2018 42nd IEEE International Conference on Computer Software & Applications

SPESC: A specification language for smart contracts

Xiao He*, Bohan Qin*, Yan Zhu*, Xing Chen†, Yi Liu‡
*School of Computer and Communication Engineering, University of Science and Technology Beijing, Beijing 100083, China
Email: {hexiao,zhuyan}@ustb.edu.cn
†Fujian Provincial Key Laboratory of Network Computing and Intelligent Information Processing,
Fuzhou University, Fuzhou 350116, China
‡National Computer Network Emergency Response Technical Team/Coordination Center of China

***Abstract*—The smart contract is an interdisciplinary concept that concerns business, finance, contract law and information technology. Designing and developing a smart contract may require the close cooperation of many experts coming from different fields. How to support such collaborative development is a challenging problem in blockchain-oriented software engineering. This paper proposes SPESC, a specification language for smart contracts, which can define the specification of a smart contract for the purpose of collaborative design. SPESC can specify a smart contract in a similar form to real-world contracts using a natural-language-like grammar, in which the obligations and rights of parties and the transaction rules of cryptocurrencies are clearly defined. The preliminary study results demonstrated that SPESC can be easily learned and understood by both IT and non-IT users and thus has greater potential to facilitate collaborative smart contract development.**

1. Introduction

Decentralized cryptocurrencies have rapidly gained in popularity since Bitcoin [1] was introduced. Conceptually, a decentralized cryptocurrency system, which is administered without relying on any trusted parties, maintains and secures some shared data that are stored using blockchains. As a data structure, a blockchain is an immutable ordered list of blocks [1], [2]. Each block consists of many transaction records and a hash of the previous block. The data stored in the blockchain cannot be deleted or altered; otherwise, the hashes will be invalidated [2].

A prominent and promising use for blockchains is to enable smart contracts [2], [3]. Technically, a smart contract is a program that is deployed and running on blockchains. Smart contracts have been used to realize various applications and operations (such as financial instruments, product traceability service, and self-enforcing or autonomous governance applications [3]).

A smart contract is encoded using smart contract programming languages, such as the script language of Bitcoin and Solidity of Ethereum [4]. Although a smart contract is usually viewed as an online program from the perspective of information technology, it is actually an interdisciplinary concept that also concerns (but is not limited to) business/finance and contract law [5]. From the point of view of business and finance, a smart contract specifies how transactions and payments are executed among different accounts. From the point of view of contract law, a contract is an agreement between parties consisting of mutual commitments; and, a smart contract is a contract in which the commitment fulfillment is performed automatically [6].

Due to its interdisciplinary nature, a smart contract may be proposed, designed and implemented collaboratively by many experts [5], such as software and information security engineers, business experts, bank managers, and lawyers, who come from different domains. A crucial issue during smart contract development is to enable domain experts to understand, discuss and specify the contract collaboratively. Existing smart contract languages, such as Solidity [4] and Hawk [7], are primarily defined from an IT perspective. The smart contracts that are encoded using these languages (as illustrated in Fig. 1) may not be readable to the practitioners outside of the IT industry.

This paper proposes SPESC, a specification language for smart contracts. SPESC is intended to be used to define the specification of a smart contract (rather than its implementation) for the purpose of collaborative design. In SPESC, a smart contract is regarded as a composition of information technology, contract law, and business/finance transaction. First, SPESC enables us to specify a smart contract in a similar form to a real-world contract. An SPESC specification consists of the descriptions of *parties* (i.e., the roles that participate in the contract) and a set of *terms* (i.e., the obligations and rights of each party, and the conditions when the terms should hold). Second, for each term, SPESC also enables us to describe how the balances of the parties' accounts should change when this term holds.

The rest of this paper is structured as follows. Section 2 briefly introduces the concepts of blockchains and smart contracts. Section 3 presents the syntax and semantics of SPESC. Section 4 evaluates the learnability and understandability of SPESC with a case study. Section 5 discusses related research efforts. The last section concludes the paper and identifies future work.

978-1-5386-2666-5/18/$31.00 ©2018 IEEE
DOI 10.1109/COMPSAC.2018.00025

```
contract OnlinePurchase {
  address public seller;
  address public buyer;
  uint public price;
  function OnlinePurchase() public {...} // constructor
  // check whether the caller is the buyer
  modifier onlyBuyer() {require(msg.sender == buyer);_;}
  // check whether the caller paid the right amount of money
  modifier rightMoney {require(msg.value==price);_;}
  // the definition of function pay,
  //   where keyword payable means the caller must send some money
  function pay() public onlyBuyer rightMoney payable {
    // transfer the money from the caller (i.e., the buyer) to the seller by
    //   using the local account held by this contract as a bridge
    // the money sent by the caller is deposited into this.balance first
    seller.transfer(this.balance);
  }
  …
}
```

Figure 1. An example Solidity program

2. Background

A block chain is a simple cryptographically secured data structure [8] that takes a number of records and puts them in a block (like collating them in a single sheet). Each block is then *chained* to the next block, using a cryptographic hash. This allows the block chain to be used like a public ledger [9], which can be shared and corroborated by anyone, without having to worry that the stored records can be changed. Therefore, *Blockchain* is the term that is used to describe this distributed and decentralized ledger system that is spread across multiple sites, countries or institutions.

As everyone knows, blockchain technology originated from Bitcoin; cryptocurrency [10] is the most successful application of blockchain. In cryptocurrency, blockchain can be used as a medium of exchange, which keeps a record of all transactions that take place across the entire system. The major innovation is that the technology allows market participants to transfer assets across the Internet without the need for a centralized third party [11]. Moreover, the encryption techniques can be used to control the creation of monetary units and to verify the transfer of funds.

Smart contracts [12], as illustrated in Fig. 1, are contracts whose terms are encoded in a programming language instead of a legal language. Smart contracts can be automatically executed by a computing system, such as a suitable distributed ledger system, such that it is convenient to complete the transfer of assets. The potential benefits of smart contracts include low contracting, enforcement, and compliance costs; consequently, it becomes economically viable to form contracts over numerous low-value transactions [13].

3. Specification language for smart contracts

A smart contract is a program that owns not only IT attributes but also financial and legal attributes. Smart contract development requires the close cooperation of experts from related domains. This section proposes our smart contract specification language, namely, SPESC. The goal of SPESC is to establish an abstraction layer atop concrete implementations of smart contracts to facilitate the collaboration among experts from different domains. In SPESC, a smart contract is viewed as a cross-disciplinary artifact and is specified from three perspectives, i.e., programming, business/finance transactions and contract law.

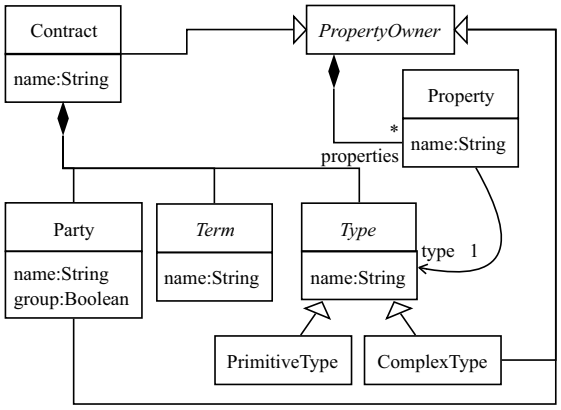

Figure 2. Metamodel of general structure

The rest of this section will use a simple *Purchase* contract to illustrate SPESC. This simple contract must be signed by a buyer and a seller. When the contract is signed, the price of the product must be specified. The buyer must pay within three days after the contract is signed, unless the buyer cancels the contract. The payment is frozen rather than transferred to the seller directly. The buyer can cancel this contract before payment. The seller must post the product within five days after the payment. The buyer must acknowledge the receipt of the product within ten days after the product was posted. While the receipt of the product is acknowledged, the payment will be transferred to the seller. If the buyer did not acknowledge the receipt of the product in time, then the seller can also collect the payment.

3.1. General structure

In SPESC, a contract consists of parties, contract properties, terms and data type definitions. The metamodel of the overall structure of SPESC is presented in Fig. 2.

Contract in Fig. 2 represents a smart contract under development. *Party* denotes a role who participates in the contract and interacts with others. *Term* specifies a certain obligation or right that is associated with a party. The detailed definitions of *Party* and *Term* will be presented later.

A contract in SPESC may also own some contract properties. *Property* denotes a static field of a certain data type. Contract properties can be used to specify the necessary properties of the object of the contract, such as the amount and the price of the goods to be sold. SPESC enables us to explicitly define data types using *Type*, which has two concrete subclasses, namely, *PrimitiveType* and *ComplexType*.

The following code snippet shows the general structure of the contract *Purchase* that is written in the concrete syntax of SPESC. Keywords **contract**, **party**, **term** and **type** indicate the corresponding concepts defined in Fig. 2.

```
contract Purchase {
  party Seller {/* details of Seller */}
  party Buyer {/* details of Buyer */}
  info : ProductInfo // contract property
  term No1 : … /* details of Term */
  term No2 : …
  …/* other Terms */
  type ProductInfo {
    price : Money,
    model :  String
  }
}
```

3.2. Parties and terms

Parties and terms are essential components of a real-world contract. The SPESC metamodel that concerns parties and terms is presented in Fig. 3

In SPESC, a *Party* is specified by a name that is used as an identifier in the rest of the contract, some party properties (i.e., the information that must be recorded on the blockchains, see Fig. 2) and a set of *Actions*. Each *Action* represents a certain act a party may or must perform. An *Action* is declared (and will be implemented) as a function in a smart contract program.

By default, a *Party* is a user of a smart contract system. SPESC also supports multiple users playing the same role by assigning the attribute *group* of *Party* true. Fig. 4 shows three party declarations. The first two examples show how the parties *Buyer* and *Seller* are defined in the concrete syntax of SPESC. The third example demonstrates a group party *Voters*, which at runtime can be played by multiple users.

After declaring an *Action* for a *Party*, we must specify when the *Action* must or may be performed by this *Party* by using contract *Terms*. In Fig. 3, a *Term* refers to a *Party* and an *Action* of this *Party* and is associated with pre/postconditions (specified by *Expressions*). *Term* has two subclasses, namely, *Obligation* and *Right*. An *Obligation* term defines that the referred *Party must* conduct the *Action* under a certain precondition. A *Right* term defines that the

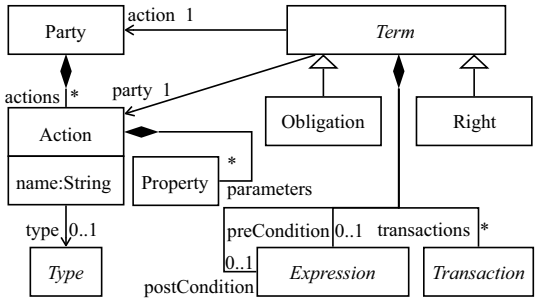

Figure 3. Metamodel of parties and terms

```
party Seller {
  post()
  collect()
}
```

```
party Buyer {
  postTo : Address
  pay()
  cancel()
  receive()
}
```

```
party group Voters {
  name:Name
  vote()
}
```

Figure 4. Party examples

Party may conduct the *Action* under a specified precondition. When the term condition does not hold at runtime, we assume that the *Party* cannot conduct the *Action*.

SPESC regards transactions as the core of smart contracts. SPESC provides a way of specifying the cryptocurrency transactions for each *Term*. In Fig. 3, *Term* may also be associated with a set of *Transaction*s. The details of *Expression* and *Transaction* will be introduced later.

For greater comprehensibility, SPESC employs a natural-language-like concrete syntax to specify contract terms. The concrete syntax for *Term*s is defined as follows in EBNF, where the concepts defined in the metamodel of SPESC are displayed in *italics* and keywords are displayed in **bold** type.

term *name* **:** *party* (**must**|**may**) *action*
(**when** *preCondition*)?
(**while** *transactions*+)?
(**where** *postCondition*)? **.**

A *Term* owns a *preCondition*, which specifies the condition under which the term must/may be fulfilled, and a *postCondition*, which specifies the condition that must hold after the term is fulfilled. The following examples show three terms (i.e., *No1* to *No3*). For instance, term *No1* is an *Obligation*

```
term No1: Buyer must pay when within 3 day after start
          while deposit $info::price.
term No2: Buyer may cancel when before Buyer did pay.
term No3: Seller must post when within 5 day after Buyer did pay.
```

Figure 5. Term examples

term. It specifies that the *Party Buyer* must perform the *Action pay* within three days after the contract starts and that *Buyer* must deposit an amount of cryptocurrencies as specified by *price* of the contract property *info*.

3.3. Expressions

SPESC supports various expressions to specify term conditions, such as time predicates, logical expressions, relational expressions, arithmetic expressions and constant expressions. Fig. 6 shows the metamodel of expressions and transactions in SPESC. Due to space limitations, Fig. 6 only shows parts of the complete metamodel. Basic expressions, such as *LogicalExpression*s, *RelationalExpression*s, *ArithmeticExpression*s and *ConstantExpression*s, which are supported in SPESC, are omitted in this paper.

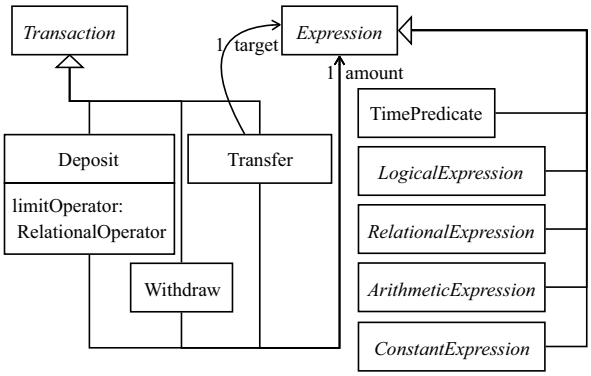

Figure 6. Metamodel of expressions and transactions (excerpt)

In a real-world contract, the obligations and rights of parties are usually defined under certain temporal conditions. For example, a buyer must pay *within three days after the contract is signed*. We assume that the temporal condition is essential for a smart contract specification. SPESC supports *TimePredicate* to define temporal conditions. The definition of *TimePredicate* is presented in Fig. 7.

TimePredicate checks the relation between the current date/time and the *base* time point. If the *comparator* of a *TimePredicate* is *before*, then this *TimePredicate* checks whether the current time is earlier than the specified *base* time point (defined as a *TimeExpression*); otherwise, this *TimePredicate* checks whether the current time is later than the *base* time point. Its concrete syntax is defined as follows.

$$(\textbf{before}|\textbf{after})\ base$$

before b is equivalent to $current\ time \leq b$, and **after** b is equivalent to $current\ time \geq b$

TimeExpression is used to specify a time point. It has five subclasses, namely, *TimeVar*, *TimeConst*, *ActionDoneQuery*, *TermDoneQuery* and *GlobalTimeQuery*. *TimeVar* refers to a variable that denotes a date/time. *TimeConst* denotes a time constant, such as 5 **day** and 1 **hour**. *GlobalQuery* can return the current date/time and the creation time of this contract.

ActionDoneQuery denotes the time point when a certain *action* is conducted by a *party*. Its syntax is as follows.

$$(\textbf{all}|\textbf{some}|\textbf{this})?\ party\ \textbf{did}\ action$$

If the *party* did not perform the *action*, then *ActionDoneQuery* returns $\perp$. $\perp$ denotes the unsolved time. Any solved time value is earlier than $\perp$.

ActionDoneQuery may contain a quantifier (i.e., **all**, **some** and **this**) when the *party* denotes a group. Consider the previous example of the party group *Voters* (Fig. 4) who can *vote* and *delegate* others to vote. Assume that there are n *Voters* in total and that ith voter *vote*d at t_i. Quantifier **all** means that *ActionDoneQuery* returns the time when the last member in the *party* performs the *action*. For instance, **all** $Voters$ **did** $vote$ returns $max(t_1, ..., t_n)$. Quantifier **some** means that *ActionDoneQuery* returns the time when the first member in the *party* performs the *action*. For instance, **some** $Voters$ **did** $vote$ returns $min(t_1, ..., t_n)$. Quantifier **this** means that *ActionDoneQuery* returns the time when the user referred by the contract *Term* performs the *action*. For example, the following term *VoteOnce* prescribes that a voter can only vote once.

term VoteOnce**:** Voters **may** vote
when before this Voters **did** vote.

For a user who belongs to *Voters*, we check his/her voting time when evaluating the term *VoteOnce*.

TermDoneQuery is similar to *ActionDoneQuery*. It denotes the time point when an *Obligation* term is fulfilled. Assume that the queried *Obligation* term states that party p must conduct an action o. A *TermDoneQuery* about this term is equivalent to an *ActionDoneQuery*, i.e., **all** p **did** o.

BoundedTimePredicate extends *TimePredicate* by an extra time *boundary*. The *boundary* may be a *closed* boundary or an open boundary. Its concrete syntax is as follows.

$$(\textbf{within})?\ boundary\ (\textbf{before}|\textbf{after})\ base$$

There are four basic forms of *BoundedTimePredicate*. Their semantics are listed as follows. **within** b **before** t: It is equivalent to check $t - b \leq current\ time \leq t$. For example, **within** 3 **day before** $Buyer$ **did** pay. b **before** t: It is equivalent to check $current\ time \leq t - b$. For example, 2 **day before** $Guest$ **did** $arrive$. **within** b **after** t: It is equivalent to check $t \leq current\ time \leq t + b$. b **after** t: It is equivalent to check $t + b \leq current\ time$.

The expressions supported by SPESC are not Turing complete. It is possible that some complex conditions cannot be encoded into SPESC currently. The major reason is that SPESC is a specification language for collaborative development rather than an implementation language. SPESC is expected to be understandable by non-IT experts. Hence, we made a trade-off between the completeness/expressive power and the learnability/understandability of SPESC. As

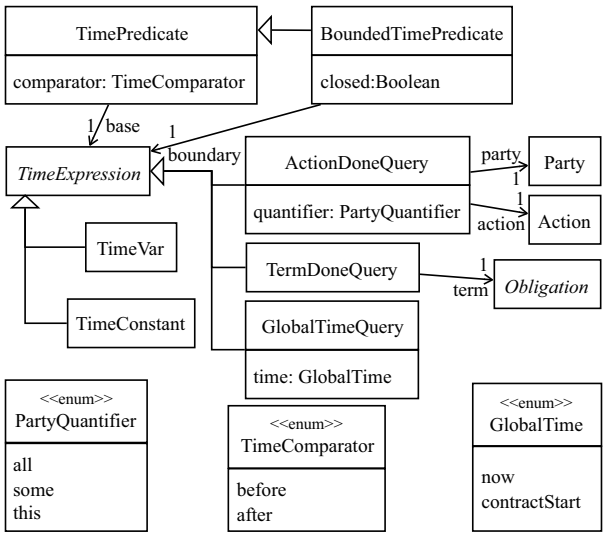

Figure 7. Metamodel of time predicates

a future work, we plan to enhance the expressive power of SPESC while keeping its learnability and understandability.

3.4. Transactions

On a cryptocurrency platform, every user has an account (or a wallet) that holds some cryptocurrencies; every smart contract program also has an account. Users may transfer some cryptocurrencies from their accounts to a smart contract's account when they are interacting with this smart contract. Users may also ask a smart contract to transfer some cryptocurrencies from its account to other accounts.

To model the transaction amount accounts, Fig. 6 also defines the concept of *Transaction*. SPESC currently supports three types of *Transaction*, namely, *Deposit*, *Withdraw* and *Transfer*. The concrete syntax of the three transaction operations is listed as follows.

deposit $(= \mid > \mid >= \mid < \mid <=)?$ **\$**$amount$
withdraw \$$amount$
transfer \$$amount$ **to** $target$

Deposit means that a user must put some cryptocurrencies into the account of the smart contract he/she is interacting with. The *limitOperator* indicates whether the cryptocurrencies to be saved should be equal to or less/greater than the specified *amount*. If the *limitOperator* is missing in concrete syntax, then it is regarded as $=$. *Withdraw* means that a user will get a certain *amount* of cryptocurrencies from the account of the smart contract he/she is interacting with. *Transfer* means that the smart contract will give the *target* some cryptocurrencies as specified by *amount*. All *amount*s of cryptocurrencies are specified by an *Expression*; *target* of *Transfer* is expected to reference a *Party*.

Recall the example *Purchase* contract. According to the contract description presented at the beginning of Section 3, *Buyer* is expected to pay for the goods when he/she performs the action *pay*. The money he/she pays will be frozen by the smart contract. Hence, in Fig. 5, term *No1* contains a *Deposit* operation to ask the *Buyer* to transfer the goods price to the smart contract's account. The contract *Purchase* also states that *Buyer* must *receive* the goods within 15 days after they are mailed, and must unfreeze and transfer the money to *Seller*; otherwise, *Seller* can *collect* the money. We realize this requirement by defining the following two terms.

```
term No4: Buyer must receive
          when within 15 day after Seller did post
          while transfer $info::price to Seller.
term No5: Seller may collect when 15 day after Seller did post
          while withdraw $info::price.
```

4. Evaluation

We conducted a case study to evaluate the learnability and understandability of SPESC, two critical features in the collaborative development of smart contracts. In our case study, participants were required to read two different smart contracts. Afterwards, participants were asked some questions. We chose Solidity, which is a state-of-the-art smart contract programming language, as the language to be compared with SPESC. The details of this evaluation are presented on our website[1]

We recruited fifteen participants in total from our university. Nine participants were from Department of Computer Science; four of them were master's students and the other five were senior bachelor's students. Six participants were from Department of Law, and all of them were master's students. We recruited participants from the department of law because we regard the smart contract as an interdisciplinary concept. All the fifteen participants were randomly divided into two groups (namely, `GA` and `GB`). `GA` consisted of four CS students (namely, `GA-CS`) and three law students (namely, `GA-L`). `GA` consisted of five CS students (namely, `GB-CS`) and three law students (namely, `GB-L`).

We chose two smart contracts, namely *Lending* and *Voting*. *Lending* is adapted from an open source Solidity program[2] on Github. *Voting* is adapted from an official example[3] described in Solidity documents. We enhanced the official example by supporting a multi-stage voting strategy. We used SPESC to re-specify the two smart contracts (namely, `Lending-SP` and `Voting-SP`) according to the source code of the two Solidity programs (namely, `Lending-SO` and `Voting-SO`). `Lending-SP` has 57 LOC and `Lending-SO` has 152 LOC; `Voting-SP` has 59 LOC and `Voting-SO` has 147 LOC.

First, we gave a one-hour tutorial in SPESC and Solidity. Participants were required to do their best to read some example programs and ask questions if they encountered any problem. We closed the tutorial after each participant confirmed that he/she had no more questions.

Second, we asked `GA` and `GB` to read `Lending-SP` and `Lending-SO`, respectively, and to complete a questionnaire about the smart contract *Lending*. We timed how many seconds each participant consumed reading and completing his/her questionnaire. After that, we asked `GA` and `GB` to read `Voting-SO` and `Voting-SP`, respectively, and to complete a questionnaire about the smart contract *Voting*.

Due to space limitation, the details of the questionnaires and the raw results are presented on our website [1]. The results showed (1) that for the first questionnaire, `GA` used less time than `GB` (745.4 s vs. 1439.6 s) and achieved a higher correct rate than GB (86% vs. 39.4%), and (2) that for the second questionnaire `GB` used less time than `GA` (1338.3 s vs. 698.6 s) and achieved a higher correct rate than GA (57.3% vs. 76.0%). These results demonstrated that SPESC has better the learnability and understandability.

This study mainly suffers from a threat to external validity that the participants and the smart contracts used in this study may not represent real practitioners and smart contract systems. In the future, we will repeat this study in a real industrial context.

1. https://bitbucket.org/ustbmde/smartcontract/wiki/Home
2. https://github.com/terzim/dapp-bin.
3. https://solidity.readthedocs.io/en/develop/solidity-by-example.html#voting

5. Related work

Although smart contracts have been studied by many researchers from various perspectives, such as concurrent programming [15], [16], security [7], [17], [18] and software engineering [5], [6], [19], to the best of our knowledge, we are among the first to propose a specification language of smart contracts for collaborative development.

Porru et al. [5] identified many challenges for blockchain-oriented software engineering. They argued that blockchain-oriented systems require professionals with skills such as finance law, and technology. This paper follows this view and proposes a new specification language for smart contracts, which is designed for better collaboration.

Frantz et al. [19] proposed a modeling approach that supports the semi-automated translation of human-readable contract representations into computational equivalents. They adapted ADICO [20] for modeling smart contracts. From ADICO-based models, they also developed a code generator to derive partial source code. The ADICO format can specify a party's obligations and rights. SPESC covers the major concepts in ADICO. Moreover, SPESC is able to specify post conditions and transaction rules and provides a user-friendly grammar.

De Kruijff et al. [6] proposed a blockchain ontology for blockchain practitioners. Kosba et al. [7] proposed a new smart contract language Hawk to realize privacy-preserving. There are also many research efforts [2], [13], [21], [22] on the application of blockchain and smart contracts.

6. Conclusion and future work

This paper proposed SPESC, a specification language for smart contracts. SPESC is intended to facilitate collaborative development of smart contract systems. Our preliminary study results demonstrated that SPESC is easy to learn and understand and is preferred by most participants.

In the future, we plan to enhance the expressive power and the grammar of SPESC. We will also develop a code generator and build a complete development environment to facilitate smart contract development. More case studies will be conducted to evaluate the expressive power, user-friendliness, learnability and understandability of SPESC, and the extent to which SPESC contributes to interdisciplinary cooperation in an industrial context.

Acknowledgments

This work was supported by the National Key R&D Program of China (Grant No. 2017YFB1002000), the National Natural Science Foundation of China (Grant Nos. 61300009, 61472032) and Joint Research Fund for Overseas Chinese Scholars and Scholars in Hong Kong and Macao (Grant No. 61628201).

References

[1] S. Nakamoto, "Bitcoin: A Peer-to-Peer Electronic Cash System," Tech. Rep. [Online]. Available: http: //bitcoin.org/bitcoin.pdf

[2] Q. Lu and X. Xu, "Adaptable Blockchain-Based Systems: A Case Study for Product Traceability," *IEEE Software*, vol. 34, no. 6, pp. 21–27, nov 2017.

[3] L. Luu, D.-H. Chu, H. Olickel, P. Saxena, and A. Hobor, "Making smart contracts smarter," in *Proc. of ACM CCS'16*, Vienna, Austria, 2016, pp. 254–269.

[4] Ethereum project. [Online]. Available: https://www.ethereum.org

[5] S. Porru, A. Pinna, M. Marchesi, and R. Tonelli, "Blockchain-oriented software engineering: Challenges and new directions," in *Proc. of IEEE/ACM 39th ICSE-C 2017*, Argentina, 2017, pp. 169–171.

[6] J. De Kruijff and H. Weigand, "Understanding the blockchain using enterprise ontology," in *LNCS*, vol. 10253, 2017, pp. 29–43.

[7] A. Kosba, A. Miller, E. Shi, Z. Wen, and C. Papamanthou, "Hawk: The Blockchain Model of Cryptography and Privacy-Preserving Smart Contracts," in *Proc. of IEEE SP 2016*, San Jose, CA, United states, 2016, pp. 839–858.

[8] I.-C. Lin and T.-C. Liao, "A survey of blockchain security issues and challenges." *IJ Network Security*, vol. 19, no. 5, pp. 653–659, 2017.

[9] A. G. Abbasi and Z. Khan, "Veidblock: Verifiable identity using blockchain and ledger in a software defined network," in *Proc. of 10th Int. Conf. on Utility and Cloud Computing*. New York, NY, USA: ACM, 2017, pp. 173–179.

[10] A. Narayanan, J. Bonneau, E. Felten, A. Miller, and S. . . Goldfeder, *Bitcoin and Cryptocurrency Technologies: A Comprehensive Introduction*. Princeton University Press, 2016.

[11] D. Chatzopoulos, M. Ahmadi, S. Kosta, and P. Hui, "Flopcoin: A cryptocurrency for computation offloading," *IEEE Transactions on Mobile Computing*, 2017.

[12] K. Delmolino, M. Arnett, A. Kosba, A. Miller, and E. Shi, "Step by step towards creating a safe smart contract: Lessons and insights from a cryptocurrency lab," in *Proc. of Int. Conf. on Financial Cryptography and Data Security*. Springer, 2016, pp. 79–94.

[13] S.-C. Cha, W.-C. Peng, Z.-J. Huang, T.-Y. Hsu, J.-F. Chen, and T.-Y. Tsai, "On design and implementation a smart contract-based investigation report management framework for smartphone applications," in *Proc. of Int. Conf. on Intelligent Information Hiding and Multimedia Signal Processing*. Springer, 2017, pp. 282–289.

[14] Xtext project. [Online]. Available: http://www.eclipse.org/xtext

[15] L. Yu, W.-T. Tsai, G. Li, Y. Yao, C. Hu, and E. Deng, "Smart-Contract Execution with Concurrent Block Building," in *Proc. 11th IEEE SOSE*, San Francisco, CA, United states, 2017, pp. 160–167.

[16] T. Dickerson, P. Gazzillo, M. Herlihy, and E. Koskinen, "Adding concurrency to smart contracts," in *Proc. of the Annual ACM Symposium on Principles of Distributed Computing*, vol. Part F1293, Washington, DC, United states, 2017, pp. 303–312.

[17] M. Frowis and R. Bohme, "In code we trust?: Measuring the control flow immutability of all smart contracts deployed on Ethereum," in *LNCS*, vol. 10436, 2017, pp. 357–372.

[18] S. Grossman, I. Abraham, G. Golan-Gueta, Y. Michalevsky, N. Rinetzky, M. Sagiv, and Y. Zohar, "Online detection of effectively callback free objects with applications to smart contracts," in *Proc. of POPL'18*, dec 2018, pp. 1–28.

[19] C. K. Frantz and M. Nowostawski, "From institutions to code: Towards automated generation of smart contracts," in *Proc. of IEEE 1st Int. Workshops on Foundations and Applications of Self-Systems*, Augsburg, Germany, 2016, pp. 210–215.

[20] S. E. S. Crawford and E. Ostrom, "A grammar of institutions," *American Political Science Review*, vol. 89, no. 03, pp. 582–600, 1995.

[21] B. Tackmann, "Secure event tickets on a blockchain," in *LNCS*, vol. 10436, 2017, pp. 437–444.

[22] A. Yasin and L. Liu, "An Online Identity and Smart Contract Management System," in *Proc. of COMPSAC'16*, vol. 2, Atlanta, GA, United states, 2016, pp. 192–198.