electronics

MDPI

Article

Formal Modeling and Verification of Smart Contracts with Spin

Zhe Yang [1], Meiyi Dai [1] and Jian Guo [2,*]

[1] MoE Engineering Research Center for Software/Hardware Co-Design Technology and Application, East China Normal University, Shanghai 200062, China
[2] National Trusted Embedded Software Engineering Technology Research Center, East China Normal University, Shanghai 200062, China
* Correspondence: jguo@sei.ecnu.edu.cn

Abstract: Smart contracts are the key software components to realize blockchain applications, from single encrypted digital currency to various fields. Due to the immutable nature of blockchain, any bugs or errors will become permanent once published and could lead to huge economic losses. Recently, a great number of security problems have been exposed in smart contracts. It is important to verify the correctness of smart contracts before they are deployed on the blockchain. This paper aims to verify the correctness of smart contracts in Ethereum transactions, and the model checker Spin is adopted for the formal verification of smart contracts in order to ensure their execution with respect to parties' willingness, as well as their reliable interaction with clients. In this direction, we propose a formal method to construct the models for smart contracts. Then, the method is applied to a study case in the Ethereum commodity market. Finally, a case model is implemented in Spin, which can simulate the process's execution and verify the properties that are abstracted from the requirements. Compared with existing techniques, formal analysis can verify whether smart contracts comply with the specifications for given behaviors and strengthen the credibility of smart contracts in the transaction.

Keywords: formal verification; LTL; model checking; smart contract; blockchain; Spin

check for updates

Citation: Yang, Z.; Dai, M.; Guo, J. Formal Modeling and Verification of Smart Contracts with Spin. *Electronics* **2022**, *11*, 3091. https://doi.org/10.3390/electronics11193091

Academic Editor: Domenico Ursino

Received: 25 July 2022
Accepted: 24 September 2022
Published: 27 September 2022

Publisher's Note: MDPI stays neutral with regard to jurisdictional claims in published maps and institutional affiliations.

Copyright: © 2022 by the authors. Licensee MDPI, Basel, Switzerland. This article is an open access article distributed under the terms and conditions of the Creative Commons Attribution (CC BY) license (https://creativecommons.org/licenses/by/4.0/).

1. Introduction

Blockchain is a decentralized, distributed digital ledger [1], allowing transactions to be processed without the necessity of a trusted third party. As a result, business activities can be completed in an efficient manner. Moreover, the immutability of blockchain also ensures distributed trust since it is impossible to tamper with any transactions stored in blockchains and all the historical transactions are auditable and traceable [2]. Transactions on the blockchain are recorded in the corresponding blocks, and the next block saves the hash value of the previous block to ensure the immutability of transaction data. Since there is no central node in the blockchain, some protocols are required to construct consensus among different nodes to ensure the ledger is always consistent [3]. After consensus is reached, valid blocks are added to the blockchain. Currently, blockchain technology is maturing at a fast pace [4], which has attracted the interest of industry and academia [5] due to the significant number of business benefits including transparency, traceability, security, and efficiency. For instance, blockchain technology has been applied to the areas of health care [6,7], logistics [8,9], and renewable energy [10].

Smart contracts were first proposed in the 1990s by Nick Szabo [11] and have become one of the most important features of blockchain technology. In a smart contract, contract clauses are defined as computer protocols, which allow credible agreements among participants without relying on third party authorities. When the trigger conditions are met, the terms embedded in smart contracts are automatically executed [3]. Smart contracts have been applied in many scenarios such as crowdfunding, voting, and medical research [12]. The most common development platform for smart contracts is Ethereum [13]. However,

smart contracts are also facing more and more attacks [14]. In June 2016, the DAO (the world's largest crowdfunding project deployed on Ethereum) was attacked by hackers, causing more than ETH three million to be separated from the DAO resource pool [15]. In September 2017, security vulnerabilities appeared in the Ethereum multi-signature wallet Parity, which resulted in the embezzlement of more than ETH 150000 (about USD 30 million) [16]. With such painful losses, how to ensure the security and correctness of smart contracts is becoming increasingly important.

In order to deal with such issues, many formal verification methods have been proposed and several tools have been developed to check the correctness of the program in smart contracts. In [17], the author proposed a verification method based on a programming language. They converted smart contracts written in Solidity into the F* language to check the security of the smart contracts. A framework named ZEUS in [18] was designed to automatically verify the correctness of smart contracts and their fairness by abstract interpretation and symbolic model detection. However, these rarely test the behavior of smart contracts interacting with clients under specific scenarios.

For this purpose, model checking is well adopted to verify whether smart contracts can interact with clients in a reliable way or not. Given a finite-state model of a system and a formal property, model checking is an automated technique, which systemically checks whether this property holds for that model [19]. The verification is performed with model checking tools such as NuSMV [20] and Spin [21]. The model checker checks automatically if each state of the model satisfies the specifications given by the user. In case there is a property that is not satisfied, the model checker provides a counterexample that can help us identify mistakes. On the other hand, if each state of the model satisfies the specification, the model is formally verified for that specific property.

The paper aims to establish a generic modeling method for Ethereum transactions, in order to apply a model-checking approach on smart contracts and their execution environment. A transition system is proposed for Ethereum transactions, and general formal models are built for smart contracts and clients based on it. Then, the method is applied to a study case in the Ethereum commodity market. All of the smart contracts and clients in the Ethereum commodity market are modeled, and the time characteristic of transaction is also taken into account. The formal models describe the interaction between smart contracts and clients in detail, which are presented with the Promela language. Moreover, some important properties are extracted from the transaction and specified using Linear Temporal Logic (LTL) formulae [22] and assertions, which are verified by the model checker Spin. The paper makes the following contributions:

- According to the specifications of blockchain transactions, we build general formal models for smart contracts and clients in the transaction. The method can describe the behavior the characteristics of smart contracts interacting with clients in the form of a state diagram under specific scenarios.
- A formal verification method of smart contracts based on Spin is proposed. The formal models of smart contracts and clients in the transaction can be presented with the Promela language. Spin can run the Promela model to simulate the process execution and verify whether the model caters to these LTL formulae and assertions.

The rest of the paper is organized as follows. Section 2 gives a presentation of the related proposals carried out in the area of smart contract modeling and verification. We propose a formal method to build the general model for Ethereum transactions in Section 3. Section 4 presents the considered study case in which the approach is applied and constructs the framework of the Ethereum commodity market. Thereafter, the transaction behavior of clients and smart contracts in the Ethereum commodity market are modeled in Section 5. In Section 6, the formal model of the transaction is presented with the Promela language, and Spin is used to simulate and verify the achieved model with its properties. Section 7 discusses the advantages and disadvantages of the proposed approach. Finally, the conclusions and future work are discussed in Section 8.

2. Related Works

Multiple efforts have been carried out in the current literature for the modeling and verification of smart contracts. The research related to the verification of smart contracts can be divided into two aspects, the first being related to the correctness of smart contracts and the second focusing on the security assurance of smart contracts [23].

2.1. Modeling of Smart Contracts

The immutability property of smart contracts establishes the non-alteration of blockchain network data after clauses of the contract are approved. For this reason, the design and development of smart contracts require more effort and care. Several solutions for smart contract modeling have been proposed to address the challenge.

Hamdaqa et al. [24] proposed a Domain-Specific Language (DSL) to help software developers create smart contracts and deploy them on a blockchain network. Software developers will be able to define models that will later generate code for different blockchain platforms including Ethereum, Hyperledger Composer, Azure, and DAML. This allows users to abstract which blockchain they are using and what peculiarities each one has.

In [25], the authors proposed such an approach that, in combination with the Unified Modeling Language (UML) Class and State machine diagrams, allows the smart contract and behavior logic to be modeled in several abstraction layers. The approach was evaluated by using three different smart contract examples from the official Solidity documentation. The results of the comparison of the code metrics and generated smart contracts tended to be quite similar. As a result, developers can focus on the structural and behavioral design of smart contracts, rather than on technique details.

In [26], the authors proposed an FSM-based approach for the design of secure smart contracts. They aimed at closing the semantic gap in Solidity by developing the FSolidM tool, which allows users to design a smart contract as a Finite State Machine (FSM), which is then automatically transformed into a Solidity smart contract. In addition, the framework extends a set of security plugins that can prevent some common vulnerabilities by patterns.

In [27], the authors developed a unifying model defining the essential components of fully specified legal smart contracts. The main goal of the approach is to compare and assess existing modeling languages for legal smart contracts' development with regard to the proposed unifying model. They introduced a set of eight existing modeling languages and demonstrated how the unifying model can be used as a basis for a holistic comparison of the languages' expressiveness.

2.2. Correctness Verification of Smart Contracts

The correctness verification is about respecting the specifications that determine how clients can interact with smart contracts and how smart contracts should behave when used correctly.

In [28], the authors used the theorem prover Isabelle/HOL and the existing EVM-formal model to verify the bytecode of smart contracts. The goal was to create a sound program logic and to use the resulting program logic for verification. A framework was created for expressing the EVM bytecode using logic, which was successfully applied to a case study. However, the framework does not support the full syntax of Solidity.

Grishchenko et al. [29] proposed the first sound static analyzer for the EVM bytecode. The tool supports reachability properties, which contain the most important security properties of smart contracts, such as single-entrancy and transaction environment dependency. This approach does not detect vulnerabilities, but provides guarantees that the code is free of certain ones.

In [30], the authors proposed a generic modeling method of smart-contract-based Ethereum applications, the model checking approach was then considered to verify the implementation's compliance with the specification. The proposed model is written in the NuSMV input language, and the properties to check are formalized into the temporal logicCTL [31]. It has three components: the kernel layer, which captures the blockchain

behavior, the application layer, which models the smart contracts, and the environment layer, which determines an execution framework for the application.

In [32], the authors proposed a tool chain for a seamless translation of smart contracts from the level of specifications toward the level of operations. The last step of this tool chain generates a code representation in Promela, which can be verified by the model checker Spin. However, the tool chain is limited to the correctness of individual contracts and needs an extension towards networks of smart contracts that interact.

2.3. Security Assurance of Smart Contracts

Any bugs or errors in the smart contract will become permanent once published on the blockchain and could cause huge economic losses. To avoid this, the security assurance aims at improving the security of smart contracts through vulnerability detection methods.

OYENTE [33] is a static analysis tool, which can detect security vulnerabilities. The tool uses symbolic execution to check for the following vulnerabilities: transaction ordering dependency, reentrancy, timestamp dependence, and unhandled exceptions.

Osiris [34] is a static analysis tool that combines symbolic execution and taint analysis to detect integer bugs in smart contracts. The tool covers three different types of integer bugs: arithmetic bugs, truncation bugs, and signedness bugs. Its architecture consists of three components: symbolic analysis, taint analysis, and integer error detection.

Chen et al. [35] developed a static analysis tool named Gasper, which focuses on gas costpatterns from existing smart contracts. Gasper looks for patterns such as dead code or expensive operations in loops to help contract developers reduce gas costs. The authors of [35] identified seven gas costpatterns.

3. The Proposed Approach

Transactions enabled by smart contracts are executed in accordance with the agreements made by participants. In order to ensure their trustworthiness, a method is proposed to build general formal models for smart contracts and clients in Ethereum transactions.

3.1. Transition System of Ethereum Transactions

The behavior of smart contracts interacting with clients is modeled by tuple $M = (S, \Sigma, \delta, I)$, where:

- S, the set of states.
- $\Sigma = Act \cup t^{'} \cup Comm$, the set of actions, where:
  - Act represents the internal action.
  - $t^{'}$ represents the change of time.
  - $Comm = \{c!v, c?x\}$, the communication action, where c denotes the channel, $c!v$ denotes the message v sent by channel c, and $c?x$ denotes that the variable x receives the message from channel c.
- $\delta \subseteq S \times Cond(V) \times \Sigma \times S$, the transition relation, where:
  - $Cond(V)$ represents the transition condition, where $V = var \cup t$, in which var is the set of internal variables and t represents the set of discrete time variables.
- $I \subseteq S$, the set of initial states.

We constructed the model $M = (S, \Sigma, \delta, I)$ to represent the the behavior of the Ethereum transaction. S represents the states of clients and smart contracts during the interaction. Σ represents the actions of clients interacting with smart contracts, which can be divided into three categories: the internal actions of clients and smart contracts, the communication actions between clients and smart contracts, and the change of time. δ represents the transition relation, which is related to transition conditions and actions. Transition conditions are related to the internal variables and time variables, and there is also unconditional transition in the model. The actions trigger the transition between

states including internal actions, communication actions, and the change of time. Time is abstracted as discrete time in the model.

3.2. General Model for Ethereum Transactions

Ethereum is an open-source public blockchain platform with a smart contract function. The roles of the transaction in Ethereum can be abstracted as clients and smart contracts. In order to model the behavior between clients and smart contracts, we considered a simple interaction scenario. In the scenario, there are two clients A and B, who build smart contracts to ensure the credibility of transactions. If client A wants to trade with client B, it needs to trigger the terms embedded in smart contracts. In the process of the transaction, smart contracts can terminate the transaction automatically in case of an abnormal situation.

As shown in Figure 1, the interaction scenario between clients and smart contracts is divided into four parts. In the *Initialization* phase, client A, client B, and the smart contracts are initialized to prepare the transaction. During the *Triggering* phase, once client A satisfies the trigger conditions, smart contracts automatically execute the terms and both parties enter the *Transaction* phase. In the *Transaction* phase, client A will trade with client B under the supervision of the smart contracts. When the transaction ends successfully, they enter the *Termination* phase. If clients violate the rules of the transaction, the smart contracts terminate the transaction and both parties enter the *Termination* phase. According to the analysis of the four stages in the transaction scenario, the general models of the clients and smart contracts can be built, respectively.

Let the client A model be $C_{m_a} = (S_1, \Sigma_1, \delta_1, I_1)$, where:

- $S_1 = \{start, request, wait, trade\}$.
- $\Sigma_1 = \{ch1, ch2, succeed, fail, end\}$.
- δ_1 is the transition relation, as shown in Figure 2.
- $I_1 = \{start\}$.

Let the client B model be $C_{m_b} = (S_2, \Sigma_2, \delta_2, I_2)$, where:

- $S_2 = \{idle, ready, trade\}$.
- $\Sigma_2 = \{ch1, ch3, fail, end\}$.
- δ_2 is the transition relation, as shown in Figure 3.
- $I_2 = \{idle\}$.

Let the smart contract model be $SC_m = (S_3, \Sigma_3, \delta_3, I_3)$, where:

- $S_3 = \{initiate, judge, execute\}$.
- $\Sigma_3 = \{ch2, ch3, succeed, fail, \}$.
- δ_3 is the transition relation, as shown in Figure 4.
- $I_3 = \{initiate\}$.

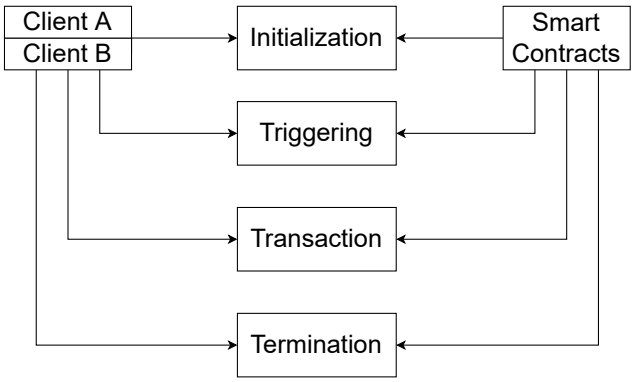

Figure 1. Interaction between clients and smart contracts.

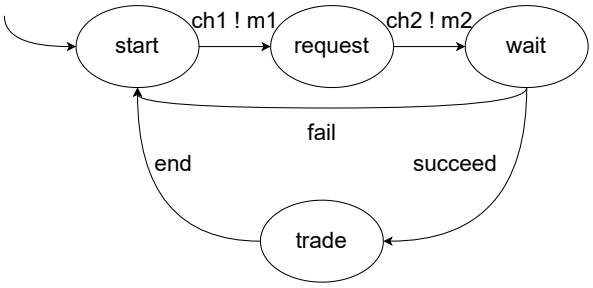

Figure 2. The client A model.

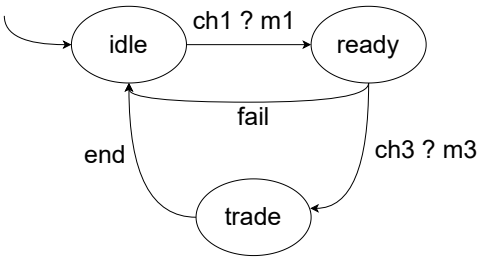

Figure 3. The client B model.

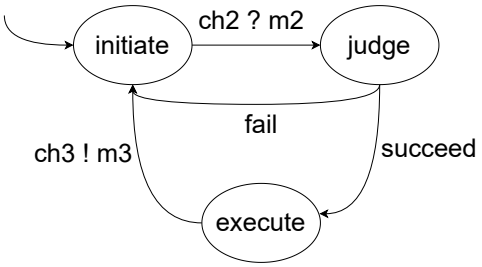

Figure 4. The smart contract model.

In the set of actions, $ch1$ is the communication actions between client A and B, $ch2$ is the communication actions between client A and the smart contracts, and $ch3$ is the communication actions between client B and the smart contracts. Meanwhile, we use *succeed* to represent the actions under normal circumstances; $fail$ represents the actions under abnormal circumstances; *end* represents the actions when the transaction is over.

4. The Ethereum Commodity Market

The method proposed in Section 2 was applied to a typical study case in the Ethereum commodity market. The framework of the transaction in the Ethereum commodity market was constructed.

4.1. The Ethereum Transaction

The roles in the study case include a factory, a supplier, a logistics company, and Ethereum. Participants can build smart contracts according to the agreement reached by them and deploy them on Ethereum. Due to the immutability of blockchain, transaction results cannot be changed once recorded. Therefore, the terms embedded in smart contracts must be credible, so that smart contracts are executed accurately. The transaction scenario is shown in Figure 5.

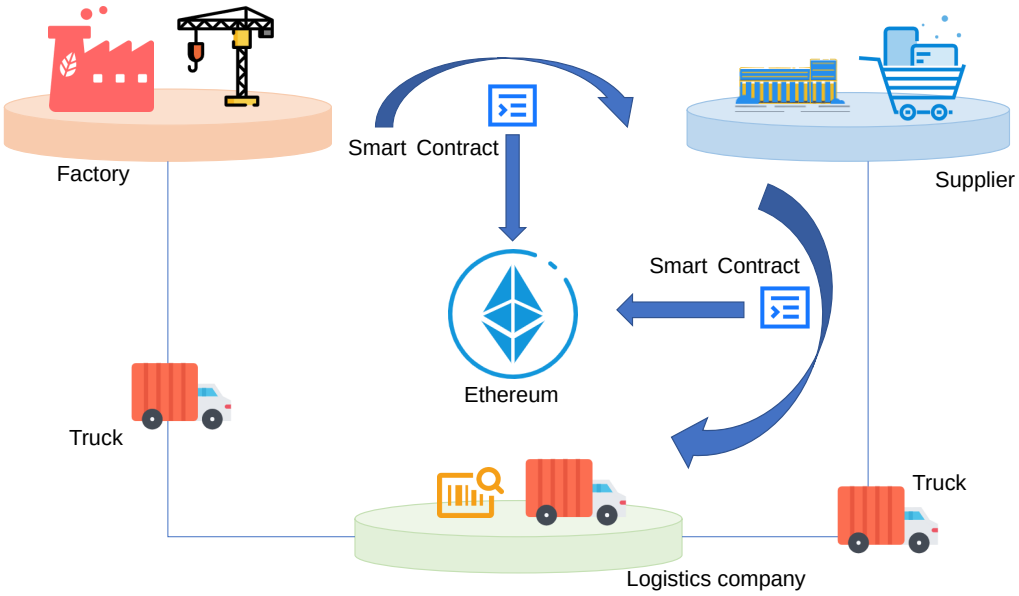

Figure 5. Transaction scenario in the Ethereum commodity market.

There are two transactions. The first transaction is requested by the factory to the supplier. After the supplier accepts the request, the factory judges whether the account balance meets the payment. If satisfied, the factory transfers the payment to the smart contract account. When the smart contract receives the payment, it notifies the supplier of the delivery.

The second transaction is executed by the supplier and the logistics company. The supplier transports goods to the factory by renting trucks from the logistics company. The offline trading time is from 9 a.m. to 5 p.m. If the logistics company has spare trucks and the current time is suitable, it accepts the request. After the logistics company accepts the request, the supplier judges whether the account balance meets the fare. If satisfied, the supplier transfers the fare to the smart contract account. When the smart contract receives the fare, it notifies the logistics company to transport the goods.

Only when the identity information and working hours of the truck are accurate can the delivery be successful. Once the factory receives the goods successfully, it confirms the receipt and the smart contract transfers the payment to the supplier. After that, the supplier confirms the shipment, and then, the smart contract transfers the fare to the logistics company, which makes the transaction end successfully.

The successful execution of the transaction needs to satisfy many conditions, and any exception leads to failure. For example, if the logistics company has no spare truck, it refuses the request from the supplier, and then, the supplier informs the factory that the transaction has failed. When the smart contract receives the confirmation from the supplier, it returns the payment to the factory. Whether normal or abnormal circumstances, the accurate execution of smart contracts ensures the credibility of the transaction.

It is worth mentioning that the whole procedure of the transaction is conducted without the intervention of a third party. Smart contracts stored in Ethereum can be automatically triggered in a decentralized way, which cuts down the service cost from the third party and improves the efficiency of the business process.

4.2. Framework

The framework is shown in Figure 6. We abstracted the whole transaction process as three clients and two smart contracts. The three clients are the factory, the supplier, and the logistics company, which are represented as $factory$, $supplier$, and $truck$ in the model. Each smart contract corresponds to a transaction. The smart contract between $factory$ and $supplier$ is represented as SCa, and the smart contract between $supplier$ and $truck$

is represented as *SCb*. At the beginning, *factory* applies for a transaction with *supplier*. When *factory* triggers the terms embedded in *SCa*, the transaction between *factory* and *supplier* officially begins. Afterwards, *supplier* applies for transaction with *truck*. When *supplier* triggers the terms embedded in *SCb*, the transaction between *supplier* and *truck* officially begins.

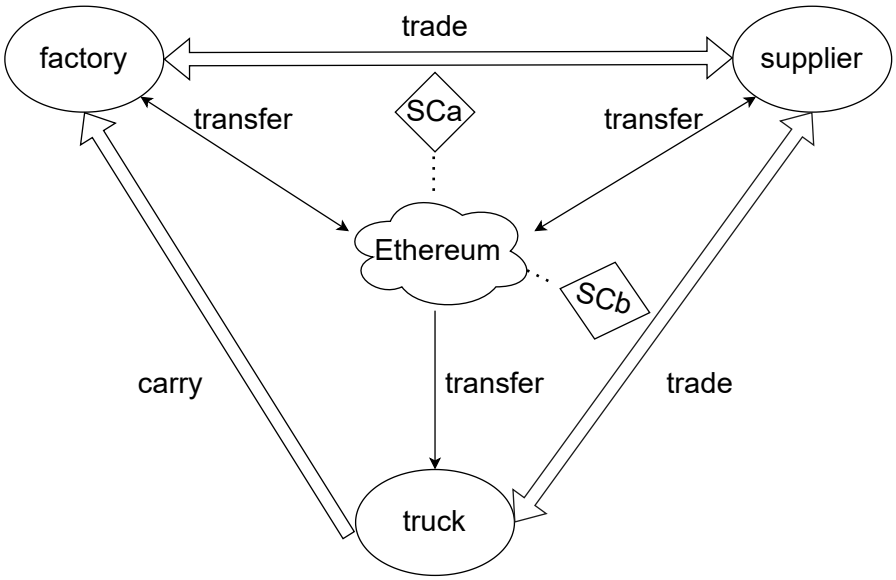

Figure 6. Framework of Ethereum commodity market.

5. Modeling the Ethereum Commodity Market

In order to verify the correctness of smart contracts, the reliability of the transaction is analyzed. The transaction behaviors of *factory*, *supplier*, *truck*, and *Ethereum* are modeled, respectively.

5.1. Factory Modeling

The behaviors of *factory* mainly include requesting a transaction, transferring the payment to *SCa*, and receiving the goods. The *factory* model is shown in Figure 7, and the main behaviors of it are described as follows:

- Requesting a transaction: In this phase, *factory* sends a transaction request to *supplier*. When *supplier* receives the transaction request, *factory* provides the order quantity to *supplier*. The states defined for *factory* include *start*, *fa_request*, *fa_inter*1, and *q_goods*.
- Transferring the payment to *SCa*: In this phase, *factory* checks whether the account balance is sufficient to pay for the goods. If satisfied, the payment is transferred to the smart contract account. The state defined for *factory* is *transfer_sca*.
- Getting the ID of *truck*: In this phase, *factory* needs to obtain the ID of *truck*, so that it can confirm the identity of *truck* when it receives the goods. The state defined for *factory* is *rec_goods*.
- Receiving the goods: When *factory* receives the goods, it confirms the receipt to *SCa*. The states defined for *factory* include *check_deliT*, *fa_rec_id*, and *check_id*.
- Confirming receipt: When *factory* receives the goods, it confirms the receipt to *SCa*. The state defined for *factory* is *get_success*.

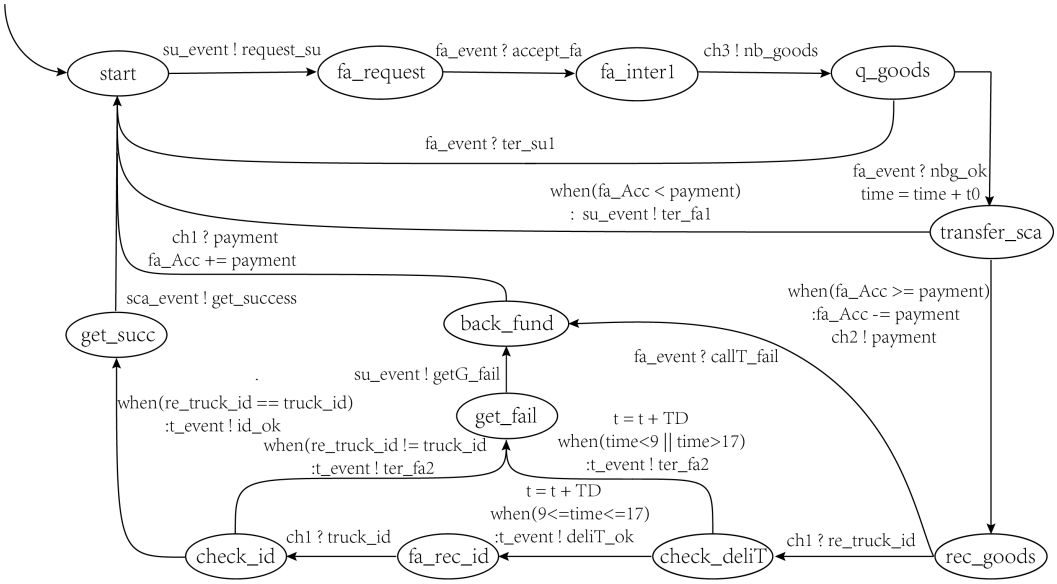

Figure 7. The factory model.

5.2. Supplier Modeling

The behaviors of *supplier* mainly include renting *truck*, transferring the fare to *SCb*, and delivering the goods. The *supplier* model is shown as Figure 8, and the main behaviors of it are described as follows:

- Responding to the request of $factory$: In this phase, *supplier* responds to the transaction request sent by $factory$ and reviews the order quantity. The states defined for *supplier* are *idle*, *respond_fa*, *su_inter*1, *check_nbg*, and *su_inter*2.
- Renting *truck*: Because of delivering the goods to $factory$, *supplier* sends a transaction request to *truck*. The states defined for *supplier* include *rent_truck*, *su_inter*3, and $verify\_carryT$.
- Transferring the fare to *SCb*: In this phase, *supplier* checks whether the account balance meets the cost of renting *truck*. If satisfied, it transfers the fare to the smart contract account; otherwise, it declares that the transaction has failed and notifies *SCa* to refund the payment. The state defined for *supplier* is $transfer\_scb$.
- Delivering the goods: After renting *truck* successfully, *supplier* delivers the goods to $factory$ by *truck*. When $factory$ confirms the receipt, *supplier* receives the payment from *SCa* and confirms shipment to *SCb*. The states defined for *supplier* include *check_pickT*, *su_rec_id*, and $feed\_fa$.
- Sending the ID of *truck* to $factory$: In this phase, *supplier* needs to send the identity information of *truck* to $factory$, so that $factory$ can recognize *truck* correctly. The state defined for *supplier* is *su_se_id*.
- Confirming the success of shipment: When *supplier* learns that $factory$ has successfully received the goods, it needs to confirm the success of shipment to *SCb*. The states defined for *supplier* are *shipping* and *ship_succ*.

5.3. Truck Modeling

The behaviors of *truck* mainly include picking up the goods and carrying the goods. The *truck* model is shown in Figure 9, and the main behaviors of it are described as follows:

- Responding to the request of *supplier*: In this phase, *truck* responds to the request of *supplier* by judging its current state. The states defined for *truck* are *initiate*, *respond_su*, and $check\_carryT$.

- Picking up the goods: If *truck* arrives during the working hours, it can pick up the goods from *supplier* successfully. The states defined for *truck* include *pick*, $verify\_pickT$, and $t\_se\_id\_su$.
- Carrying the goods: If the delivery time and identity information of *truck* are correct, it can deliver the goods to $factory$ successfully. The states defined for *truck* include $verify\_deliT$, $t\_se\_id\_fa$, and $carry\_succ$.

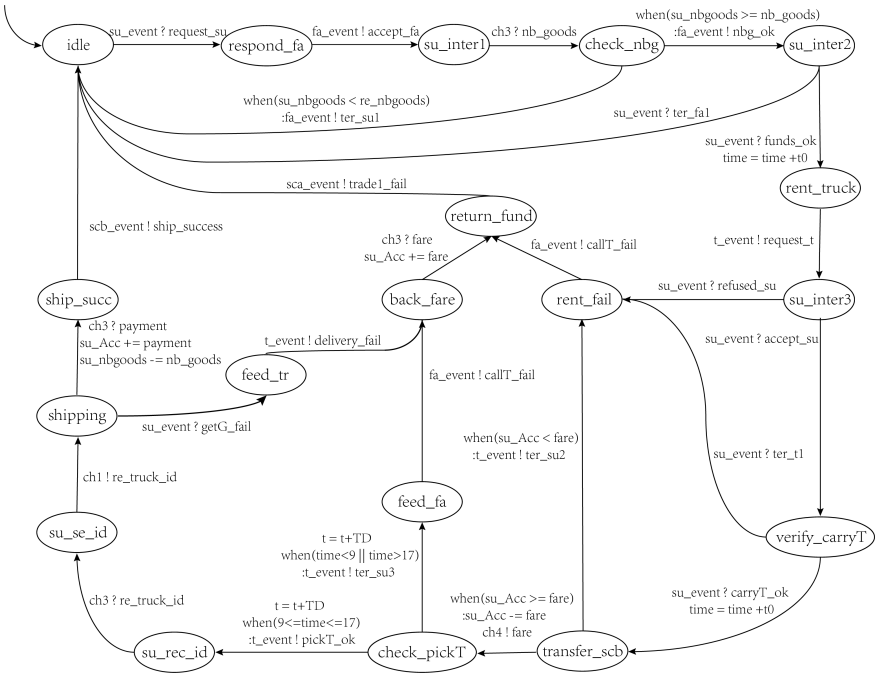

Figure 8. The supplier model.

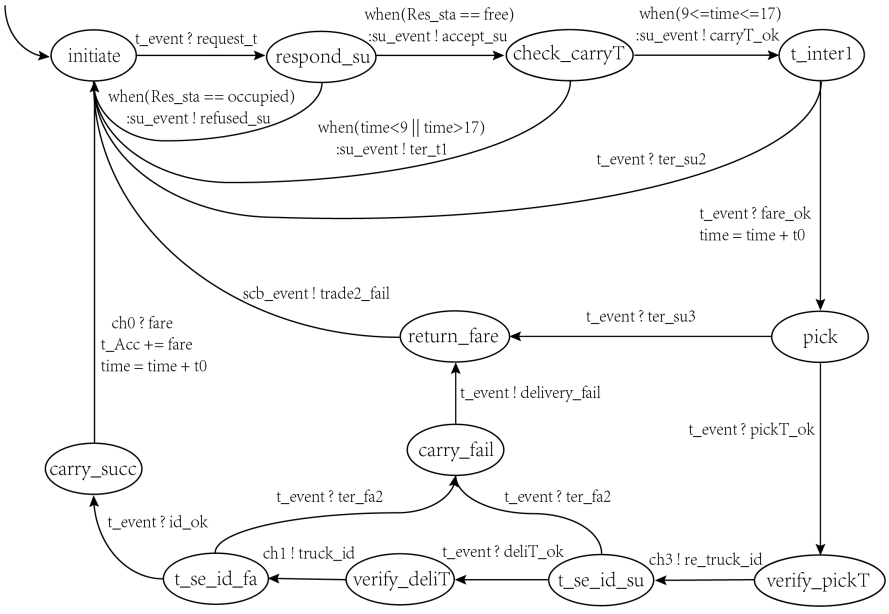

Figure 9. The truck model.

5.4. Ethereum Modeling

Ethereum is the platform on which transactions take place and makes the transaction secure and trustworthy through smart contracts. There are two smart contracts deployed on *Ethereum* in the transaction. The *Ethereum* model is shown in Figure 10.

The behaviors of *SCa* mainly include notifying *supplier* of delivery and transferring the payment to *supplier*. These behaviors are described as follows:

- Waiting for $factory$ to transfer the payment: The safety of the transaction between $factory$ and $supplier$ is guaranteed by *SCa*. At the beginning, *SCa* waits for $factory$ to transfer the payment to trigger the transaction. The state defined for *SCa* is *begin*.
- Notifying *supplier* of delivery: Once *SCa* receives the payment, it notifies *supplier* of the delivery. The states defined for *SCa* are *sca_re_f* and *sca_fund_ok*.
- Transferring the payment to *supplier*: If $factory$ confirms the receipt of the goods, *SCa* transfers the payment to *supplier*; otherwise, it returns the payment to $factory$ eventually. The states defined for *SCa* are *pay_su* and *return_fa*.

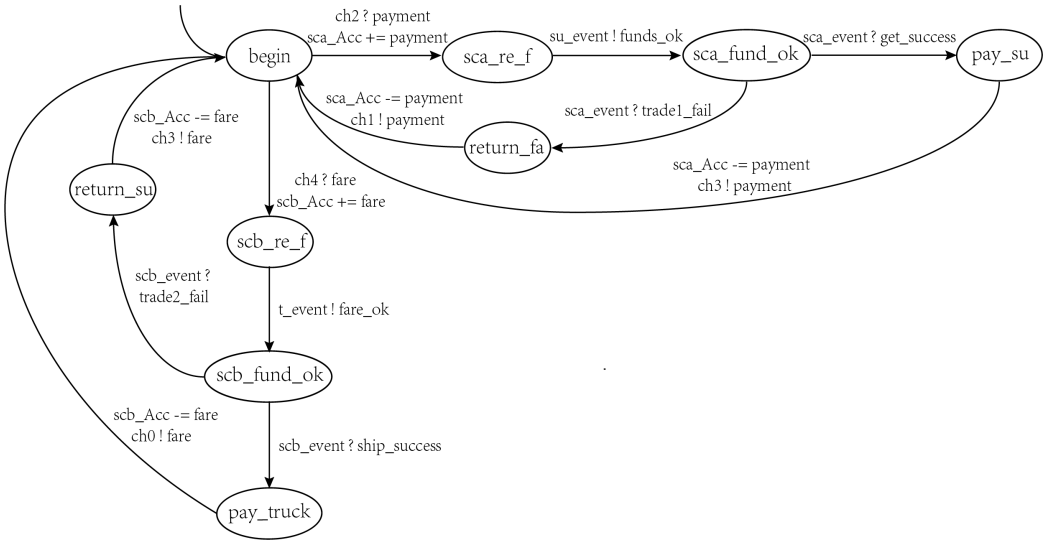

Figure 10. The Ethereum model.

The behaviors of *SCb* mainly include notifying *truck* of the work and transferring the fare to *truck*. These behaviors are described as follows:

- Waiting for *supplier* to transfer the fare: The safety of the transaction between *supplier* and *truck* is guaranteed by *SCb*. At the beginning, *SCb* waits for *supplier* to transfer the fare to trigger the transaction. The state defined for *SCb* is *begin*.
- Notifying *truck* of work: Once *SCb* receives the fare, it notifies *truck* of the work. The states defined for *SCb* are *scb_re_f* and *scb_fund_ok*.
- Transferring the fare to *truck*: If *supplier* confirms the shipment, *SCb* transfers the fare to *truck*; otherwise, it returns the fare to *supplier* eventually. The states defined for *SCb* are *pay_truck* and *return_su*.

6. Implementation

For the purpose of verifying the correctness of the models, we adopted two methods: simulation and formal verification. The first one was completed by executing the models and observing the simulation results. The second one is a static method. Some properties are extracted from specification and verified with the model checker Spin.

6.1. Model Checker Spin

Spin is a model checker—a software tool for verifying models of physical systems, in particular computerized systems. First, a model is written that describes the behavior of the system; then, the correctness properties that express the requirements of the system's behavior are specified; finally, the model checker is run to check if the correctness properties hold for the model and, if not, to provide a counterexample: a computation that does not satisfy a correctness property. Promela is the language that is used for writing models in Spin.

6.2. Modeling the Ethereum Commodity Market with Spin

The transaction model was implemented with the Promela language, which is the modeling language in the tool Spin. Smart contracts and clients are abstracted as processes in Promela. The important parameters and overall structure of the model are described as follows:

- Each state name is translated into an *mtype* value, and each variable representing the current state in a state machine is also declared an *mtype* variable. The following is the initial declaration of the state machines' current state:

```
mtype fa_currState = start;
mtype su_currState = idle;
mtype tr_currState = initiate;
mtype sca_currState = begin;
mtype scb_currState = begin;
```

- The interact signals between state machines are sent and received through channels. The type of all channels is abstract as *chan*, and the type of all messages is abstracted as *mtype*. In addition, the size of each channel is set to 0 to ensure synchronization between processes. The declaration of the channels is as follows:

```
chan fa_event = [0] of {mtype};
chan su_event = [0] of {mtype};
chan t_event = [0] of {mtype};
chan sca_event = [0] of {mtype};
chan scb_event = [0] of {mtype};
```

- The transition function and the step function are defined for each state. The transition function represents the procedure of the model migrating from one state to the next. The step function represents the change in the model state, which is called by the transition function. The functions of each state are translated into *inline* macros. The transition function of the *truck* model in state *check_carryT* is declared as follows:

```
inline T_check_carryT(){
 if
 :: (time>=9 && time<=17) -> su_event ! carryT_ok;
        S_t_inter1_entry();
 :: else -> su_event ! ter_t1;
    S_initiate_entry();
 fi
}
```

- All state transition functions of each model are put into a region function. By calling the function in a loop, each process can find the current state and determine the operation that should be performed to move to the next state. The region functions of *factory*, *supplier*, *truck*, *SCa*, and *SCb* are, respectively, function *R_fa*, function *R_su*, function *R_tr*, function *R_sca*, and function *R_scb*, which are translated into *inline* macros. The region function of *SCa* is declared as follows:

```
inline R_sca(){
 if
 :: (sca_currState == begin) -> T_sca_begin();
 :: (sca_currState == sca_re_f) -> T_sca_re_f();
 :: (sca_currState == sca_fund_ok) -> T_sca_fund_ok();
 :: (sca_currState == pay_su) -> T_pay_su();
 :: (sca_currState == return_fa) -> T_return_fa();
 fi
}
```

- There are five executable processes *Fa_stm*, *Su_stm*, *Tr_stm*, *SCa_stm*, and *SCb_stm*, which describe the behavior of *factory*, *supplier*, *truck*, *SCa*, and *SCb* in the transaction. Besides the above processes, another process *init* is created. This process declares that the behaviors of processes *Fa_stm*, *Su_stm*, *Tr_stm*, *SCa_stm*, and *SCb_stm* are active in the initial system. Keyword *run* starts these processes, which run concurrently in the system from then on. The process *init* is declared as follows:

```
init {
 run Fa_stm();
 run Su_stm();
 run Tr_stm();
 run SCa_stm();
 run SCb_stm();
}
```

6.3. Simulation

The sequence of messages shown in Figure 11 represents a simulation of the transaction. The five vertical lines show the life cycle of process *Fa_stm*, process *Su_stm*, process *Tr_stm*, process *SCa_stm*, and process *SCb_stm*, respectively. The slashes indicate the steps performed by each process in turn.

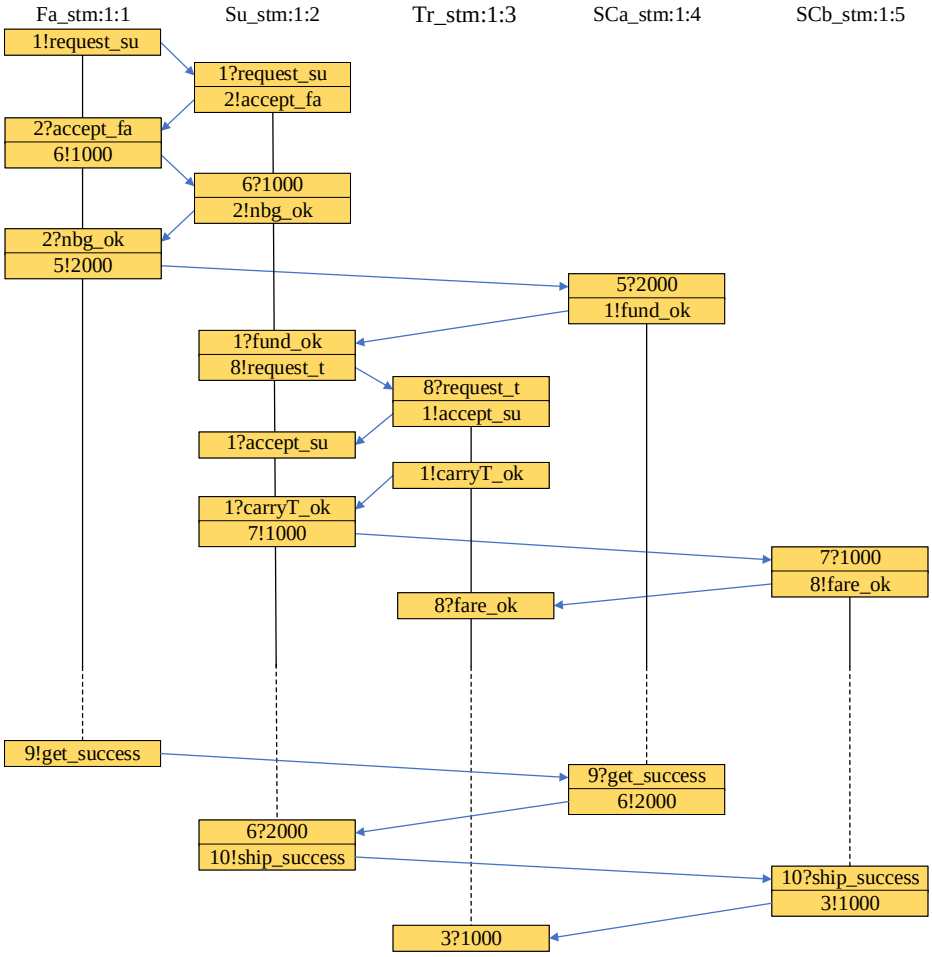

Figure 11. Message sequence diagrams.

As shown in the simulation, *factory* sends the message *request_su* to *supplier* via the channel at the beginning, which represents the transaction between *factory* and *supplier*. *supplier* responds to *factory*'s transaction request. According to the agreement embedded

in the smart contracts, $factory$ transfers the amount to the contract account. *SCa* then sends the message $fund\_ok$ to $supplier$, which means $supplier$ can send the goods to $factory$ safely. Therefore, $supplier$ sends the message *request_t* to *truck*, which represents the transaction between $supplier$ and *truck*. *truck* responds to $supplier$'s transaction request. Afterwards, *SCb* sends the message $fare\_ok$ to *truck*, which means *truck* can deliver the goods safely. When $factory$ has successfully received the goods, it sends the message *get_success* to *SCa*, which in turn transfers the payment to $supplier$. Then, $supplier$ sends the message *ship_success* to *SCb*, which in turn transfers the fare to *truck*. The experiment simulates the procedure of message passing among processes, which complies with the specification of the transaction.

6.4. Verification

Based on the specification in the Ethereum commodity market, some properties were extracted. These properties include deadlock, invariant, safety, and liveness.

6.4.1. Properties

The correct execution of smart contracts should ensure the orderly execution of the transaction, and the funds should flow safely among accounts during the transaction. The specific properties to be verified are as follows:

Property 1. Deadlock Free (DF, Deadlock) Deadlock is not allowed in the model, which is the situation in which two or more processes are waiting for each other's resource in a circular chain. This property could be checked in Spin by default without stating it as assertions or LTL formulae.

Property 2. Total Account Balance (TAB, Invariant) The total amount of the factory, the supplier, the truck, and the smart contracts' account should remain unchanged throughout the transaction. The property is described in LTL as follows:

$$[](fa\_Acc + su\_Acc + t\_Acc + sca\_Acc + scb\_Acc == invar)$$

Property 3. Supplier rents truck After Factory transfer (SAF, Safety) During the transaction, it is impossible for the supplier to rent the truck before the factory transfers the payment to SCa. The property is described in LTL as follows:

$$[]((!rent\_truck U pay\_sca)||!pay\_sca)$$

Property 4. Truck receives fare After Supplier receives payment (TAS, Safety) During the transaction, it is impossible for the truck to receive the fare before the supplier receives the payment. The property is described in LTL as follows:

$$[]((!tr\_recfund U su\_recfund)||!su\_recfund)$$

Property 5. Total Balance Remains Same (TBRS, Liveness) After the transaction, the total amount of the factory, the supplier, and the truck's account will eventually remain unchanged. The property is described in LTL as follows:

$$[](trade\_finish \longrightarrow <> (fa\_Acc + su\_Acc + t\_Acc == invar))$$

Property 6. Balance Reaches Correct Value (BRCV, Liveness) If the transaction is successful, the factory account balance is the initial balance minus the payment, the supplier account balance is the initial balance plus the payment minus the fare, and the truck account balance is the initial balance plus the fare. The property is described in LTL as follows:

$$\begin{aligned}[](trade\_success \longrightarrow & <> (fa\_Acc == fa\_AccInit - pay\_goods)\&\& \\ & <> (su\_Acc == su\_AccInit - fare\_goods + pay\_goods)\&\& \\ & <> (t\_Acc == t\_AccInit + fare\_goods))\end{aligned}$$

Property 7. Truck Working Time (TWT, Liveness) Under normal trading conditions, when the truck delivery is successful, the time should be between 9 a.m. and 5 p.m. We express it with the assertion as follows:

$$assert(time >= 9\&\&time <= 17)$$

6.4.2. Analysis

Spin supports user-defined properties specified as *assert* and LTL formulae, while LTL formulae need to be translated into Promela *never* claims automatically [36]. The verification results of each property are summarized in Table 1. For each property, we present the kind of properties, (i.e., default property (D.P.), LTL formula, or Assertion (A)) and the number of states stored and the transitions, which represent unique global system states stored in the state space and transitions explored in the search. Finally, whether the model Passes (P) the verification of the property or not (F) is determined. The result shows that all the properties are proven to be valid in the model.

Table 1. The result of verification.

Property	Name	Kind	States Stored	Transitions Usage	Result
Deadlock	DF	D.P.	1,084,815	3,301,742	P
Invariant	TAB	LTL	759,662	2,269,701	P
Safety	SAF	LTL	759,662	2,269,701	P
	TAS	LTL	759,662	2,269,701	P
Liveness	TBRS	LTL	759,662	2,269,701	P
	BRCV	LTL	967,856	2,869,858	P
	TWT	A	1,084,815	2,216,927	P

Firstly, deadlock does not exist in the model. The invariant property ensures that the total amount of accounts is invariable in the whole transaction procedure. The safety properties guarantee the agreement reached by the participants. SAF verifies the order of the factory transferring the payment to the smart contract and the supplier renting the truck, and TAS verifies the order of the supplier receiving the payment and the truck receiving the fare. The liveness properties prove that the account amount is correct eventually and the transaction time complies with the specification. TBRS verifies that the total account amount of the factory, the supplier, and the truck remain unchanged at the end of the transaction. BRCV verifies that the account amount of the factory, the supplier, and the truck would eventually reach the correct value when the transaction ends successfully, and TWT verifies that the truck can only deliver successfully during working hours. Based on the above properties satisfied in the model, the behaviors of the smart contracts interacting with clients are safe, which makes the transaction credible when the transaction results are recorded on the decentralized and tamper-proof blockchain.

7. Discussion and Limitations

The use of blockchain comes with a number of advantages, but its implementation is accompanied by development difficulties, such as dealing with the immutability of smart contracts. There are many methods for designing smart contracts correctly before they are deployed on the blockchain. In [24], the authors proposed iContractML 2.0: a blockchain-agnostic framework for modeling and deploying smart contracts on multiple blockchain platforms. The main components of the iContractML 2.0 modeling framework include the concrete syntax, validation rules, and transformation templates. In addition to supporting multiple blockchain platforms, it also can model the behavior of smart contracts. The developers define models that will later generate code for different blockchain platforms.

In [25], the research aimed to demonstrate that the principles of Model-Driven Architecture (MDA) and Unified Modeling Language (UML) diagrams can be successfully applied to facilitate the development of smart contracts for blockchain. The authors presented the algorithm for transformation from the Platform Independent Model (PIM) to the Ethereum Solidity-Platform-Specific Model (PSM) and Solidity smart contract code generated from the specified PSM. The approach enables the developers to focus on the structural and behavioral design of smart contracts.

In this paper, we proposed a formal method for smart contract verification with Spin. A general modeling method for smart contracts in the transaction was established, specifically expressed in the form of the transition system and state machine. Then, formal models can be presented in Promela, which can be verified by the model checker Spin. Compared with the above methods, our approach can model the behavior of smart contracts interacting with clients and verify whether the smart contracts comply with their specifications for a given behavior of the stakeholders. However, our approach cannot generate the correct smart contracts automatically; it can only model the designed smart contracts and verify them. Besides, it does not apply to the situation where there are vulnerabilities existing in the program of smart contracts. In this case, combining our approach with vulnerability detection tools is better.

8. Conclusions

Whether the behavior between smart contracts and clients complies with the corresponding specifications needs to be checked. In this paper, we applied model checking to verify the correctness of smart contracts. The transition system of Ethereum transactions was proposed, and the general models for smart contracts and clients were established. Moreover, the approach was applied to a study case in the Ethereum commodity market, and formal models of smart contracts interacting with clients were built. Thereafter, the Promela model in Spin was implemented. Models of the study case were simulated. Finally, some properties such as deadlock, invariant, safety, and liveness were extracted from the specification of the Ethereum commodity market. The experiment showed that our model satisfies these properties. Through formal modeling and analysis of smart contracts, the behavior of smart contracts interacting with clients is reliable.

In the future, our work has two main directions. Firstly, we will work on modeling and analyzing more challenging commodity transactions in Ethereum, such as introducing malicious attacks from the outside. Secondly, since smart contracts are written in program code, there may be some vulnerabilities in the program, such as integer overflow, memory leak, and out of bounds. We can develop some tools to detect vulnerabilities in the code, which is a challenging and meaningful work.

Author Contributions: Conceptualization, Z.Y. and J.G.; methodology, Z.Y. and J.G.; validation, Z.Y., M.D. and J.G.; formal analysis, Z.Y.; investigation, Z.Y. and M.D.; resources, Z.Y.; writing—original draft preparation, Z.Y.; writing—review and editing, Z.Y., M.D. and J.G.; supervision, M.D. and J.G.; project administration, J.G.; funding acquisition, J.G. All authors have read and agreed to the published version of the manuscript.

Funding: This work is supported by the National Key Research and Development Program (2019YFB2102600), and Shanghai Trusted Industry Internet Software Collaborative Innovation Center.

Institutional Review Board Statement: Not applicable.

Informed Consent Statement: Not applicable.

Data Availability Statement: Not applicable.

Conflicts of Interest: The authors declare no conflict of interest.

References

1. Yaga, D.; Mell, P.; Roby, N.; Scarfone, K. Blockchain technology overview. *arXiv* **2019**, arXiv:1906.11078.
2. Zheng, Z.; Xie, S.; Dai, H.N.; Chen, W.; Chen, X.; Weng, J.; Imran, M. An overview on smart contracts: Challenges, advances and platforms. *Future Gener. Comput. Syst.* **2020**, *105*, 475–491. [CrossRef]
3. Zheng, Z.; Xie, S.; Dai, H.; Chen, X.; Wang, H. An overview of blockchain technology: Architecture, consensus, and future trends. In Proceedings of the 2017 IEEE International Congress on Big Data (BigData Congress), Boston, MA, USA, 11–14 December 2017; pp. 557–564.
4. Belchior, R.; Vasconcelos, A.; Guerreiro, S.; Correia, M. A survey on blockchain interoperability: Past, present, and future trends. *ACM Comput. Surv. (CSUR)* **2021**, *54*, 1–41. [CrossRef]
5. Zohar, A. Bitcoin: Under the hood. *Commun. ACM* **2015**, *58*, 104–113. [CrossRef]
6. Haleem, A.; Javaid, M.; Singh, R.P.; Suman, R.; Rab, S. Blockchain technology applications in healthcare: An overview. *Int. J. Intell. Netw.* **2021**, *2*, 130–139. [CrossRef]
7. Farouk, A.; Alahmadi, A.; Ghose, S.; Mashatan, A. Blockchain platform for industrial healthcare: Vision and future opportunities. *Comput. Commun.* **2020**, *154*, 223–235. [CrossRef]
8. Raja Santhi, A.; Muthuswamy, P. Influence of blockchain technology in manufacturing supply chain and logistics. *Logistics* **2022**, *6*, 15. [CrossRef]
9. He, M.; Wang, H.; Sun, Y.; Bie, R.; Lan, T.; Song, Q.; Zeng, X.; Pustiněk, M.; Qiu, Z. T2L: A traceable and trustable consortium blockchain for logistics. *Digit. Commun. Netw.* **2022**. [CrossRef]
10. Wang, X.; Yang, W.; Noor, S.; Chen, C.; Guo, M.; van Dam, K.H. Blockchain-based smart contract for energy demand management. *Energy Procedia* **2019**, *158*, 2719–2724. [CrossRef]
11. Szabo, N. The idea of smart contracts. *Nick Szabo's Pap. Concise Tutor.* **1997**, *6*, 199.
12. Vujičić, D.; Jagodić, D.; Ranđić, S. Blockchain technology, bitcoin, and Ethereum: A brief overview. In Proceedings of the 2018 17th International Symposium Infoteh-Jahorina (Infoteh), East Sarajevo, Bosnia and Herzegovina, 21–23 March 2018; pp. 1–6.
13. Buterin, V. A next-generation smart contract and decentralized application platform. *White Pap.* **2014**, *3*, 2-1.
14. Atzei, N.; Bartoletti, M.; Cimoli, T. A survey of attacks on ethereum smart contracts (sok). In Proceedings of the International Conference on Principles of Security and Trust, Uppsala, Sweden, 24–25 April 2017; Springer: Berlin/Heidelberg, Germany, 2017; pp. 164–186.
15. Kushwaha, S.S.; Joshi, S.; Singh, D.; Kaur, M.; Lee, H.N. Systematic review of security vulnerabilities in ethereum blockchain smart contract. *IEEE Access* **2022**, *10*, 6605–6621.
16. Liu, J.; Liu, Z. A survey on security verification of blockchain smart contracts. *IEEE Access* **2019**, *7*, 77894–77904. [CrossRef]
17. Bhargavan, K.; Delignat-Lavaud, A.; Fournet, C.; Gollamudi, A.; Gonthier, G.; Kobeissi, N.; Kulatova, N.; Rastogi, A.; Sibut-Pinote, T.; Swamy, N.; et al. Formal verification of smart contracts: Short paper. In Proceedings of the 2016 ACM Workshop on Programming Languages and Analysis for Security, Vienna, Austria, 24 October 2016; pp. 91–96.
18. Kalra, S.; Goel, S.; Dhawan, M.; Sharma, S. Zeus: Analyzing safety of smart contracts. In Proceedings of the 25th Annual Network and Distributed System Security Symposium (NDSS), San Diego, CA, USA, 18–21 February 2018; pp. 1–12.
19. Baier, C.; Katoen, J.P. *Principles of Model Checking*; MIT Press: Cambridge, MA, USA, 2008.
20. Cimatti, A.; Clarke, E.; Giunchiglia, E.; Giunchiglia, F.; Pistore, M.; Roveri, M.; Sebastiani, R.; Tacchella, A. Nusmv 2: An opensource tool for symbolic model checking. In Proceedings of the International Conference on Computer Aided Verification, Copenhagen, Denmark, 27–31 July 2002; Springer: Berlin/Heidelberg, Germany, 2002; pp. 359–364.
21. Holzmann, G.J. The model checker SPIN. *IEEE Trans. Softw. Eng.* **1997**, *23*, 279–295. [CrossRef]
22. Shoukry, Y.; Nuzzo, P.; Balkan, A.; Saha, I.; Sangiovanni-Vincentelli, A.L.; Seshia, S.A.; Pappas, G.J.; Tabuada, P. Linear temporal logic motion planning for teams of underactuated robots using satisfiability modulo convex programming. In Proceedings of the 2017 IEEE 56th Annual Conference on Decision and Control (CDC), Melbourne, Australia, 12–15 December 2017; pp. 1132–1137.
23. Almakhour, M.; Sliman, L.; Samhat, A.E.; Mellouk, A. Verification of smart contracts: A survey. *Pervasive Mob. Comput.* **2020**, *67*, 101227. [CrossRef]
24. Hamdaqa, M.; Met, L.A.P.; Qasse, I. iContractML 2.0: A domain-specific language for modeling and deploying smart contracts onto multiple blockchain platforms. *Inf. Softw. Technol.* **2022**, *144*, 106762. [CrossRef]
25. Jurgelaitis, M.; Butkienė, R. Solidity Code Generation From UML State Machines in Model-Driven Smart Contract Development. *IEEE Access* **2022**, *10*, 33465–33481. [CrossRef]
26. Mavridou, A.; Laszka, A. Designing secure ethereum smart contracts: A finite state machine based approach. In Proceedings of the International Conference on Financial Cryptography and Data Security, Nieuwpoort, Curacao, 26 February–2 March 2018; Springer: Berlin/Heidelberg, Germany, 2018; pp. 523–540.
27. Ladleif, J.; Weske, M. A unifying model of legal smart contracts. In Proceedings of the International Conference on Conceptual Modeling, Vienna, Austria, 3–6 November 2019; Springer: Berlin/Heidelberg, Germany, 2019; pp. 323–337.
28. Amani, S.; Bégel, M.; Bortin, M.; Staples, M. Towards verifying ethereum smart contract bytecode in Isabelle/HOL. In Proceedings of the 7th ACM SIGPLAN International Conference on Certified Programs and Proofs, Los Angeles, CA, USA, 8–9 January 2018; pp. 66–77.
29. Grishchenko, I.; Maffei, M.; Schneidewind, C. *Ethertrust: Sound Static Analysis of Ethereum Bytecode*; Technische Universität Wien: Vienna, Austria, 2018; pp. 1–41.

30. Nehai, Z.; Piriou, P.Y.; Daumas, F. Model-checking of smart contracts. In Proceedings of the 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), Halifax, NS, Canada, 30 July–3 August 2018; pp. 980–987.
31. Browne, M.C.; Clarke, E.M.; Grümberg, O. Characterizing finite Kripke structures in propositional temporal logic. *Theor. Comput. Sci.* **1988**, *59*, 115–131. [CrossRef]
32. Osterland, T.; Rose, T. Model checking smart contracts for ethereum. *Pervasive Mob. Comput.* **2020**, *63*, 101129. [CrossRef]
33. Luu, L.; Chu, D.H.; Olickel, H.; Saxena, P.; Hobor, A. Making smart contracts smarter. In Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, Vienna, Austria, 24–26 October 2016; pp. 254–269.
34. Torres, C.F.; Schütte, J.; State, R. Osiris: Hunting for integer bugs in ethereum smart contracts. In Proceedings of the 34th Annual Computer Security Applications Conference, San Juan, PR, USA, 3–7 December 2018; pp. 664–676.
35. Chen, T.; Li, X.; Luo, X.; Zhang, X. Under-optimized smart contracts devour your money. In Proceedings of the 2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER), Klagenfurt, Austria, 20–24 February 2017; pp. 442–446.
36. Neumann, R. Promela formalization. *Arch. Form. Proofs* **2014**, *2014*, 1–103.