid) after the auctioneer initiates the auction and before auction end. Among them, the condition "$after\ auctioneer\ did\ StartBidding$" expresses the time after auctioneer initiates the auction, and "$before\ BiddingStopTime$" indicates the time before the end of auction. If the bid (expressed by $value$) is greater than the current highest price ($hightestPrice$), the action Bid will succeed; otherwise, it fails.

After the term is executed, the highest bidder's attribute ($this\ bidder :: amount$) should record the sum of new highest value and the amount of failed bids. For convenience, the keyword $Origin$ represents the value before the term is executed, so that the amount of failed bids is obtained by $this\ bidder :: Origin\ amount$. The current highest price and the highest bidders should be set to the current bid ($value$) and the bidder ($this\ bidder$).

Term 3. As described in Section 5, the transfer refunding an invalid bid has a big security risk in that any account may be registered as a bidder on target language Solidity. Therefore, it is more secure for recipients to withdraw their own money. The $WithdrawBid$ will be added when the contract was written.

The $WithdrawBid$ term can be divided into two sub-terms according to the fact whether or not the bidder is the highest Bidder. The first sub-terms is defined as:

```
term no3_1 : bidders can WithdrawBid,
  when this bidder is not highestBidder and
    this bidder::amount > 0
  while withdraw $this bidder::amount
  where this bidder::amount = 0
```

This term indicates that the bidder can withdraw the bid if he is not the highest bidder and there is a deposit in the current account ($this\ bidder :: amount > 0$). After the withdraw operation is successfully executed, the bidder's deposit record should be cleared. Furthermore, the second sub-terms is defined as:

```
term no3_2 : bidders can WithdrawBid,
  when this bidder is highestBidder and
    this bidder::amount > highestPrice
  while withdraw $this bidder::amount - highestPrice
  where this bidder::amount = highestPrice
```

This term indicates that the bidder can withdraw the bid if he is the highest bidder and there is a failed bid in the current account ($this\ bidder :: amount > highestPrice$). After the withdraw operation is successfully executed, the bidder's deposit record should be current highest price.

Term 4. The term corresponding to the action StopBidding is defined as follows:

```
term no4 : auctioneer can CollectPayment,
  when after BiddingStopTime
    and before auctioneer did CollectPayment
  while withdraw $highestPrice.
```

This term indicates that the auctioneer is allowed to stop bidding for collecting the payment (by taking out the highest price) when the auction time is over and the auctioneer has not enforced this term.

6 Target Code Generation

Through the example of auction contract, we describes how to generate the target code of SPESC-based contracts. The target code will be built in Solidity that is an object-oriented programming language for the Ethereum Virtual Machine (EVM) [19], [20]. More importantly, we require that the code generation should be a (semi-)automatic process, called SPESC translator, for conversing SPESC contracts to Solidity codes.

The SPESC-Translator offers a program architecture of the converted contracts with one main-contract and some party-contracts. Specifically, these two contracts are described as

- **Main-contract** corresponds to the principal part of SPESC-based contract which includes the definition of variables, modifiers and auxiliaries, as well as the description of functions derived from terms.
- **Party-contract** is a party-oriented contract which is mainly responsible for the management of party's personnel data, the response of events, and the query of history records.

Taking the previous SPESC contract as an example, we will describe how to generate these two kinds of contracts.

6.1 Framework of Target Code

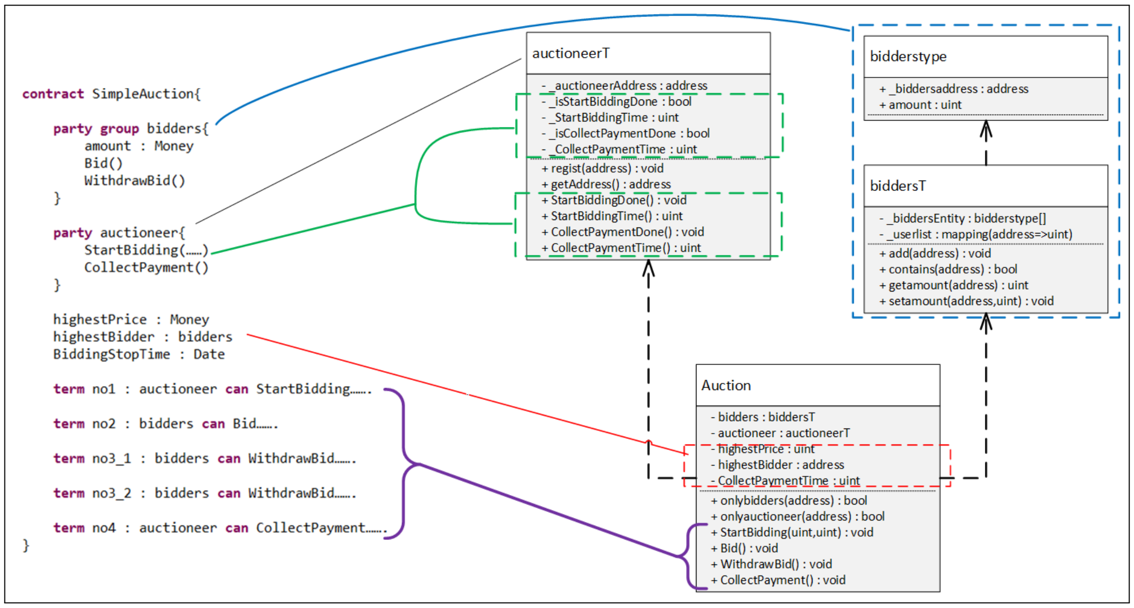
As shown in Fig. 6, we show the class structure of auction contract in Solidity language automatically generated by the SPESC translator. Moreover, the color lines are used to illustrate the conversion regulation from the SPESC contract to the Solidity code. On the whole, the generated contract in Solidity consists of three parts: an auctioneer's party-contract (called **auctioneer**), a bidder's party-contract (**bidders**) and a term-derived main-contract (**auction**).

Firstly, derived from the auctioneer's definition in SPESC, the auctioneer contract contains the account address of the auctioneer, registration and query functions for the personnel attributes, and some records of two terms, $StartBidding$ (no1) and $CollectPayment$ (no4). The reason of placing these two terms here is that they will be performed by the auctioneer.

Secondly, converted from the bidders' definition in SPESC, the bidders contract is a group-oriented party-contract that includes a new structure, called **bidderstype**, consisting of an address and the variable 'amount' defined in SPESC, in order to manage the dynamic bidders. Based on this, the bidders contract contains a structure array, a mapping table, functions for personnel joining and query, interface with getter and setter of the amount, and so on.

Finally, derived from all terms in the SPESC contract, the generated main-contract contains the definitions of parties, three additional information ($highestPrice$, $highestBidder$, $BiddingStopTime$), two modifiers ($onlybidders$ and $onlyauctioneer$) and four functions generated from five terms. The reason of four functions instead of five ones is that both terms no3_1 and no3_2 declare the same action $WithdrawBid$, so only one function is generated.

In addition, the SPESC-Translator can provide many operations for party management in the process of translation. Some unused management operations, such as obtaining a list of party groups, deleting individuals, will be provided in the form of annotations, but they can be uncommented if the programmer want to use them. For the terms that do not participate in timing control, such as Bid and $WithdrawBid$, the attributes and functions related to the execution states will not be generated in the target code. However, the actual execution states can still be obtained and traced from the blockchain.



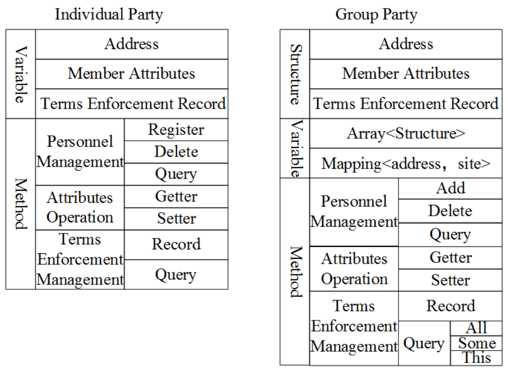
```
contract SimpleAuction{

    party group bidders{
        amount : Money
        Bid()
        WithdrawBid()
    }

    party auctioneer{
        StartBidding(……)
        CollectPayment()
    }

    highestPrice : Money
    highestBidder : bidders
    BiddingStopTime : Date

    term no1 : auctioneer can StartBidding…….

    term no2 : bidders can Bid…….

    term no3_1 : bidders can WithdrawBid…….

    term no3_2 : bidders can WithdrawBid…….

    term no4 : auctioneer can CollectPayment…….
}
```

auctioneerT
- _auctioneerAddress : address
- _isStartBiddingDone : bool
- _StartBiddingTime : uint
- _isCollectPaymentDone : bool
- _CollectPaymentTime : uint
+ regist(address) : void
+ getAddress() : address
+ StartBiddingDone() : void
+ StartBiddingTime() : uint
+ CollectPaymentDone() : void
+ CollectPaymentTime() : uint
bidderstype
+ _biddersaddress : address
+ amount : uint
biddersT
- _biddersEntity : bidderstype[]
- _userlist : mapping(address=>uint)
+ add(address) : void
+ contains(address) : bool
+ getamount(address) : uint
+ setamount(address,uint) : void
Auction
- bidders : biddersT
- auctioneer : auctioneerT
- highestPrice : uint
- highestBidder : address
- CollectPaymentTime : uint
+ onlybidders(address) : bool
+ onlyauctioneer(address) : bool
+ StartBidding(uint,uint) : void
+ Bid() : void
+ WithdrawBid() : void
+ CollectPayment() : void

Fig. 6: Correspondence between SPESC and the generated contract class.

6.2 Generation of Party-Contracts

In the conversion process, the SPESC-Translator generates a Solidity party-contract for each contracting party defined in SPESC contract. According to the different parties, the party-contracts are divided into two types: individual party-contract and group party-contract, as shown in Fig. 7.



Individual Party

Variable	Address	
	Member Attributes	
	Terms Enforcement Record	
Method	Personnel Management	Register
		Delete
		Query
	Attributes Operation	Getter
		Setter
	Terms Enforcement Management	Record
		Query

Group Party

Structure	Address		
	Member Attributes		
	Terms Enforcement Record		
Variable	Array<Structure>		
	Mapping<address, site>		
Method	Personnel Management	Add	
		Delete	
		Query	
	Attributes Operation	Getter	
		Setter	
	Terms Enforcement Management	Record	
		Query	All
			Some
			This

Fig. 7: The structure of individual and group party-contracts.

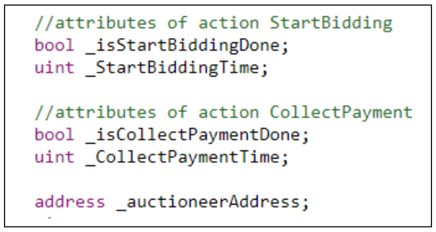
6.2.1 Case 1: Individual party-contract for the auctioneer

As shown in the left of Fig. 7, the individual party-contract utilizes a variety of variables and methods. The variables store an arbitrary piece of data, including account address, member attributes, term enforcement records, for later use. The methods specify some functions to manage the personnel information and term's execution, e.g., personnel management, attributes operation, and term enforcement management. For example, the auctioneer party-contract consists of three following parts.

Case 1.1 Member Variables. Each individual party-contract will record the following attributes: account address, member attributes, and term's execution records. The account address, used as unique identity, refers to the blockchain address corresponding to the account. The member attribute is a set of attributes defined in the SPESC contract. The term's execution records is the journal that meet generation rules: for each term t, the record is formal defined as

$$Record ::= \{<\_is \ll t \gg Done, \_ \ll t \gg Time >\},$$

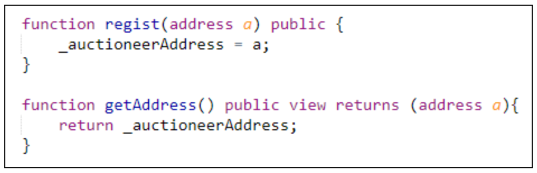
where $\ll t \gg$ represents the action name of term t, and the variable name starts with an underline to distinguish it from user-defined variables. For example, in the action $StartBidding$ corresponding to term no1, two variables will be generated according to the above rules. Moreover, $\_isStartBiddingDone$ is used to record whether the execution of term t is completed, and $\_StartBiddingTime$ records the execution time of term t.



```
//attributes of action StartBidding
bool _isStartBiddingDone;
uint _StartBiddingTime;

//attributes of action CollectPayment
bool _isCollectPaymentDone;
uint _CollectPaymentTime;

address _auctioneerAddress;
```

Fig. 8: Auctioneer's attributes in Solidity.

In Fig. 8, we show an example of auctioneer's attributes derived from the SPESC-based auction contract. Although the auctioneer does not have attributes in the SPESC contract, there are two related terms and actions, $StartBidding$ in term no1 and $CollectPayment$ in term no4. Thus, the SPESC translator follows the above-mentioned generation rules to generate party's attributes for each term, e.g., $\_isStartBiddingDone$ and $\_StartBiddingTime$ for $StartBidding$ in term no1. In addition, the SPESC translator defines the variable $\_auctioneerAddress$ to record the auctioneer address.

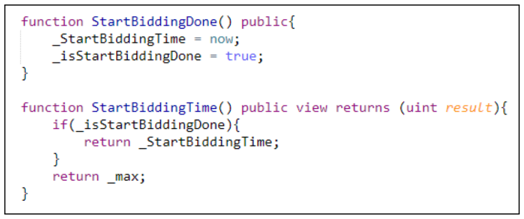


```
function regist(address a) public {
    _auctioneerAddress = a;
}

function getAddress() public view returns (address a){
    return _auctioneerAddress;
}
```

Fig. 9: Auctioneer personnel management in Solidity.

Case 1.2 Personnel Management. According to parties who are groups or individuals, corresponding personnel management methods should be generated. Personnel management including $Register$, $Delete$ and $Query$ is relatively simple in comparison with group personnel management. In the auction, personnel management of auctioneer just involves registering and querying the auctioneer's address, as shown in Fig. 9.

Case 1.3 Term Enforcement Management. For each term associated with contracting party, term enforcement management contains some term-related functions that allow the programmer to handle the terms and provides the appropriate feedback. The code generated from term no1 is shown in Fig. 10.



```
function StartBiddingDone() public{
    _StartBiddingTime = now;
    _isStartBiddingDone = true;
}

function StartBiddingTime() public view returns (uint result){
    if(_isStartBiddingDone){
        return _StartBiddingTime;
    }
    return _max;
}
```

Fig. 10: Auctioneer's term execution management.

In this example of $StartBidding$, the SPESC translator generates a record function $StartBiddingDone$ after term enforcement and a time query function $StartBiddingTime$ for term completion. Through these two functions, the auctioneer's attributes have been automatically specified.

6.2.2 Case 2: Group party-contract for the bidders

The group party-contract has more complex structure than individual one in order to support dynamic parties. As the right subfigure of Fig. 7, the group party-contract also includes rich functions for personnel management (see $Array$ and $Mapping$ structure) and various functions for record query (see All, $Some$, and $This$ query in terms enforcement management) in addition to the contents of individual party-contract. The specific generation rules are as follows.

Case 2.1 Personnel Management. In order to deal with a group of contracting parties, the array structure is adopted to store the parties' information in a group-oriented contract. Meanwhile, a mapping table from account addresses to array coordinates is used to facilitate query on the individual information. Besides, as shown in Fig. 7, the party's attributes recorded in a structure that is the same as the member attributes of individual party-contract.

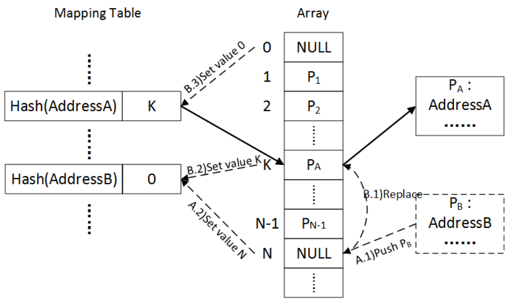
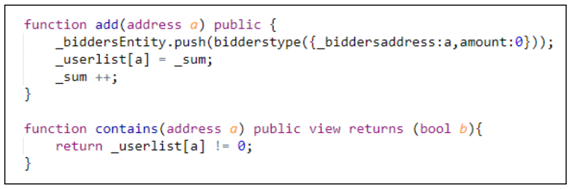


Fig. 11: Mapping table and array for group-oriented contracting party management.

In Fig. 11, we show how to use the mapping table and array to manage a group of dynamic contracting parties. Firstly, assume a new party individual P_B be appended into the contract, the Add operation in the 'Personnel Management' will perform the following steps: A.1) insert the individual information P_B into the last of array which has the coordinate N; and A.2) record the coordinate N on the mapping table with hash index of individual address $AddressB$, where the mapping table includes a pair of address hash and array coordinate. The code of add operation is shown in Fig. 12, where $\_biddersEntity$ and $\_userlist$ represent the array and the mapping table, respectively. An additional operation $contains$ is generated to check whether a given account is in the mapping table.



```
function add(address a) public {
    _biddersEntity.push(bidderstype({_biddersaddress:a,amount:0}));
    _userlist[a] = _sum;
    _sum ++;
}

function contains(address a) public view returns (bool b){
    return _userlist[a] != 0;
}
```

Fig. 12: Bidders' personnel management.

Next, we consider the $Delete$ operation for revoking an individual P_A from the group of parties. After querying its array coordinate K according to $AddressA$, the operation will proceed as follows: B.1) replace the content in the corresponding unit (K-th) of array with the individual P_B in the last (N-th) of array, and clear the N-th unit; B.2) replace the array coordinates recorded in the mapping table for P_B with the new coordinate K; and B.3) replace the coordinate of the P_A with the initial value 0. If the deleted individual is the last in the array, the operation directly sets the corresponding unit of the array and the mapping table to empty.

Case 2.2 Term Enforcement management. The other difference between group party-contract and individual party-contract is term enforcement management, where the former involves some new functions corresponding to the terms' states besides the $Record$ functions included in the latter. Exactly, these new functions consist of three queries:

- **'All' query:** which outputs the completion state of the last individual for a certain term, e.g., *after all voters did vote*;
- **'First' query:** which outputs the completion state of the first individual for a term, e.g., *before first buyers did pay*;
- **'This' query:** which outputs the completion state of the current individual for a term, e.g., *after this bidders did bid*.

For example, these functions can provide the query for the recorded time when the terms are completed by the first, last, or current person, respectively.

6.3 Main-Contract Generation

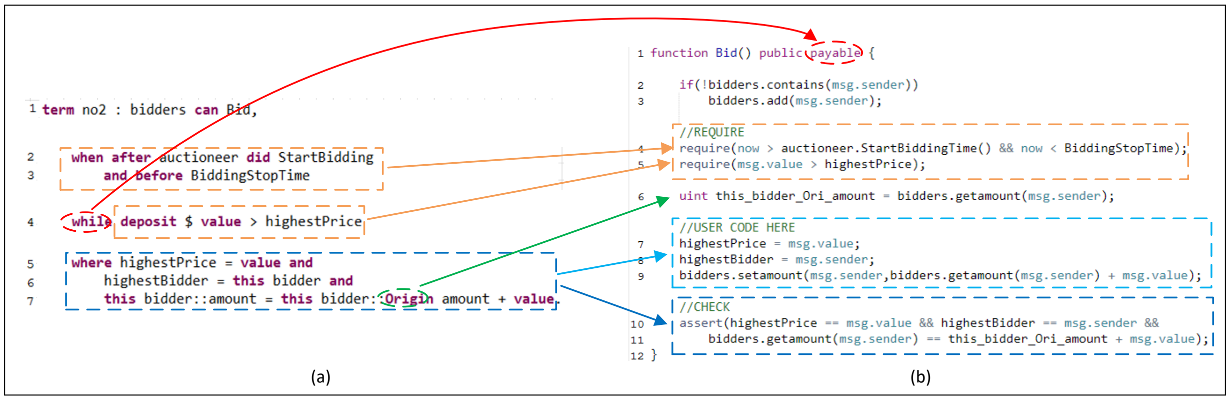
The generated main-contract is divided into two parts. The first involves the definition and initialization of contract attributes and parties. The contract attributes refer to the information defined by programmer in the original SPESC contract, and parties are some instances of party-contract classes as mentioned above. Among them, the default access permission of generated variables is public. Such that, the public variables in Solidity will be accessed directly through the contract to obtain the contract state. For example, a bidder can query $highestBidder$ in the contract to obtain the address of the highest bidder, so as to determine whether he/she has won the bid.

The second part is used to deal with the terms. Considering each term in the SPESC contract represents an action, a function corresponding to the term (called *term-derived function*) would be generated in Solidity. This kind of functions includes three parts: *execution condition*, *function body* and *result detection*. If the detection of the execution condition fails, the program will explicitly throw an exception to handle it at runtime in the process of execution condition and result detection.

6.3.1 Restriction detection in term implementation

In accordance with the syntax of the SPESC terms, there are three types of restrictions on the implementation of terms:

- **Participant restriction:** which is used to stipulate which of parties can perform the current term;
- **Conditional restriction:** which refers to the preconditions defined in the SPESC terms;



```
1 term no2 : bidders can Bid,

2    when after auctioneer did StartBidding
3        and before BiddingStopTime

4    while deposit $ value > highestPrice

5    where highestPrice = value and
6        highestBidder = this bidder and
7        this bidder::amount = this bidder::Origin amount + value.
```

```
1  function Bid() public payable {

2      if(!bidders.contains(msg.sender))
3          bidders.add(msg.sender);

       //REQUIRE
4      require(now > auctioneer.StartBiddingTime() && now < BiddingStopTime);
5      require(msg.value > highestPrice);

6      uint this_bidder_Ori_amount = bidders.getamount(msg.sender);

       //USER CODE HERE
7      highestPrice = msg.value;
8      highestBidder = msg.sender;
9      bidders.setamount(msg.sender,bidders.getamount(msg.sender) + msg.value);

       //CHECK
10     assert(highestPrice == msg.value && highestBidder == msg.sender &&
11         bidders.getamount(msg.sender) == this_bidder_Ori_amount + msg.value);
12 }
```

Fig. 13: The conversion relationship between the term no2 in SPESC and the term-derived function in Solidity.

- **Deposit restriction:** which refers to a limit on the amount of cash that be deposited into the contract when calling this function, and is embodied in the deposit statement in the transaction operation.

To detect the condition, Solidity has three functions, $require$, $assert$, and $revert$, designed for checking state changes to prevent potential errors [21]. The function $require$ can be used to guarantees validity of conditions, such as inputs, contract state variables, and return values, that cannot be detected before execution. If the detection fails, the Ethereum platform will return the remaining gas. The function $assert$ can be applied for internal errors or analyzing invariants. Once this kind of detection fails, it means that there is an error in program itself and the code should be fixed. Finally, the function $revert$ can be used to flag an error and revert the current call.

In the function translated from the SPESC term, the detection failure of these three restrictions is normal program state, so that the exception-throwing function, $require$ or $revert$, may be used to return the remaining cash and roll back to a previous state in the main-contract. Furthermore, the SPESC-Translator will generate the detection sentence of corresponding execution results according to the predefined $PostCondition$. These sentences will assist the programmer to write the *function body* and check run-time errors in the program. Once the detection fails, it means that there is an error in the program, so that the function $assert$ can be used to request for manual assistance and legal action for breach of contract.

Except for restriction detection, the function body can be acquired from $PostCondition$ of term by the SPESC-Translator, and the post-condition will be considered as a primary reference for programmers. Therefore, the programmer only needs to write the function body and verify the correctness of the execution for given inputs and results.

6.3.2 Example for generating term-derived function

Let us continue to take the auction contract as an example for illustrating the implementation of term-derived functions. In this example, the program logic is so simple that all the programmer need to do is to append the codes for parties' registration and check the logic of the codes after the main-contract is automatically generated by the SPESC-Translator.

In Fig. 13, we show how to convert term no2 into the code in Solidity, where the former is written in the left side (A) and the latter is in the right side (B). By using arrows in this example, we describe the conversion relationships between SPESC and Solidity. These relationships can be described as follows:

1) Generate the function Bid that has same name as the action Bid in the first line of A subgraph (denoted as $A1$). If it is detected that A contains an asset transaction statement, e.g., the '**while**' sentence in $A4$, the keyword '**payable**' must be appended to $B1$ in order to indicate that the function can receive or send Ether;
2) Accomplish the *participant restriction* for the '**bidders**' defined in $A1$. In this example, according to actual intention of the contract, the one who deposit the cash must be the bidder. Therefore, it is necessary to remove the participant restriction by the SPESC translator and manually add the party registration code (see B2-3): if the caller is not a bidder, he will be registered as a bidder;
3) Accomplish the *conditional restriction* for the '**when**' sentence according to the preconditions (A2-3). Two $require$-type functions are generated to guarantee the validity of the following conditions: one is that the bid time would be after the auctioneer starts the bidding and before the end time of auction ($A4$), and the other is that the bid would be greater than the highest price ($A5$).
4) Accomplish the *function body* corresponding to the '**where**' sentence. According to the post-conditions ($A5$-7), three lines of execution code ($B7$-9) are automatically generated to record the runtime states, including the highest price, highest bidder and highest bidder's fund pool, which are used as the programmer's reference;
5) Accomplish the *result detection* according to post-condition ($A5$-7). An $assert$-type assertion ($B10$-11) is generated to check the correctness of the highest price, highest bidder and highest bidder's fund pool, as mentioned above. Moreover, the keyword '**Origin**' is used in $A7$ to indicate that the variable $amount$ should be the value before the function is executed, such that the code in $B6$ is appended to record this value before the function body is executed.

Note that the order of post-conditions has an impact on the generation of code. For example, we replace the post-condition in $A5$-7 with the following statement:

```
where this bidder::amount =
  this bidder::Origin amount + highestPrice and
  highestPrice = value and highestBidder = this bidder.
```

The new term have the same meaning as the old one from SPESC, but the generated program will incorrectly add the last highest price to the bidder's funding pool. Fortunately, the assertion can successfully detect this kind of error to remind that the programmer manually adjusts the sequence of statements. In addition, the complete target code of auction contract in Solidity is listed in the appendix.

7 Experimental Results

In this section, a series of experiments are carried out to verify the validity of the auction program generated from the SPESC

TABLE 1: The account states of each steps in the contract experiment.

Participant	Function	Params	balance/eth			total gas	execution gas	store gas
Auctioneer A	Depolay		A:99.9;	B1:100.0;	B2:100:0	2124040	1577980	546060
Auctioneer A	StartBidding	reserve:2eth; time:300s	A:99.9;	B1:100.0;	B2:100:0	110737	89017	21720
Bidder B1	Bid	deposit:3eth	A:99.9;	B1:96.9;	B2:100:0	156552	135280	21272
Bidder B2	Bid	deposit:4eth	A:99.9;	B1: 96.9;	B2:95.9	141552	120280	21272
Bidder B1	WithdrawBid		A:99.9;	B1:100.0;	B2:100:0	44361	38089	6272
Bidder B1	Bid	deposit:5eth	A:99.9;	B1:94.9;	B2:95:9	84480	63208	21272
Auctioneer A	CollectPayment		A:104.9;	B1:94.9;	B2:95:9	81861	60589	21272
Bidder B2	WithdrawBid		A:104.9;	B1:94.9;	B2:99:9	44361	38089	6272

TABLE 2: The account statement for three nodes.

	A	B1	B2
Account Address	0xCA35b7d91 5458EF540aD e6068dFe2F4 4E8fa733c	0x14723A09A Cff6D2A60Dc dF7aA4AFf30 8FDDC160C	0x4B0897b05 13fdC7C541B 6d9D7E929C4 e5364D2dB
Balance	100eth	100eth	100eth

contract to the Solidity code. The experimental platform and results are described as below.

7.1 Auction Contract's Experiments and Results

The SPESC's syntax and translator are implemented by using Xtext, where Xtext is a powerful tool for development of grammar language. Meanwhile, the source code of SPESC's syntax is more than 400 lines, including about 70 syntax elements and 64 syntax rules. The SPESC translator has more than 1000 lines of source code, which are encapsulated into a plugin. Our auction contract in the SPESC language is written through the plugin. The SPESC translator can automatically develop more than 80% of the target code in Solidity. Our experiments are carried out on these converted Solidity codes after a simple revision of codes.

The initial experimental platform runs on three virtual machines with Windows 7 operation system. Each virtual machine is allocated with 2.80GHz, 4G RAM and dual-core Intel CPUs, and is connected to the Ethereum network through NAT. Thus, the three virtual machines can access each other. In this platform, three virtual machines were deployed as three blockchain nodes [22], including an auctioneer node A and two bidder nodes, $B1$ and $B2$, respectively. As shown in TABLE 2 The initial balance of the account is 100 eth for each of three nodes.

The Solidity code need to be compiled into machine code which runs in the Ethereum virtual machine. We adopt the Ethereum's Remix compiler to compile our auction contract, where the Remix is a Solidity IDE that contains functions for writing contracts, testing, debugging, and deploying. The blockchain uses the Ethereum platform. The Ethereum Geth client (1.7.0) is used to take part in the ethereum network.

The contract testing is an interactive process among three nodes. Node A performs the deployment and initialization as:

1) After A deploys the contract's target code into blockchain, Remix exports contracts binary interface. And then, the auctioneer is registered by A, so that only A is admitted to invoke the function $StartBidding$.
2) A executes the $StartBidding$ to start the auction by setting the reserve price to 2 eth and the end time to 5 minutes after execution (The time in Ethereum is measured on the current block time that defines when the block was generated. The Ethereum's average block time is about 15s, so there may be a little deviation between the measured time and the real time).

After this, the functions, including $StartBidding$ and $CollectPayment$, cannot be executed before the end time. Any bidder can execute the function Bid to make a formal offer, but if the offer is less than 2 eth, the execution fails. Next, two nodes, $B1$ and $B2$, implements the bid process as follows:

1) $B1$ executes the function Bid and offers 3 eth to participate in the auction. At this time, the highest bidder is $B1$, the highest bid becomes 3 eth, and B1's fund pool has 3 eth. The contract situation is the same as before.
2) $B2$ executes Bid and offers 4 eth. The highest bidder is $B2$, the highest price becomes 4 eth, and B2's fund pool has 4 eth. As a loser, $B1$ is allowed to withdraw bid from pool.
3) $B1$ executes $WithdrawBid$ to take back the failed bid, then executes Bid again and offers 5 eth. The highest bidder is $B1$, $B1$'s fund pool has 5 eth, and $B2$'s fund pool has 4 eth.

After the above steps, wait until the auction time is over. At last, A executes $CollectPayment$ to collect the highest offer 5 eth, and $B2$ executes $WithdrawBid$ to withdraw the failed bid (in no particular order). So, there is no funds in contract.

In TABLE 1, we show the results of the above experimental process in the Ethereum blockchain, including the states of each steps. The balance of accounts and the gas consumption during execution are also shown in the table. Here, the gas consumption of execution refers to how much gas consumed for executing the smart contract program in the Ethereum virtual machine, that is considered as the actual computational overheads performed by the program. The gas consumption of storage refers to the cost for uploading the variables to the blockchain. According to the table, the experimental results are consistent with the expected values of bids and account balances as mentioned above. This also shows that it is effective for the contract conversion from SPESC to Solidity.

7.2 SPESC-Translator Experiments and Results

We next conduct experiments to test the conversion efficiency and reliability of SPESC-Translator. Our experiments were designed for developing and translating four SPESC contract instances, including loan contract, auction contract, house-leasing contract, and purchase contract. We test the converted smart contracts in Ethereum Rinkeby testnet, which gives developers a chance to freely test and debug the contracts before deployment using real eths in Ethereum mainnet. The experiment results show that SPESC-Translator is a well-behaved and effective tool in Ethereum testnet.

In our experiments, the above contracts are firstly converted into executable contracting programs in Solidity by the translator. Next, a small amount of codes might be manually appended or modified so that four experimental contracts can be compiled and run on the Ethereum testing network. For simplicity, LLOC is denoted as the number of Logical Lines of Codes. In order to measure the performance of the SPESC-Translator, we define the following two concepts. Conversion Rate (CR) represents

TABLE 3: Efficiency and completeness of SPESC-Translator.

Contract name	SPESC	Solidity			Δ	
	LLOC	total LLOC	modified LLOC	auto-generated LLOC	CR	PR
Loan contract	55	225	28	197	87.56%	409.09%
Auction contract	41	309	34	275	89.00%	753.66%
House-leasing contract	71	344	67	267	80.52%	484.51%
Purchase contract	60	418	77	341	81.58%	696.67%
Average	56.75	324	51.5	270	84.67%	585.98%

the percentage of automatically generated lines in the total lines of contract codes, i.e.,

$$\text{CR} = \frac{\text{auto-generated LLOC}}{\text{total LLOC}} \times 100\%. \tag{1}$$

Production Ratio (PR) indicates the ratio of the converted Solidity codes to the original SPESC codes, i.e.,

$$\text{PR} = \frac{\text{total LLOC}}{\text{SPESC LLOC}} \times 100\%. \tag{2}$$

The higher the conversion rate is, the less lines of codes will be modified. On the other hand, the higher the production ratio is, the higher the efficiency of the SPESC-Translator is, and the more labor will be saved.

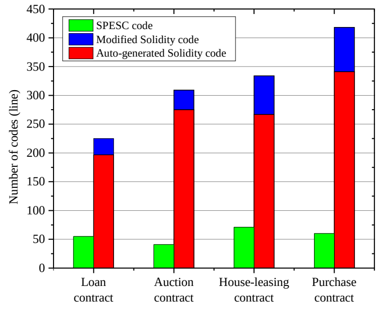


Fig. 14: Logical lines of SPESC and Solidity codes.

Among four tested SPESC contracts, the auction contract is mentioned as above, and three remaining contracts are described as follows:

1) **Loan contract,** refers to a contract where a lender delivers a certain amount of funds to a borrower, and the latter returns the same amount of funds within a certain period of time and pays interests to the lender. There are 6 terms in the contract that consists of two aspects: 1) the borrower shall submit a loan application stating the amount he/she wants to borrow, and 2) the lender may deposit funds into the contract account until the target of loan is reached. Before borrowing, the contract requires that the borrower must confirm the loan information and check the historical loan records, and then calculate the corresponding interests. Meanwhile, the contract asserts that the borrower pays off the loan amount before the loan date comes.
2) **House-leasing contract,** takes assets as the transaction subject between lessor and lessee. There are 7 terms in the contract, including two aspects: 1) the lessor can register the house after depositing the rental deposit, and 2) the lessee can keep the right to use the house during the leasing period after depositing the rent. After confirming the leased house, the contract requires that the lessee shall pay the rent, and the lessor must transfer the right to the lessee within one week. Meanwhile, the contract stipulates the pre-condition for the lessor to charge the rent paid by the lessee, and the post-condition for both parties to get their deposits back after the house inspection.
3) **Purchase contract,** transfers the ownership of goods from sellers to buyers through logistics company. There are 10 terms in the contract, including three parts: 1) once the goods are confirmed, seller will sign a contract and take twice the contracting price as the delivery price, 2) the goods are sent to buyer through logistics when the buyer purchases the goods, and 3) only when the logistics successfully delivers the goods to the buyer can the seller get back the payment. According to relevant laws, the contract requires that buyer can apply for compensation if goods do not arrive within 10 days after payment, or the seller can automatically confirm the arrival if the buyer does not confirm or apply for refund within 7 days after arrival.

As shown in TABLE 3, the number of SPESC lines are 55, 41, 74 and 60 for four experimental contracts, respectively. After converting the executable Solidity programs, the total number of Solidity code lines are 225, 309, 344 and 418, respectively. The number of Solidity code lines manually modified are 28, 34, 67 and 77, respectively. Furthermore, the number of remaining lines automatically generated into Solidity codes are 197, 275, 267 and 341, respectively. Therefore, we can compute the conversion rates CR as 87.56%, 89.00%, 80.52% and 81.58%, and the production ratio PR as 409.09%, 753.66%, 484.51% and 696.67%, respectively.

As shown in Fig. 14, we describe the relationship of logical lines between SPESC and Solidity codes. In the figure, three colors are used to describe the number of code lines in three different states of contracts. Exactly, the green part represents the number of SPESC contract code lines, the blue part represents the number of Solidity code lines that need to be manually modified, and the red part represents the number of lines that is automatically converted into Solidity codes by translator.

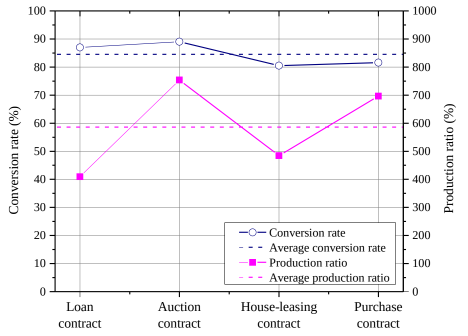


Fig. 15: Conversion rate and production ratio of translator.

It is easy to find that the number of SPESC code lines is much less than that of Solidity, so it indicates that SPESC is highly expressive and reduces labor intensity. Through our experimental observation, the modified codes mainly include two aspects: 1) type declaration of variables need be manually appended into Solidity codes since SPESC does not declare types of few variables, 2) low-level functionalities, e.g., contains and underflow, need be appended into Solidity codes since SPESC does not support these special functions in Solidity. Meanwhile, the fact, that the number of modified Solidity code lines is far less than that of auto-generated lines, indicates that

TABLE 4: Testing results of four converted Solidity contracts under several vulnerability detection tools.

	Environment	Function	Objective	Result
Madmax [23]	Arch Linux, Intel(R) Core(TM) i5-8250U CPU @ 1.60GHz x 8	Detect Dao attacks and other potential threats	EVM bytecode	No vulnerability detected
Mythril [24]	Docker	Effectively detect the security vulnerabilities about 37 types in Smart contract Weakness Classification (SWC), and provide protection for the security of smart contract	.sol file	There are SWC-101, SWC-116, and SWC-105 vulnerabilities in the simple auction contract and only SWC-101 vulnerability in the other three smart contracts.
Oyente [19]	Docker	Integer underflow and overflow, parity multi-sig bug 2, callstack depth attack vulnerability, Transaction-Ordering Dependence (TOD), timestamp dependency, re-entrancy vulnerability	.sol file	No abnormality detected
Smartcheck [25]	Web sites and pages	Check contract code syntax problems and vulnerabilities	.sol file	Pass-test

the proportion of human involvement is relatively low, so as to reduce the difficulty of developing smart contracts.

We further turn our attention to conversion rate and production ration of the SPESC-Translator in Fig. 15. The four hollow circular points on the blue broken line represent the conversion rates of the SPESC-Translator corresponding to the four SPESC contracts, where the conversion rates range from 80% to 90%. And, the blue dotted line indicates the average value of the four instance programs, which is 84.67%. In addition, the solid square dots on the magenta broken line indicate the production ratios of the converted Solidity code lines to the SPESC code lines, where the production ratios range from 4 to 8 times. And the magenta dotted line indicates the average ratio of the four instance programs, which is 5.86 times.

7.3 Contracts' Security Experiments and Results

We assess the security of converted Solidity contracts for verifying the validity of SPESC-Translator. Several vulnerability detection tools are selected to detect the vulnerability of Solidity contracts after investigating the existing relevant researches. The selected testing tools include Madmax [23], Mythril [24], Oyente [19] and Smartcheck [25]. Meanwhile, the test environments, i.e., Docker, Web sites and virtual machine, corresponding to testing tools are deployed to test the four contracts.

The specific test environments, functions, objectives and results are shown in TABLE 4. Among our tests, Mythril is a security analysis tool that uses symbolic execution, SMT solving and taint analysis to detect a variety of vulnerabilities about 37 types for smart contracts. As a result, Mythril detected SWC-101, SWC-105 and SWC-116 vulnerabilities in the simple auction contract according to Smart contract Weakness Classification (SWC), and only SWC-101 vulnerability in the other three contracts. On SWC website (ttps://swcregistry.io/), it was found that SWC-101 is integer overflow and underflow, SWC-105 is unprotected ether withdraw, and SWC-116 is block values as a proxy for time.

The other tools, including the Madmax, Oyente, and Smartcheck, basically cover the main vulnerability detections for smart contracts, but no abnormality was found in the test results of these tools. The latest version of SPESC-Translator has fixed the above vulnerabilities in the converted Solidity codes. In summary, the above experimental results show that the SPESC-Translator is reliable and efficient in real-world scenarios, and verifies the security and effectiveness of the translator's functions.

8 Conclusion

In this paper, a series of generation rules are proposed to realize conversion from the SPESC contract to the Solidity code. Based on these rules, it makes possible for non-programmers to write more effective law-oriented smart contracts. This conversion process not only automatically generates personnel management, data structure and function declaration, but also provides convenient query interfaces. Users do not need to consider the storage structure of parties and accounts, so it increases the applicability despite slightly increasing the computational overheads. Moreover, timing control can be generated according to pre-conditions and post-conditions in contract terms.

There are syntax distinctions in different smart contract languages, but the working structure and mode are not much different among the converted codes in the same target language. This is similar to the case that stack is a universal structure of function processing for various structured programming languages, e.g., Java and ALGOL. Therefore, our result is of useful guideline for the generation of smart legal contracts.

References

[1] C. Linnhoff-Popien, R. Schneider, and M. Zaddach, *Digital Marketplaces Unleashed*. Springer Berlin Heidelberg, 2018.

[2] N. Szabo, "Smart contracts: building blocks for digital markets," *EXTROPY: The Journal of Transhumanist Thought,(16)*, vol. 18, no. 2, 1996.

[3] F. Idelberger, G. Governatori, R. Riveret, and G. Sartor, "Evaluation of logic-based smart contracts for blockchain systems," in *International Symposium on Rules and Rule Markup Languages for the Semantic Web*, vol. 9718. Springer, 2016, pp. 167–183.

[4] C. K. Frantz and M. Nowostawski, "From institutions to code: Towards automated generation of smart contracts," in *2016 IEEE 1st International Workshops on Foundations and Applications of Self* Systems (FAS* W)*. IEEE, 2016, pp. 210–215.

[5] F. Schrans, S. Eisenbach, and S. Drossopoulou, "Writing safe smart contracts in flint," in *Conference Companion of the 2nd International Conference on Art, Science, and Engineering of Programming*, 2018, pp. 218–219.

[6] M. Coblenz, "Obsidian: a safer blockchain programming language," in *2017 IEEE/ACM 39th International Conference on Software Engineering Companion (ICSE-C)*. IEEE, 2017, pp. 97–99.

[7] I. Sergey, A. Kumar, and A. Hobor, "Scilla: a smart contract intermediate-level language," *arXiv preprint arXiv:1801.00687*, 2018.

[8] R. O'Connor, "Simplicity: A new language for blockchains," in *Proceedings of the 2017 Workshop on Programming Languages and Analysis for Security*, 2017, pp. 107–120.

[9] E. Regnath and S. Steinhorst, "Smaconat: Smart contracts in natural language," in *2018 Forum on Specification & Design Languages (FDL)*. IEEE, 2018, pp. 5–16.

[10] O. Choudhury, N. Rudolph, I. Sylla, N. Fairoza, and A. Das, "Autogeneration of smart contracts from domain-specific ontologies and semantic rules," in *2018 IEEE International Conference on Internet of Things (iThings) and IEEE Green Computing and Communications (GreenCom) and IEEE Cyber, Physical and Social Computing (CPSCom) and IEEE Smart Data (SmartData)*. IEEE, 2018, pp. 963–970.

[11] A. Biryukov, D. Khovratovich, and S. Tikhomirov, "Findel: Secure derivative contracts for ethereum," in *International Conference on Financial Cryptography and Data Security*, 2017, pp. 453–467.
[12] X. He, B. Qin, Y. Zhu, X. Chen, and Y. Liu, "Spesc: A specification language for smart contracts," in *2018 IEEE 42nd Annual Computer Software and Applications Conference (COMPSAC)*, vol. 1. IEEE, 2018, pp. 132–137.
[13] M. Hamdaqa, L. A. P. Metz, and I. Qasse, "icontractml: A domain-specific language for modeling and deploying smart contracts onto multiple blockchain platforms," in *Proceedings of the 12th System Analysis and Modelling Conference*, 2020, pp. 34–43.
[14] M. Wöhrer and U. Zdun, "Domain specific language for smart contract development," in *2020 IEEE International Conference on Blockchain and Cryptocurrency (ICBC)*. IEEE, 2020, pp. 1–9.
[15] K. Kasprzyk, "The concept of smart contracts from the legal perspective," *Review of Comparative Law*, vol. 34, no. 3, 2018.
[16] J. Goldenfein and A. Leiter, "Legal engineering on the blockchain:smart contracts as legal conduct," *Law and Critique*, vol. 29, no. 2, pp. 141–149, 2018.
[17] J. G. Allen, "Wrapped and stacked:smart contracts and the interaction of natural and formal language," *European Review of Contract Law*, vol. 14, no. 4, pp. 307–343, 2018.
[18] S. S. Gomes, "Smart contracts: legal frontiers and insertion into the creative economy," *Brazilian Journal of Operations & Production Management*, vol. 15, no. 3, pp. 376–385, 2018.
[19] L. Luu, D.-H. Chu, H. Olickel, P. Saxena, and A. Hobor, "Making smart contracts smarter," in *2016 ACM SIGSAC conference on computer and communications security*, 2016, pp. 254–269.
[20] N. Atzei, M. Bartoletti, and T. Cimoli, "A survey of attacks on ethereum smart contracts (sok)," in *International conference on principles of security and trust*, vol. 10204. Springer, 2017, pp. 164–186.
[21] K. Bhargavan, A. Delignat-Lavaud, C. Fournet, A. Gollamudi, G. Gonthier, N. Kobeissi, N. Kulatova, A. Rastogi, T. Sibut-Pinote, N. Swamy *et al.*, "Formal verification of smart contracts: Short paper," in *Proceedings of the 2016 ACM workshop on programming languages and analysis for security*, 2016, pp. 91–96.
[22] R. M. Parizi, A. Dehghantanha *et al.*, "Smart contract programming languages on blockchains: An empirical evaluation of usability and security," in *Proceedings of 2018 International Conference on Blockchain (ICBC 2018)*. Springer, 2018, pp. 75–91.
[23] N. Grech, M. Kong, A. Jurisevic, L. Brent, B. Scholz, and Y. Smaragdakis, "Madmax: Surviving out-of-gas conditions in ethereum smart contracts," *Proceedings of the ACM on Programming Languages*, vol. 2, no. OOPSLA, pp. 1–27, 2018.
[24] MythX, "Mythril classic: Security analysis tool for ethereum smart contracts," in *https://github.com/ConsenSys/mythril-classic; [accessed 30.09.19]*, 2018.
[25] S. Tikhomirov, E. Voskresenskaya, I. Ivanitskiy, R. Takhaviev, E. Marchenko, and Y. Alexandrov, "Smartcheck: Static analysis of ethereum smart contracts," in *Proceedings of the 1st International Workshop on Emerging Trends in Software Engineering for Blockchain*, 2018, pp. 9–16.

E Chen received the B.S. degree from the department of School of Mathematics and Physics, University of Science and Technology Beijing, China. She is currently a Ph.D. candidate with the department of School of Computer and Communication Engineering, University of Science and Technology Beijing, China. Her research interests include attribute based system, blockchain, smart contract and lattice cryptography. (Email: chene5546@163.com)

Bohan Qin received the B.E. degree from the department of School of Computer and Communication Engineering, University of Science and Technology Beijing, China. He is currently a MA.Sc. candidate with the department of School of Computer and Communication Engineering, University of Science and Technology Beijing, China. His research interests include blockchain and smart contract. (Email: qinbohan@126.com)

Yan Zhu was an Associate Professor of Computer Science with the Institute of Computer Science and Technology, Peking University, China, from 2007 to 2013. He was a visiting scholar with Arizona State University from 2008 to 2009, as well as University of Michigan-Dearborn in 2012. He is currently a Professor with University of Science and Technology Beijing, China. His research interests include cryptography and security. (Email:zhuyan@ustb.edu.cn)

Weijing Song received the B.E. degree from the department of School of Computer and Communication Engineering, University of Science and Technology Beijing, China. She is currently a MA.Sc. candidate with the department of School of Computer and Communication Engineering, University of Science and Technology Beijing, China. Her research interests include blockchain and smart contract. (Email: songweijing@xs.ustb.edu.cn)

Shengdian Wang received the B.E. degree from the department of School of Computer and Communication Engineering, University of Science and Technology Beijing, China. He is currently a MA.Sc. candidate with the department of School of Computer and Communication Engineering, University of Science and Technology Beijing, China. His research interests include blockchain and smart contract. (Email: wsd@xs.ustb.edu.cn)

Stephen S. Yau is currently professor of Arizona State University (ASU), USA. He served as the chair of the Department of Computer Science and Engineering at ASU. Previously, he was on the faculties of Northwestern University, Evanston, Illinois, and the University of Florida, Gainesville. He served as the president of IEEE Computer Society, and was on the board of directors of IEEE and of Computing Research Association. He served as the editor-in-chief of IEEE Computer magazine, organizing committee chair of 1989 World Computer Congress sponsored by IFIP, and the chair of COMPSAC 1977 and its steering committee chair in subsequent years sponsored by IEEE Computer Society. He was general chair of 2018 IEEE World Congress on Services, and an honorary co-chair of 2017 IEEE Smart World Congress. He received Tsutomu Kanai Award and Richard E. Merwin Award of IEEE Computer Society, and Outstanding Contributions Award of Chinese Computer Federation. He is a Fellow of the AAAS and IEEE. (Email: yau@asu.edu)

Cheng-Chung William Chu (SM'19) received the M.S. and Ph.D. degrees in computer science from Northwestern University, Evanston, IL, USA, in 1987 and 1989, respectively. He is currently a Distinguished Professor with the Department of Computer Science, Tunghai University, Taichung, Taiwan, where he had served as the Director of Software Engineering and Technologies Center from 2004 to 2016 and as the Dean of Research and Development office from 2004 to 2007. He has edited several books and authored or coauthored more than 100 referred papers and book chapters, as well as participated in many international activities, including organizing conferences, serving as the steering committee for the IEEE COMPSAC, the APSEC, the IEEE QRS, the ISSSR, and the program committee of more than 70 conferences. He is an Associate Editor for the IEEE TRANSACTIONS ON RELIABILITY, the Journal of Software Maintenance and Evolution, and the International Journal of Advancements in Computing Technology.(Email: cchu@thu.edu.tw)