Formal Verification of Smart Contracts from the Perspective of Concurrency

Meixun Qu[1], Xin Huang[1](✉), Xu Chen[2], Yi Wang[3], Xiaofeng Ma[3], and Dawei Liu[1]

[1] Xi'an Jiaotong-Liverpool University, Suzhou, China
Meixun.Qu16@student.xjtlu.edu.cn, {Xin.Huang,Dawei.Liu}@xjtlu.edu.cn

[2] Tongji Blockchain Research Institute, Suzhou, China
chenxu@tj-fintech.com

[3] Tongji University, Shanghai, China
{xiaofengma,1631665}@tongji.edu.cn

Abstract. Blockchain is an emerging technology with broad applications. As an important application of the blockchain, smart contracts can formulate trading rules to manage thousands of virtual currencies. Nowadays, the IoT (Internet of Things) combined with blockchain has become a new trend and smart contract can implement different transaction demands for IoT-blockchain systems. Once there exits vulnerability in the smart contract program, the security of the virtual currency will not be guaranteed. However, ensuring the security of smart contracts is never an easy task. On the one hand, existing smart contracts cannot identify fake users or malicious programs, which is difficult to be regulated at present; on the other hand, smart contracts involving in multiple trading users are very similar to shared-memory concurrent programs. To deal with these problems, this study uses formal verification methods, adopting the Communicating Sequence Processes (CSP) theory to formally model concurrent programs. Then the FDR (Failure Divergence Refinement), a refinement checker or model checker for CSP, is utilized to successfully detect the vulnerability regarding concurrency in one smart contract public in Ethereum. The results show the potential advantage of using CSP and FDR tool to check the vulnerability in smart contracts especially from the perspective of concurrency.

Keywords: Blockchain · Smart contracts · Concurrency
CSP theory · FDR

1 Introduction

The security of smart contracts is a prerequisite for ensuring the normal operation of the blockchain system. However, this characterization may not live up to people's expectation due to the frequent reveal of vulnerabilities in smart contracts recently. The reasons behind these incidents are various, but there is one perspective that people tend to overlook, which is the analogy between smart contracts and conventional concurrency programs [1]. And adversaries often utilize this kind of vulnerability to obtain virtual currencies, thus causing huge losses. One prominent example is the DAO contract [1], which suffered the loss of 60 million dollars due to its vulnerability in a concurrency environment. Therefore, it is significantly important to analyze the logic design of smart contracts.

In order to detect the vulnerabilities of smart contracts, many formal verification methods are proposed. Several verification tools such as Why3 [2], F* [3] and Oyente [4] are developed to detect problems in programming languages like array overflow, poor handling of return value and etc. However, these tools rarely test the design logic of the smart contract itself [5]. If the programs have concurrency features, the vulnerabilities are most likely caused by the unreasonable design logic itself.

Given the above analysis, this study will focus on concurrency-related vulnerabilities in smart contracts and use the formal method to check it. In order to model the smart contracts in a formal way, we adopt Communicating Sequential Processes (CSP) theory. Proposed by Hoare as an algebraic language, it specially describes the interactions of entities in a concurrent system [6]. We then use FDR, a model checking tool based on CSP to do automated verification [7]. The past decades have witnessed many successful industrial applications of CSP theory, like modelling the control flow of in European train control system [8]. And FDR is well-known for its successful detection of attack in concurrent security protocol [9]. Therefore, the scientific nature of CSP theory and the convenience of FDR can provide strong support for the detection of concurrent vulnerabilities in smart contracts.

In the following sections, Part 1 will introduce basics of CSP theory used for modelling and present the workflow of formal verification of smart contract. One classic smart contract will be analyzed in Part 2. Part 3 serves as discussion. At last, Part 4 concludes this study.

Preliminaries

1.1 Basics of CSP Theory

This part introduces some basics of CSP that will be used in this study. The details can be referred to [6].

Process. Process is the abstraction of a series of behaviors of an object. The behaviors of the object can be illustrated by a finite set of events called *alphabet*. In this study, the name of a process begins with a capital and the name of an event begins with a lowercase letter. For example, the following formula describes the behaviors of a ticket vending machine.

$$\alpha TVM = \{in1d, single\}$$

The ticket vending machine is a process and can be represented as TVM. αTVM means the alphabet. It includes 2 events, which are *in1d* (input 1 dollar), *single* (buy a single ticket). Among all the processes, *STOP* is the simplest one which means "doing nothing at all".

Prefix. Process $(x \rightarrow P)$ is called prefix. It first executes event x and follows the instructions of process P for subsequent actions. Combining process and prefix will derive the basic model of interactions between processes. For example,

$$Bob\_morning \ = getup \ \rightarrow \ shower \ \rightarrow \ breakfast \ \rightarrow \ Work$$

$$Work \ = \ programming \ \rightarrow lunch \rightarrow \ STOP$$

In this example, process *Bob_morning* characterizes the actions of Bob in the morning, which begins with getting up. After taking a shower, he will have breakfast. This event is followed by another process named *Work*. In the *Work* process, he programs. Then he has lunch, which finally ends the morning with *STOP*.

Input and Output

$$c?x : T \rightarrow P(x)$$

$$c!y : T \ \rightarrow P(y)$$

As a whole, the two formulae are prefix notations. Both *c?x:T* and *c!y:T* are events and c is a channel like a buffer. *c?x:T* means x is chosen from event set T by the environment and then is put into a channel c. *c!y:T* means y is chosen from event set T by the environment and then is pushed out of a channel *c.P(x)* and *P(y)* are processes which contain event x and y, respectively.

Traces and Interleaving. The trace is a finite sequence of events that processes execute until a certain moment. $\langle x, y\rangle$ is a trace with event y happening closely after event x. The concept of traces is of great significance to this study. The main purpose of using CSP to model smart contracts is to generate a set of traces that includes possible behaviors of smart contracts under any condition. The attacker model is also a trace which depicts the sequence of malicious events under specific circumstances. If the attack sequence exists in the trace set mentioned above, the smart contract is vulnerable. And this vulnerability may be exploited by the attacker to become a real attack point to steal digital assets in the contract. If the attack sequence does not exist in the trace set, it means that the contract is able to defend this kind of attack.

If we take the smart contract as a whole process, then it can be divided into several child processes. The trace set of the smart contract is composed of all possible behaviors generated by interleaving child processes. In a concurrent circumstance, the interleaving among process X, Y can be formally defined as $X|||Y$. And their trace set can be described as

$$traces(X|||Y) = \{t|\exists u \in traces(X), \ \exists v \in traces(Y), \ t \ interleaves(u, v)\}\}$$

In this formula, t *interleaves(u, v)* means trace alternatively extracts the sequences of u and v while preserves their partial order. For example, if

$$u = \langle a, b, c, d, e \rangle \in traces(X),$$

$$v = \langle f, a, g \rangle \in traces(Y)$$

then one element of $traces\,(X|||Y)$ can be described as

$$\langle a, b, f, a, c, g, d, e \rangle \in traces\,(X|||Y)\,.$$

1.2 Workflow of Verification

In this study, the procedure for formal verification of smart contracts can be generally divided into four steps as shown in Fig. 1. Part III will illustrate these steps in detail.

(1) Analyze the given smart contract from the code level;
(2) Model the contract program by translating its codes into formal language with CSP theory;
(3) Design attacker model;
(4) Convert the modelling results into the FDR-supported CSP_M language and use FDR for verification.

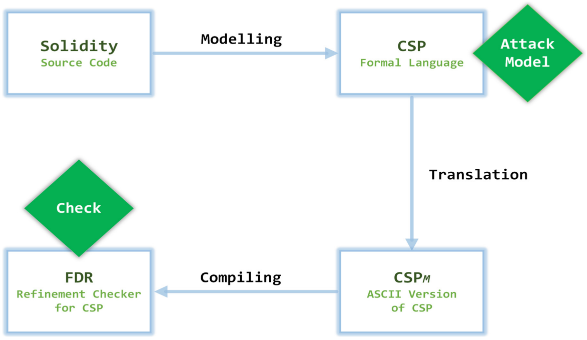


Fig. 1. Workflow of formal verification in this study.

2 The Safe Remote Purchase

This part will demonstrate how to use CSP and FDR to check the vulnerability of one smart contract case public in Ethereum. The programming language for it is Solidity.

2.1 Code Analysis

Published on [10], the Safe Remote Purchase smart contract contains 99 lines of source codes and aims to support safe and reliable transaction operation of untrusted buyers and sellers on the decentralized e-commerce platform.

For convenience to analyze, in this smart contract we assume that there are only three accounts including the buyer, the seller and the contract where the contract is the third party.

We firstly represent *value* as the price of the item involved in a single order. Then the contract is expected to work as follows. In the constructor function served for initialization, the account who sends the required ethers (2*value as the guaranty) to the contract is considered as the seller. If the constructor function executes successfully, the state of the order becomes *created*. Then the seller waits for the purchase confirmation from buyer. Before the buyer confirms to purchase, the seller has the opportunity to call the **abort()** function to abort the order, then the guaranty will return to the seller's account after the state of the order becomes *inactive*.

The account who invokes the **confirmPurchase()** function is the buyer. As stated in **confirmPurchase()** function, only if the buyer successfully sends the deposit (2*value) to the contract can the state of the order become *locked*, which finishes the confirmation procedure. The transaction will proceed smoothly until the buyer receives the item.

At last, the buyer calls the **confirmReceived()** function. In this function, the state of the order becomes *inactive*, then 1*value ethers are transferred back to the buyer account, and finally the remaining 3*value ethers in the contract account are fully returned to the seller account.

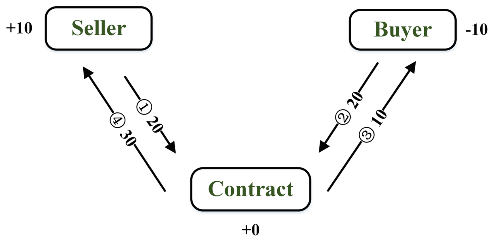


Fig. 2. Flowchart of normal transaction (value = 10 ethers).

Figure 2 shows the flowchart of normal transaction described above. From the perspective of sequential programs, the above content is conformed to the expected design purpose of Safe Remote Purchase contract. However, for a concurrent program, the problems of data race may bring about undesired behaviors that can be exploited by adversaries, which is exactly the motivation of this study. As shown in Fig. 3, the vulnerability we have discovered lies in the line order of line 72, which sends Ethers to the contract account, and line 77, which changes the order state into *locked*. In the course of the executed line 72 and non-executed line 77, the order state is still created. Hence, in this specified period, the seller is allowed to call the **abort()** function. Once **abort()** is invoked, the state of the order becomes aborted and the total balance of the contract's account, containing the guaranty from the seller and the deposit from the buyer, will be transferred to the seller's account.

```
13  constructor() public payable {
14      seller = msg.sender;
15      value = msg.value / 2;
16      require((2 * value) == msg.value, "Value has to be even.");
17  }
    ......
55  function abort()
56      public
57      onlySeller
58      inState(State.Created)
59  {
60      emit Aborted();
61      state = State.Inactive;
62      seller.transfer(address(this).balance);
63  }
    ......
66  /// Transaction has to include `2 * value` ether.
67  /// The ether will be locked until confirmReceived
68  /// is called.
69  function confirmPurchase()
70      public
71      inState(State.Created)
72      condition(msg.value == (2 * value))
73      payable
74  {
75      emit PurchaseConfirmed();
76      buyer = msg.sender;
77      state = State.Locked;
78  }
    ......
82  function confirmReceived()
83      public
84      onlyBuyer
85      inState(State.Locked)
86  {
87      emit ItemReceived();
    ......
91      state = State.Inactive;
    ......
96      buyer.transfer(value);
97      seller.transfer(address(this).balance);
98  }
```

Fig. 3. Safe remote purchase code fragments.

2.2 CSP Modelling on the Safe Remote Purchase

This section mainly discusses how CSP is employed to model the Safe Remote Purchase smart contract. CSP can model systems that is decomposed into multiple concurrent sub-components. In this study, we model the smart contract mainly by defining CSP processes corresponding to public-type functions in the program. And all these processes constitute a system with interleaving mode. CSP can also describe the attack sequence which is termed as the undesired trace hidden in the program. Then, FDR is used to test whether the attack sequence exits theoretically. Considering the properties of smart contract, the modelling process will focus on formal definition for five types of objects: variable, event, process, concurrent model and attacker model.

Variable. Variables in the Safe Remote Purchase are split into two types: global variables (GVs) and key variables. Analogous to the shared variable in conventional concurrent programs, multiple accounts may read or write the GVs in smart contracts in an inappropriate way, which potentially causes the concurrent vulnerabilities. Therefore, precise modelling on GVs is virtually required. In the case of Safe Remote Purchase, the global variable is the state of the order and we define it as

$$State := \{created, locked, inactive\}$$

State is a set containing three elements representing three different states of the order. We use *state* (beginning with a lowercase letter) to represent any element in *State*, namely $state\ := created \mid locked \mid inactive \in State.$ In this formula, $''|''$ $meansselection.$

The key variable is also abstracted from the contract. Here we list two groups of variables that are often used in this contract. The first group is

$$Object := \{seller, buyer, eth\}$$

In this group, *seller*,*buyer* and *eth* mean the account of the seller, buyer and the smart contract, respectively. And we also have $object\ := seller \mid buyer \mid eth \in Object.$

The second group represents the extra information with the behavior of money transfers and it is described as

$$MoneyInfo := \{guaranty, deposit, balance, trans\}$$

In this group, *guaranty* indicates the money transferred is paid as guaranty; *deposit* means the money transferred is used to pay the deposit; *balance* indicates that all the balance of a particular account is transferred to other accounts; and *trans* means ordinary money transfer behavior without any special information. Still, we can define $moneyInfo\ := guaranty \mid deposit \mid balance|trans \in MoneyInfo.$

Event. The part of actions in the contract can generally be abstracted into events such as initialization of the constructor function, money transfer, changing the value of certain variables and so on. In addition, the judging course of the certain condition in the program can also be considered as events although there may not be any real actions occurring. Since there can be more than one scenarios involved in an event, a set is generally used to describe a type of events. To avoid name confusion, the definition of event names generally ends with "msg".

The following shows the event in which the seller initializes the constructor function:

$$INITmsg \wedge = \{init.Int\}$$

This event contains only one element which only represents the initialization action semantically. "." indicates the information carried with the event. If there are more than one types of information, this symbol can be used repeatedly. *Int* is an integer by default. Therefore, *init.Int* can be specified as the event in which the seller initializes the transaction and an order valued *Int* ethers gets generated.

Then we define the money transfer event shown as below:

$$\begin{aligned} MTmsg \wedge = \{\ object.moneyInfo.object.Int|\ & object \in Object, \\ & moneyInfo \in MoneyInfo\} \end{aligned}$$

This formula describes a series of money transfer events in which *object. moneyInfo.object:Int* represents that one object transfers a certain amount of ethers to another object in a certain way. If *moneyInfo=balance*, then *Int* is automatically marked as all balances of the first *object* and *Int* is denoted by "*overall*". The following three types of events model the conditional statements in this contract.

$$RImsg \wedge = \{access?object \mid object \in Object\}$$

$$RSmsg \wedge = \{state' \; ? \; state \mid state \in State\}$$

$$RDmsg \wedge = \{buyer \; ? \; Int\}$$

In order to verify the given condition, the first formula depicts events in which the identity of the function caller is input into channel *access*. The second one means the current state of the order is input into channel *state'*. The last one represents that the deposit paid by the buyer is put into channel *buyer*. The formula below defines the event in which the state of the order gets changed.

$$WSmsg \wedge = \{state' \; !state \mid state \in State\}$$

In this event, the renewed state gets written and is pushed out of channel *state'*. There are some statements in smart contracts that indicate some information is sent to all participants during transaction. In Solidity, such information is modified with the keyword *emit*. When modelling, we only assign a name to these events. In the case contract, these events are

$$abort, \; purchaseConfirmed, \; itemReceived, \; warning.$$

Process. We model four functions (shown in Fig. 3), namely **constructor()**, **abort()**, **confirmPurchase()** and **itemReceived()** into four processes - *INIT*, *ABORT(x)*, *ConfirmPurchased(x)* and *itemReceived(x)*, respectively.

$$\begin{aligned}\textbf{INIT} = \; & init?msg\_value : Int \rightarrow \textbf{if}\,(msg\_value \; \% \; 2 == 0)\\ & \textbf{then}(seller.guaranty.eth.msg\_value \rightarrow state'!state = created \rightarrow\\ & PURCHASE\,(state)\,\textbf{else}(warning \rightarrow INIT)\end{aligned}$$

$$\begin{aligned}\textbf{ABORT}(x) = \; & access?object : Object \rightarrow \textbf{if}\,(object \;==\; seller)\\ & \textbf{then}(state'?x \rightarrow \textbf{if}\,(x \;==\; created)\,\textbf{then}(abort \rightarrow state!inactive \rightarrow\\ & \quad eth.balance.seller.overall \rightarrow STOP)\\ & \qquad \textbf{else}(warning \rightarrow STOP))\\ & \qquad ` \; \textbf{else}(warning \rightarrow STOP)\end{aligned}$$

$$\begin{aligned}\textbf{ConfirmPurchased}\,(x) = \; & access?object : Object \rightarrow state'?x \rightarrow \textbf{if}\;(\; x \;==\; created\;)\\ & \textbf{then}(buyer?value : Int \rightarrow \textbf{if}\,(value \;==\; msg\_value)\\ & \quad \textbf{then}(purchaseConfirmed \;\rightarrow state'!locked \rightarrow STOP)\\ & \quad \textbf{else}(warning \rightarrow STOP))\\ & \textbf{else}(warning \rightarrow STOP)\end{aligned}$$

$$\begin{aligned}\mathbf{ItemReceived}\,(x) = \; & access?object \;:\; Object \rightarrow \mathbf{if}\,(object \;==\; buyer)\\ & \mathbf{then}(state'?x \rightarrow \mathbf{if}\,(x \;==\; locked)\,\mathbf{then}(itemReceived\\ & \qquad \rightarrow state'!inactive \rightarrow eth.trans.buyer.\,(msg\_value/2)\\ & \qquad\quad \rightarrow eth.balance.seller.overall \rightarrow STOP)\\ & \quad \mathbf{else}\,(warning \;!\; STOP))\\ & \mathbf{else}\,(warning \;!\; STOP)\end{aligned}$$

Concurrent Model. As we have detailed before, except for the *INIT* process, it is not hard to see the order of occurrence of *ABORT(x)*, *ConfirmPurchased(x)* and *ItemReceived(x)* is non-deterministic. Owing to the existence of GV, the proceeding of each process is largely related to the changing value of the element of GV, *state*. Therefore, we use interleaving as the concurrent mode to model all possible sequences happening in theory. The parameter x represents any element of the GV and will be synchronously transmitted to each process. Then, the PURCHASE(x) is defined as

$$\mathbf{PURCHASE(x)} \;= (ABORT(x)|||ConfirmPurchased(x)|||ConfirmReceived(x))\,.$$

$$\mathbf{traces(PURCHASE(x))}$$
$$= \left\{\begin{aligned} & s \mid \exists t \in traces\big(ABORT(x)\big),\\ & \quad \exists u \in traces\big(ConfirmPurchased(x)\big),\\ & \quad \exists w \in traces(ConfirmReceived(x)),\\ & \qquad\qquad\qquad s\ interleaves(t,u,w)\end{aligned}\right\}$$

The definition of *traces(PURCHASE(x))* shows that PURCHASE(x) can engage in any possible sequences when the trace of the specific three processes gets interleaved.

As for *traces(SYS)*, it represents a set including all possible traces as well as state transitions for the Safe Remote Purchase contract, which essentially equals to *trace(INIT).*

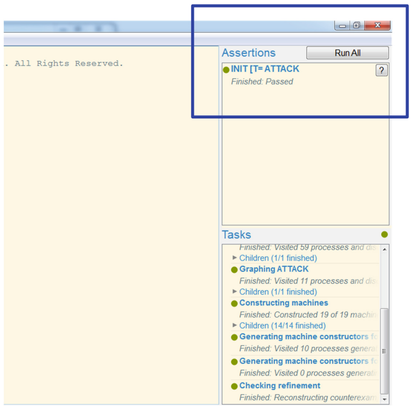
$$traces(SYS) = \{s\,|s \in traces(INIT)\,\}$$

Attacker Model. Attacker model is a trace where the vulnerability resides in. And formal definition of this model is shown as follows.

$$attack = \left\langle\begin{aligned} & init.msg\_value,\ seller.guarantee.eth.msg\_value,\ state'!created,\\ & access?buyer,\ state'?created,\ buyer?msg\_value,\ access?seller,\\ & state'!?created,\ abort,\ state'!inactive,\ eth.balance.seller.overall,\\ & purchaseConfirmed,\ state'!locked,\ STOP\end{aligned}\right\rangle$$

2.3 Using FDR to Verify the Contract

The above formal language is translated into CSP_M language for automated verification by FDR. As is shown in the blue rectangular box of Fig. 4, "*Passed*" means the *attack* trace exists in the *traces(SYS).*



```
. All Rights Reserved.
```

Assertions

Run All

● INIT [T= ATTACK
Finished: Passed

Tasks

Finished: Visited 59 processes and dis
▸ Children (1/1 finished)
● Graphing ATTACK
Finished: Visited 11 processes and dis
▸ Children (1/1 finished)
● Constructing machines
Finished: Constructed 19 of 19 machin
▸ Children (14/14 finished)
● Generating machine constructors fo
Finished: Visited 10 processes generat
● Generating machine constructors fo
Finished: Visited 0 processes generati
● Checking refinement
Finished: Reconstructing counterexam

Fig. 4. Screenshot for verification result. (Color figure online)

3 Discussion

From the analysis and verification in part 2, theoretically we can make sure that there is vulnerability in the Safe Remote Purchase smart contract. The vulnerabilities exposed in the former case are very similar to racing problems in traditional concurrent programs. In this case, both **abort()** and **confirm-Purchase()** have the ability to execute writing operation to the global variable named state. The value of this global variable gets tampered by calling **abort()** while **confirmPurchase()** is under execution. Therefore, this kind of concurrent vulnerability can be grouped into the *Tamper by Interference* problems. Regarding this, these problems are probably ubiquitous in smart contracts especially in application of e-commerce. How to efficiently check and repair them is considered to be a future direction in the verification field.

Another point that will be discussed here is the seemingly atomic property of smart contract functions. Many people hold the view that functions in smart contracts will not be interrupted during execution. They believe that the consensus mechanism of the blockchain will guarantee this property. However, it seems logically insufficient for programming languages to rely on third parties like consensus mechanism to avoid the responsibilities of the language itself. And to the best of our knowledge so far, there has not been any evidence in literature to prove this atomic property can be ensured only through consensus mechanism. Hence, it should be admitted that although programming languages like Solidity is flexible to write smart contracts, their support for concurrency is not perfect.

This study only provides a framework of using CSP and FDR to check the vulnerability of smart contracts. And the readers may get puzzled that in the case study part, the vulnerability is known ahead of modelling through CSP. However, if we verify other unfamiliar smart contracts, the vulnerability may not be revealed with ease. And if the program is very complicated itself, it is nearly impossible to find its vulnerability from the code level. In this regard, it is still a good approach to use CSP and FDR to do formal verification but refinements should be accomplished.

In this study, the smart contract program is manually translated into CSP language, which is both time-consuming and easy to make mistakes especially when the scale of programs become large. Therefore, a compiler is needed in order to realize automatic or semi-automatic translation from smart contract programming languages to CSP language.

Through the endeavors mentioned above, we will be able to efficiently verify many smart contracts and accumulate the experience of designing attacker models. Then we can sum up a set of methods to detect several types of concurrent vulnerabilities, which can be avoided by theoretical proof when designing new contracts.

4 Conclusion

In this paper, we analyze the vulnerability of smart contracts especially from the perspective of concurrency. The analogy between smart contracts and concurrency programs is vividly revealed with the illustration of one example contract, in which possible attacks can be implemented by virtue of operating shared variables. Then CSP theory and FDR model checking tool are employed to detect the existing vulnerability. As shown, the example contract and its attack sequences can be modelled in a formal way by CSP theory with the interleaving of processes constituting the state space. Finally, the automated checking of FDR manifests that the attack sequence does exist in the state space, which proves the efficiency of CSP and FDR to check the vulnerability of smart contracts regarding concurrency.

Acknowledgment. This work was supported by the XJTLU research development fund projects under Grant RDF140243 and Grant RDF150246, in part by the National Natural Science Foundation of China under Grant No. 61701418, in part by Innovation Projects of The Next Generation Internet Technology under Grant NGII20170301, in part by the Suzhou Science and Technology Development Plan under Grant SYG201516, and in part by the Jiangsu Province National Science Foundation under Grant BK20150376.

References

1. Sergey, I., Hobor, A.: A concurrent perspective on smart contracts. In: Brenner, M., et al. (eds.) FC 2017. LNCS, vol. 10323, pp. 478–493. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-70278-0_30
2. Filliâtre, J.-C., Paskevich, A.: Why3—where programs meet provers. In: Felleisen, M., Gardner, P. (eds.) ESOP 2013. LNCS, vol. 7792, pp. 125–128. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-37036-6_8
3. Swamy, N., Hriţcu, C., Keller, C., et al.: Dependent types and multi-monadic effects in F. ACM SIGPLAN Notices **51**(1), 256–270 (2016)

4. Luu, L., Chu, D.H., Olickel, H., et al.: Making smart contracts smarter. In: Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, pp. 254–269. ACM (2016)
5. Atzei, N., Bartoletti, M., Cimoli, T.: A survey of attacks on ethereum smart contracts (SoK). In: Maffei, M., Ryan, M. (eds.) POST 2017. LNCS, vol. 10204, pp. 164–186. Springer, Heidelberg (2017). https://doi.org/10.1007/978-3-662-54455-6_8
6. Hoare, C.A.R.: Communicating sequential processes. Commun. ACM **21**(8), 666–677 (1978)
7. Roscoe, A.W.: Understanding Concurrent Systems. Springer, London (2010). https://doi.org/10.1007/978-1-84882-258-0
8. Faber, J., Jacobs, S., Sofronie-Stokkermans, V.: Verifying CSP-OZ-DC specifications with complex data types and timing parameters. In: Davies, J., Gibbons, J. (eds.) IFM 2007. LNCS, vol. 4591, pp. 233–252. Springer, Heidelberg (2007). https://doi.org/10.1007/978-3-540-73210-5_13
9. Lowe, G.: Breaking and fixing the needham-schroeder public-key protocol using FDR. In: Margaria, T., Steffen, B. (eds.) TACAS 1996. LNCS, vol. 1055, pp. 147–166. Springer, Heidelberg (1996). https://doi.org/10.1007/3-540-61042-1_43
10. Safe Remote Purchase contract. https://solidity.readthedocs.io/en/v0.4.24/solidity-by-example.htmlsafe-remote-purchase