Model Checking Concurrency in Smart Contracts with a Case Study of Safe Remote Purchase

Yisong Yu[1,2], Naipeng Dong[3], Zhe Hou[4], and Jin Song Dong[2]

[1] Ningbo University, Ningbo City, Zhejiang Province 315211, China
[2] National University of Singapore, 21 Lower Kent Ridge Road 119077, Singapore
[3] The University of Queensland, St Lucia QLD 4072, Australia
[4] Griffith University, 170 Kessels Rd, Nathan QLD 4111, Australia
yisongyu98@gmail.com,n.dong@uq.edu.au,
z.hou@griffith.edu.au,dcsdjs@nus.edu.sg

Abstract. Blockchain technology has evolved beyond its initial role in supporting cryptocurrencies like Bitcoin, with Ethereum introducing smart contracts for decentralised applications in various domains. However, ensuring the safety and security of smart contracts remains a critical challenge, particularly concerning concurrency issues. This is of paramount importance because the smart contract ecosystem is concurrent by nature as its underlying blockchain is decentralised, and the concurrency-related vulnerabilities within smart contracts have resulted in substantial financial losses. We observe that in the literature, concurrency is handled with two strong assumptions, leading to either undetected attacks or false alarms. Taking the Safe Remote Purchase smart contract as a case study, we investigated the root causes and introduced a novel method that incorporates blockchain-specific characteristics into the verification process. Our contributions include a formal framework, an automated model generator, and a compelling case study that illustrates a reduction in false attacks, thus advancing the state of smart contract security in blockchain ecosystems. The formal models and the framework generator are available online at `https://github.com/FormalVerificationBlockchain/Concurrency`.

Keywords: Smart Contracts · Concurrency · Formal Verification · Blockchain.

1 Introduction

Blockchain was introduced as the fundamental technology for supporting the well-known cryptocurrency — Bitcoin in 2008. Later, Ethereum [26], known as Blockchain 2.0, aims to provide an open and decentralised platform for general-purpose computing through the introduction of a groundbreaking concept called *smart contract*. It enables the development of decentralised applications across various domains, from finance to supply chain management.

Smart contracts, which are Quasi-Turing-Complete programs running on top of a blockchain, have a unique feature that, once deployed, it is costly to change them (*i.e.*, patching is nearly infeasible) due to the irreversible feature of the underlying blockchain that stores the programs. Therefore, verifying the correctness of smart contracts before deploying them to a blockchain is crucial. Time and again, incidents have led to huge financial losses due to bugs, breaches and logic flaws in smart contracts — *e.g.*, the well-known DAO attack [18], Parity Multisig Wallet attack [8], and the King of the Ether Throne attack [1]. A variety of techniques have been developed to verify the safety and security of smart contracts and their applications in the past several years, including design patterns [27], informal vulnerability detection [3] and formal verification approaches [24]. Formal verification stands out because many smart contract applications are safety-critical *e.g.*, supply chain, finance, and medical services, and formal verification provides rigorous proof contrary to other approaches.

However, most of the formal verification techniques take smart contracts independently, isolating them from other participants, and consider their semantics as sequential programs, as pointed out by Sergey and Hobor [17]. Such methods overlook complex interactions and can lead to security risks. Luu et al. [13] first identified security issues caused by the network participants, particularly pointing out that the transaction orders are non-deterministic, which can be utilised by malicious users to gain benefits, known as the Transaction-Ordering Dependence (TOD) problem. A subsequent study [17] generalised the problem to be the concurrency issues and introduced two additional types of concurrency scenarios: *Multitasking* and *Multi-Transactional Tasks*, by drawing analogies to the concurrent objects in shared memory in the well-studied concurrency field named as *concurrent-objects-as-contracts*.

To address these concurrency issues, Luu et al. [13] developed a tool, OYENTE, to detect TOD in smart contracts based on symbolic execution. Although rigorous, OYENTE still suffers from a large number of both falsely reported TOD and missed TOD cases because its execution environment of Ethereum is not fully simulated [15]. This observation was also noted by Sergey and Hobor [17], highlighting that verification of smart contracts requires modelling the interactions with other components. While Sergey and Hobor [17] demonstrated the concurrency problems using example smart contracts and their vulnerabilities, they did not delve into verification approaches. Building upon this approach — concurrent-objects-as-contracts, a later work [16] took the first step towards formalising concurrency. Taking the Safe Remote Purchase smart contract as a case study, the work [16] studied its source code, modelled the smart contract using the Communicating Sequence Processes (CSP), proposed an attack model, and verified the existence of a concurrency attack using the model checker FDR.

Upon examining the model and verification in [16], we identified two limitations: 1) The attack is not automatically detected. The attack model represented as a trace embedding of the identified vulnerability, requires the analysts to have prior knowledge of vulnerability exploration and analyse it manually. 2) More significantly, naively adopting the concurrent-objects-as-contracts approach results in *false attacks*. We show that the attack reported in [16] is, in fact, a false alarm in the context of the blockchain ecosystem.

The first limitation can be straightforwardly addressed by modifying the query, given that most model checkers are capable of automatically detecting concurrency issues, as shown in Section 4. The second limitation stems from a misunderstanding of the blockchain foundation, which is more challenging. The blockchain foundation cannot be treated as a single machine, as demonstrated in the above works; otherwise, there will be concurrency vulnerabilities. On the other hand, the transactions are not executed in a completely interleaving manner; otherwise, there will be false attacks. To strike a balance between these two ends of the spectrum of modelling approaches, we show that including a blockchain component in the modelling, even an abstracted one, would address the limitation. Moreover, we developed a general method by proposing a verification framework that facilitates verification beyond the case study.

The contributions of this work can be summarised as follows:

- We identified false attacks within existing approaches to verify the concurrency of smart contracts and have investigated their root causes, namely the absence of a correct blockchain execution model.
- We proposed an approach that addresses the identified issue by modelling the environment participants, including the blockchain and the external actors.
- We generalised the approach and developed a script to automatically generate formal models that facilitate the verification of smart contracts.

2 Concurrency Issues

EVM is a *single-threaded* state machine which cannot process instructions in parallel (except parallel EVM *e.g.*, [25]), suggesting that smart contract methods can be deemed, in a traditional programming context, as *sequential programs*. This is deceptive and misleading, given concurrent behaviours *e.g.*, reentrancy and recursive calls, can still be observed, if investigating it from a different perspective — at the level of blockchain ecosystem. More specifically, the following complications have been observed and demonstrated in the literature:

Transaction-Ordering Dependence. Although each transaction is always deterministic, *non-determinism* may still arise from races between transactions *i.e.*, out-of-order executions from the perspective of an external actor, leading to distinct outcomes. For instance, malicious external users could front run a transaction by providing higher fee (front-running attack), and malicious miners could reorder transactions to gain profits (Miner Extractable Value attack) [10].

Multitasking. Calling other contracts or oracles, dictates an explicit "yield" or "relinquishment" of control that will not be handed back to the caller until the callee contract returns. During this time, several things can go wrong: The callee contract can run whatever code it likes or even call other external contracts to engage in unexpected activities without getting interrupted. The caller may be malicious *e.g.*, in the DAO attack the caller re-enters the smart contract to draw extra fund. The callee's input arguments and return values are passed using volatile memory space that may be compromised.

Multi-Transactional Tasks. The contract logic of some programming tasks, which should conceptually be grouped into a single logical transactional entity, sometimes needs to be separated across multiple physical blockchain transactions. Between these transactions, other transactions whose method invocation may involve manipulation of the state unanticipated by the original proposing actor, can take place, resulting in true concurrent behaviours.

```
pragma solidity ^0.8.4;
contract Counter {
   address public id; uint private count;
   constructor() payable {
      id = msg.sender; count = msg.value; } }
function get() returns (uint) { return count; }
function set() returns (uint) {
      uint oldCount = count; count = msg.value;
      msg.sender.transfer(oldCount);
      return oldCount; }
```

Fig. 1: Example Smart Contract Code with Concurrency Issues.

Example 1 (Smart Contract with Multi-Transactional Issues). The smart contract in Figure 1 has three functions: a **constructor** function that imitates the smart contract by assigning values to *id* and *count*, a **get** function that reads the value of *count*, and a **set** function that updates the *count* value and transfers fund worth the previous *count* value (*i.e.*, *oldCount*) to the caller.

Imagine the following scenario: Alice wants to update the value *count* by calling the function **set**, and before doing so, she first calls the function **get** to check how much fund she would receive *i.e.*, the current value of *count*. Even the order of Alice's transactions (**get** followed by **set**) is correctly preserved, other problem arises, as there is no guarantee that other function invocation requests won't be scheduled in between Alice's two transactions. For example, Bob may have called the **set** in between, which resets the *count* value. In this case, Alice would get a different amount of fund (Bob's updated value of *count*) and Bob's set value of *count* will be re-written by Alice's update.

Multi-transactional behaviours are the problem that the concurrent-objects-as-contracts approach claims to be able to address. Note that the above concurrency issues in Example 1 is a true attack, which can be confirmed by both the concurrent-objects-as-contracts approach and our formal verification framework (detailed in Section 5.2). In contrast, in the subsequent section, we show another smart contract as our running example to demonstrate false attacks occur if using the concurrent-objects-as-contracts approach naively.

3 The Safe Remote Purchase Smart Contract

We analyse the same smart contract — the Safe Remote Purchase — as in [16] that uses concurrent-objects-as-contracts approach.

3.1 The Case Study Smart Contract

The smart contract aims to help mutually distrusting parties, for example, a seller and a buyer in the simplest configuration, to achieve a safe and reliable purchase transaction on a decentralised e-commerce platform built on top of Ethereum, by resolving the issue with the confirmation of the shipment and the

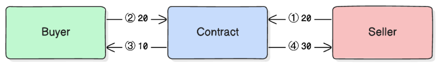

Fig. 2: Flowchart of Safe Remote Purchase Transaction [16].

```
1   contract Purchase {
2      uint public value;
3      address payable public seller, buyer;
4      enum State { Created, Locked, Release,
             Inactive }
5      error OnlyBuyer(), OnlySeller(), InvalidState
             (), ValueNotEven();
6      modifier condition(bool condition_) {
7         require(condition_); _;}
8      modifier onlyBuyer() {
9         if (msg.sender != buyer)
10           revert OnlyBuyer(); _;}
11     modifier onlySeller() {
12        if (msg.sender != seller)
13           revert OnlySeller(); _;}
14     modifier inState(State state_) {
15        if (state != state_)
16           revert InvalidState(); _;}
17
18     event Aborted(), PurchaseConfirmed(),
             ItemReceived();
19     constructor() payable {
20        seller = payable(msg.sender);
21        value = msg.value / 2;
22        if ((2 * value) != msg.value)
23           revert ValueNotEven(); }
24     function abort()
25        external onlySeller inState(State.Created) {
26        emit Aborted(); state = State.Inactive;
27        seller.transfer(address(this).balance); }
28     function confirmPurchased()
29        external inState(State.Created)
30        condition(msg.value == (2 * value))
31        payable { emit PurchaseConfirmed();
32        buyer = payable(msg.sender);
33        state = State.Locked; }
34     function confirmReceived()
35        external onlyBuyer inState(State.Locked) {
36        emit ItemReceived();
37        state = State.Release;
38        buyer.transfer(value);
39        seller.transfer(address(this).balance); } }
```

Fig. 3: Solidity Code of Safe Remote Purchase [19].

acknowledgement of the confirmation prior to settlement (Figure 2). The general strategy proposed by this contract is to enforce a guaranty/deposit of twice the value of the item to be transferred into the contract account as escrow on both parties of a remote purchase transaction. The deposit stays locked until the buyer confirms the receipt of the goods, triggering a refund of the appropriate amount to both parties, *i.e.*, the value equivalent to half of the deposit to the buyer and three times the value equivalent to the guaranty plus the value to the seller.

The solidity code of the smart contract is shown in Figure 3. The constructor function (lines 19-23) is called by the seller to create a contract and pay the guaranty. Then, the buyer can initiate a purchase by calling the function confirmPurchased with the payment of a deposit as a parameter ("msg.value"). Then, the smart contract locks the payment (line 33). After the seller delivers the purchased item, the buyer invokes the function confirmReceived to notify the contract that the item has been received, where the smart contract refunds the seller and buyer *i.e.*, releasing the lock and paying the corresponding funds to the seller and buyer (line 37-39). In this process, the seller is allowed to abort the contract by calling the function abort. The abort function sends the balance of the contract to the seller (line 27).

3.2 Formal Analysis of Concurrency in Existing Work

Wang et al. [16] showed an attack trace when performing model checking using the concurrent-objects-as-contracts approach, *i.e.*, analogous concurrency in smart contracts to racing problems in traditional concurrent programs. The vulnerability discovered lies in-between the execution of line 28 in Figure 3, which sends deposit to the contract account, and line 33, which locks the payment. From a concurrency perspective, in this specified period, the seller is allowed to call the `abort` function, which changes the state of the purchase to be aborted and the total balance of the contract's account, containing the guaranty/deposit of both seller and buyer, will be transferred to the seller's account.

Critiques. We identified two limitations of the above formal modelling and verification: First, differing from traditional model checking approaches, this model came up with an attacker trace in advance as the input to the FDR model checker to only confirm the existence of such a counterexample trace in all possible executions of the modelled smart contract. The attack trace is an artefact usually not available beforehand, as no one would be able to know what kind of attacks can be launched against some vulnerabilities that may or may not even exist in his/her built contract. Second, this model lacks environmental considerations, such as expected and unexpected user interactions, *e.g.*, possible behaviours of a seller/buyer through the provided interfaces (those demonstrated in Figure 2).

4 Analysing Existing Verification Approach in PAT

The first critique can be straightforwardly addressed, as the attack trace can be automatically identified by many existing model checkers. We illustrate it by mimicking the modelling and verification of [16] using the CSP# modelling language supported by the model checker PAT (Process Analysis Toolkit) [21]. The reason for choosing another model checker is that the approach used in [16] lacks support for state variables and many other common programming constructs for flow controls, which hinders our goal of developing a general framework (detailed in Section 5), while PAT enables us to develop external libraries and functions.

4.1 Introduction to CSP# and PAT

PAT is a self-contained framework that supports the composing, simulating, and reasoning of concurrent systems using various model-checking techniques. More importantly, PAT has been developed as a generic framework, which can be easily extended to support new modules and frameworks. The input language of PAT is CSP#, whose syntax is shown in Definition 1.

Definition 1 (Syntax of Processes in CSP#).

$$\begin{aligned} P,\ Q\ ::=\ & \mathit{Stop} \mid \mathit{Skip} \mid e \rightarrow P \mid e\{\mathit{prog}\} \rightarrow P \mid c!\mathit{exp} \rightarrow P \mid c?x \rightarrow P \mid \\ & P[]Q \mid if(b)\{P\}\ \mathit{else}\ \{Q\} \mid ifa(b)\{P\}\ \mathit{else}\ \{Q\} \mid P;Q \mid P||Q \mid P|||Q \end{aligned}$$

Stop and *Skip* are processes denoting inaction and termination, respectively. Process $e \rightarrow P$ engages in an atomic event e first and then behaves as process P. The event is allowed to attach an atomically executed program, denoted as $e\{\mathit{prog}\} \rightarrow P$. Channel communication is supported: $c!\mathit{exp} \rightarrow P$ denotes sending exp over channel c, while $c?x \rightarrow P$ denotes reading from channel c and referring to the message as x. Two types of choices are supported: $P[]Q$ denotes unconditional choice, and $if(b)\{P\}\ \mathit{else}\ \{Q\}$ is conditional branching, where b is a boolean expression. $ifa(b)\{P\}\ else\ \{Q\}$ is a variation of conditional branching that performs the condition checking and first operation of P/Q together. There are three types of process relations: Process $P; Q$ behaves as P until P terminates and then behaves as Q. The parallel composition of two processes is written as $P||Q$, where P and Q may communicate via multi-party event synchronisation. If P and Q only communicate through variables, then it is written as $P|||Q$.

4.2 Mimicking the Modelling of the Existing Work

To demonstrate the capability of PAT, we first faithfully replicate the model in [16] into a corresponding CSP# model with all required modifications to maximise the semantic equivalence between them. That is, we expect to see the same result as in the previous work, *which may not be correct.* Essentially, each function in Figure 3 is modelled as a process in CSP# and the smart contract is the interleaving of the functions, as shown in Figure 4. The modelling is straightforward: the model of function constructor is in line 8-12 in Figure 4; the model of abort is in line 13-19; confirmPurchased in line 21-28; and confirmReceived in line 29-33; line 34 is the composed smart contract.

```
1  enum {CONTRACT, SELLER, BUYER};
2  enum {NULL, CREATED, LOCKED, RELEASE,
         INACTIVE};
3  channel constructor 1;
4  channel buyer 1;
5  channel access 1;
6  var guaranty = -1;
7  var state = NULL;
8  Constructor() =
9    constructor?msg_value ->
10   ifa (msg_value % 2 == 0) {
11     seller_guaranty_eth_msg_value{guaranty
         = msg_value;} -> state_created{state
         = CREATED;} -> Purchase()
12   } else { warning -> Skip };
13 Abort() =
14   access?object ->
15   ifa (object == SELLER) {
16     ifa(state == CREATED) { abort ->
17       state_aborted{state = INACTIVE;} ->
          eth_balance_seller_overall -> Skip
18     } else { warning -> Skip }
19   } else { warning -> Skip };
20
```

```
21 ConfirmPurchased() = access?object ->
22   ifa (object == BUYER) { ifa (state == CREATED) {
23     buyer?deposit -> ifa (deposit == guaranty) {
24         purchase_confirmed ->
25         state_locked{state = LOCKED;} -> Skip
26       } else { warning -> Skip }
27     } else { warning -> Skip }
28   } else { warning -> Skip };
29 ConfirmReceived() = access?object ->
30   ifa (object == BUYER) { ifa (state == LOCKED) {
31       item_received -> state_inactive{state = RELEASE;} -> Skip
32     } else { warning -> Skip }
33   } else { warning -> Skip };
34 Purchase() = Abort() ||| ConfirmPurchased() ||| ConfirmReceived
         ();
35 Deployer(guaranty_value) = constructor!guaranty_value -> Skip;
36 Seller() = access!SELLER -> Skip;
37 Buyer(deposit_value) = access!BUYER -> buyer!deposit_value ->
         access!BUYER -> Skip;
38 BlockchainSystem() = Constructor() || (Deployer(10); (Seller() |
         || Buyer(10)));
39 #assert BlockchainSystem() |= [] (purchase_confirmed -> !<>
         abort);
```

Fig. 4: Replicated CSP# Model Code Snippets of Safe Remote Purchase [16].

To address the second critique (cf. Section 3.2), we integrate additional processes (the processes in line 35-38 in Figure 4) to model the behaviours of external actors — the seller (deployer) and the buyer, to make this model complete. The external actors use the channel constructs to communicate with the smart contract functions, modelling the invocation of the functions in real-world scenarios. The input operations in the channel constructs must have the corresponding output operations to be specified in order to prevent a deadlocked scenario; therefore, we correspondingly add channel receive at the beginning of each function (*i.e.*, line 9, 14, 21 and 29 respectively).

We then formalised concurrency as assertions using the supported LTL (Linear Temporal Logic). The assertion (line 39 in Figure 4) captures the interference between the confirming purchase and aborting by querying the statement — if the purchase is confirmed, then the seller would not abort.

Assertion	BlockchainSystem() |= [](purchase_confirmed -> !<>abort)
Counterexample	init -> constructor!10 -> access!BUYER -> buyer!10 -> constructor?10 -> seller_guaranty_eth_msg_value -> state_created -> access?BUYER -> access!BUYER -> warning -> access?BUYER -> access!SELLER -> buyer?10 -> **purchase_confirmed** -> access?SELLER -> **abort**

Table 1: Verification Result of the Replicated CSP# Model.

Verification Result Evaluations. Running this CSP# model in the PAT model checker produces the verification result as shown in Table 1 with a counter-example trace given as the proof of the invalidity of the assertion that states the safety property of this contract, resembling the attacker model trace presented in the original literature. Although not exactly identical, it captures the same attack trace where the event abort (line 13 in Figure 4) is preceded chronologically by the event purchase_confirmed (line 24). As a consequence, the whole smart contract balance (comprised of guaranty/deposit from both buyer and seller) is transferred from the contract address to the seller address in the function abort immediately after the deposit transfer from the buyer address to the contract address, but before the state transition (from CREATED to LOCKED in line 25) operation gets performed in the function confirmPurchased (to bypass the pre-condition check of the function abort). This reveals a potential vulnerability inherent in the original contract design, which may be exploited by a malicious seller to successfully steal the deposit of the buyer if they can somehow manipulate the execution to enforce this particular sequence of executions.

Modelling Strategy Critiques. However, the above-described attack, identified in [16] and our above verification, in reality, is *not feasible* given the underlying execution model of EVM where a transaction that encapsulates a contract function invocation is an atomic operation that cannot be divided into several chunks for executions, and thus, a contract function whose execution cannot be interleaved with other functions is either completely done or not at all, implying that a *false positive* result is found based on this wrong assumption.

4.3 An Analysis of Issues in Existing Methods

The above false positive originates from a wrong interpretation of the execution model of EVM. In reality, the function abort can never get its turn for execution until the other function confirmPurchased fully finishes in the example contract. It is an incorrect use of the interleaving operator to compose the three possible actions to be performed by an external participant, corresponding to the processes Abort, ConfirmPurchased, and ConfirmReceived in Figure 4.

In this particular case, there is an easy fix *i.e.*, replacing the interleaving of the three processes (Abort, ConfirmPurchased and ConfirmRecieved) with unconditional choices of all possible permutations of these three constituent processes sequentially composed with each other. In this way, only one single function can be executed sequentially until termination without being interrupted/preempted by other functions. However, writing this kind of long process definition spanning over more than five lines is tedious, error-prone and unsustainable. Given that there are in total $n!$ permutations for n constituent processes, hinting that it will soon become impractical to write them by hand when n becomes larger as it demands $O(n!)$ time to verify the result, making the modelling alone is an NP-hard problem already, which suffers from combinatorial explosion, before we even start tackling the state explosion issue that we are likely to encounter during the verification.

In addition, the model (in [16] and thus the same in the above model) does not handle the case of subsequent attempts of function invocations after the created purchase order has been finalised, *i.e.*, the processes are only executed once. The likelihood of the functions abort, confirmPurchased, and confirmReceived being invoked multiple times in arbitrary order due to the contract's immortality once deployed (assuming no *self-destruct* function) can render verification results flawed. Without accurately reflecting these non-terminating behaviours in the model, the state space may not cover all possible intricate interactions, potentially leading to unexpected side effects and false positive vulnerabilities. This issue can be resolved by making these processes recursive. Similarly, naive recursion without the correct execution model of the underlying EVM over-approximates the state space since transactions in a block are executed after the transactions in the previous blocks, prompting the development of a general framework to reduce false positives in verification results.

5 Our Approach

The above analysis indicates that the challenges can be addressed by modelling the correct interpretation of the execution model of EVM. Instead of adding a process of the EVM merely working for this case study, we aim to tackle the problem in general, as we observe that there is a clear line of demarcation drawn between the *control logic* (the underlying blockchain) and the *business logic* (the smart contract functions) being common to the modelling of any smart contract, whether already deployed or yet to be developed. This inspired us to develop a modelling approach that can be extended into a standard practice in the form of a universal framework that can be easily and effectively exploited by an average smart contract verification engineer.

In a nutshell, we abstract away unnecessary details in the *control logic* and separate it from the actual *business logic*, which is unique and thus must be independently specified and tailored by the model engineer for each target smart contract so that the control logic common to all contract model specifications can be automatically generated by our framework to address the main challenge of misunderstanding issues, which eventually ties back to a reduction in the false positive rate of the verification results and in turn becomes an improvement in the precision of the verification outcomes in general while accounting for other concerns, including a faithful translation of the possibility of never-ending invocations of functions exposed by an alive contract on the chain and the incorporation of the environmental factors such as consensus clients into the framework to leave space for further extensions as part of the future work.

```
#include "blockchain_framework.csp";
#define INITIAL_VALUE 0;
#define ITEM_PRICE_VALUE 5;
enum {CONTRACT_ADDRESS, SELLER_ADDRESS,
       BUYER_ADDRESS};
enum {NULL_STATE, CREATED_STATE, LOCKED_STATE,
       RELEASE_STATE, INACTIVE_STATE};
channel constructor 0;
var contract = CONTRACT_ADDRESS;
var seller = SELLER_ADDRESS;
var buyer = BUYER_ADDRESS;
var state = NULL_STATE;
var value = INITIAL_VALUE;
#alphabet Constructor {smart_contract_initialized};
Constructor() =
  constructor?msg_sender.msg_value ->
  ifa (msg_value % 2 == 0) {
    seller_set_to.msg_sender{seller = msg_sender;}
       ->
    value_as_item_price_set_to_half_of_guaranty.
       msg_value{value = msg_value / 2;} ->
    state_transitioned_to_created{state =
       CREATED_STATE;} ->
    smart_contract_initialized -> Purchase()
  } else { odd_msg_value_exception.msg_value ->
       Constructor() };
#alphabet Abort {aborted, end};
Abort() =
  abort?msg_sender.msg_value ->
  ifa (msg_sender == seller) {
    ifa (state == CREATED_STATE) {
      state_transitioned_to_inactive{state =
       INACTIVE_STATE;} ->
      contract_balance_transferred_to_seller.
       balances[contract].seller{
        balances[seller] = balances[seller] +
       balances[contract];
        balances[contract] = 0; } ->
      aborted -> end -> Skip
    } else { non_created_state_exception.state ->
      end -> Skip }
  } else { non_seller_exception.msg_sender ->
    end -> Skip };
#alphabet ConfirmPurchased {purchase_confirmed, end};
ConfirmPurchased() =
  confirm_purchased?msg_sender.msg_value ->
  ifa (state == CREATED_STATE) {
    ifa (msg_value == 2 * value) {
      state_transitioned_to_locked{state = LOCKED_STATE
       ;} ->
      contract_balance_added_with_deposit_of.msg_value{
        balances[contract] = balances[contract] +
       msg_value;
        if (msg_sender == BUYER_ADDRESS) {
          balances[buyer] = balances[buyer] - msg_value;
        } else { balances[seller] = balances[seller] -
       msg_value; }
      } -> buyer_set_to.msg_sender{buyer = msg_sender;}
       -> purchase_confirmed -> end -> Skip
    } else { non_matching_msg_value_exception.msg_value
       -> end -> Skip }
  } else { non_created_state_exception.state -> end ->
       Skip };
#alphabet ConfirmReceived {item_received, end};
ConfirmReceived() =
  confirm_received?msg_sender.msg_value ->
  ifa (msg_sender == buyer) {
    if (state == LOCKED_STATE) {
      state_transitioned_to_release{state =
       RELEASE_STATE;} ->
      half_of_deposit_returned_to_buyer.value.msg_sender
       {
        balances[contract] = balances[contract] - value;
        if (msg_sender == BUYER_ADDRESS) {
          balances[buyer] = balances[buyer] + value;
        } else {
             balances[seller] = balances[seller] + value
       ; } } -> contract_balance_transferred_to_seller.
       balances[contract].seller{ balances[seller] =
       balances[seller] + balances[contract];
        balances[contract] = 0;
      } -> item_received -> end -> Skip
    } else { non_locked_state_exception.state ->
      end -> Skip }
  } else { non_buyer_exception.msg_sender -> end -> Skip
        };
```

Fig. 5: Correct Model of Safe Remote Purchase (part 1).

5.1 Correct CSP# Modelling

Following the approach, in the case study, the business logic model is roughly the same as the previous smart contract modelling in Figure 4, except that tiny modifications (detailed later) and syntactical renaming of events (see Figure 5). In addition, we added recursion to model that a function can be called multiple times, line 1-3 in Figure 6.

```
1  AbortR() = Abort(); AbortR();
2  ConfirmPurchasedR() = ConfirmPurchased();
          ConfirmPurchasedR();
3  ConfirmReceivedR() = ConfirmReceived();
          ConfirmReceivedR();
4  Purchase() = AbortR() ||| ConfirmPurchasedR() |||
          ConfirmReceivedR();
5  Deployer(address, guaranty_ether_value) =
6    constructor!address.guaranty_ether_value -> Skip;
7  Seller(address) =
8    abort_invocation!address.0 -> Seller(address);
9  Buyer(address, deposit_ether_value) =
10   confirm_purchased_invocation!address.
          deposit_ether_value ->
11   Buyer(address, deposit_ether_value)
12   [] confirm_received_invocation!address.0 ->
13   Buyer(address, deposit_ether_value);
14 BlockchainSystem() = Constructor() ||
15   (Deployer(SELLER_ADDRESS, ITEM_PRICE_VALUE * 2);
16     ((Seller(SELLER_ADDRESS) ||| Buyer ) |||
17       ( Buyer(BUYER_ADDRESS, ITEM_PRICE_VALUE * 2)
18                 ||| Seller(BUYER_ADDRESS)))
19   ) || BlockchainNetwork();
20 #define buyer_balance_remains_unchanged (balances[buyer] =
          = INITIAL_BALANCE);
21 #define seller_balance_remains_unchanged (balances[seller]
           == INITIAL_BALANCE);
22 #define buyer_balance_deducted_by_item_price_value (
          balances[buyer] == INITIAL_BALANCE -
          ITEM_PRICE_VALUE);
23 #define seller_balance_added_by_item_price_value (balances
          [seller] == INITIAL_BALANCE + ITEM_PRICE_VALUE);
24 #define buyer_is_BUYER_ADDRESS (buyer == BUYER_ADDRESS);
25 #define buyer_is_SELLER_ADDRESS (buyer == SELLER_ADDRESS);
26 #assert BlockchainSystem() |= [] (aborted -> <> (
          buyer_balance_remains_unchanged &&
          seller_balance_remains_unchanged));
27 #assert BlockchainSystem() |= [] (item_received &&
          buyer_is_BUYER_ADDRESS -> <> (
          buyer_balance_deducted_by_item_price_value &&
          seller_balance_added_by_item_price_value));
28 #assert BlockchainSystem() |= [] (item_received &&
          buyer_is_SELLER_ADDRESS -> <> (
          buyer_balance_remains_unchanged &&
          seller_balance_remains_unchanged));
29 #assert BlockchainSystem() reaches non_constant_balances;
```

Fig. 6: Correct Model of Safe Remote Purchase (part 2).

```
 1  #define N 3; #define INITIAL_BALANCE 100;
 2  #define EXIT_CODE_SUCCESS 0; #define EXIT_CODE_ERROR 1;
 3  var balances = [0, INITIAL_BALANCE, INITIAL_BALANCE];
 4  hvar counter = 0;
 5  channel abort_invocation 0; channel abort 0;
 6  channel confirm_purchased_invocation 0;
 7  channel confirm_purchased 0;
 8  channel confirm_received_invocation 0;
 9  channel confirm_received 0; channel release 0;
10  AbortExecutor() =
11    abort_invocation?msg_sender.msg_value ->
12    abort!msg_sender.msg_value ->
13    release?exit_code -> Skip;
14  ConfirmPurchasedExecutor() =
15    confirm_purchased_invocation?msg_sender.msg_value ->
16    confirm_purchased!msg_sender.msg_value ->
17    release?exit_code -> Skip;
18  ConfirmReceivedExecutor() =
19    confirm_received_invocation?msg_sender.msg_value ->
20    confirm_received!msg_sender.msg_value ->
21    release?exit_code -> Skip;
```

```
22  ExecutionClient(i) =
23    (start_execution_client.i -> Skip);
24    (AbortExecutor() [] ConfirmPurchasedExecutor() []
            ConfirmReceivedExecutor());
25    (end_execution_client.i -> Skip);
26    ExecutionClient(i);
27  ConsensusClient(i) =
28    [counter == i]
29    start_execution_client.i ->
30    end_execution_client.i ->
31    tau{counter = (counter + 1) % N} ->
32    ConsensusClient(i);
33  BlockchainNode(i) = ExecutionClient(i) ||
          ConsensusClient(i);
34  BlockchainNetwork() = BlockchainNode(0) ||
          BlockchainNode(1) || BlockchainNode(2);
35
36  #define non_constant_balances (balances[0] +
          balances[1] + balances[2] != 0 +
          INITIAL_BALANCE + INITIAL_BALANCE);
```

Fig. 7: Modelling the Blockchain in Control Logic (right) and its Interface (left).

The newly added model component is the control logic and its interface shown in Figure 7. In the control logic, we model an abstracted blockchain (line 34) consisting of Blockchain nodes (line 33) running a consensus algorithm (line 27). Each blockchain node has an EVM that stores all the smart contract functions and executes functions to update the global state upon invocation. In the case study, this is modelled by allowing the node to be able to execute all the smart contract functions *i.e.*, calling the interface of the smart contracts (line 24). Since the EVM is a single machine, meaning that each function is executed atomically, the relations between the executions are internal choices capturing that once called, the function execution is not interrupted by other functions in the same EVM, which is the key to avoiding false attacks. The non-determinism is modelled by the uncertainty of the node who proposes the block, *i.e.*, which node is the next to execute the triggered smart contract functions in EVM. In reality, the consensus algorithm decides the block proposer. There are various consensus algorithms and modelling them in details is challenging as stated in [22]. In this work, serving the purpose of reducing false attacks, does not require full details of the consensus algorithms. Therefore, we model an abstracted version in the consensusClient process (line 22-26), where the nodes take turns to be the block proposer. While some blockchain uses this model (*e.g.*, Tendermint implements a round-robin approach to decide the schedule of nodes), it is indeed a naive approach that may suffer from attacks (*e.g.*, malicious nodes may prepare a fork of the chain to launch double spending knowing the scheduling of the proposers). We assume the nodes are honest and leave the full detail modelling of consensus to future work.

The left part of Figure 7 (line 10-21) models the interfaces between the actual smart contract model (business logic) and the blockchain model (control logic). That is, once a proposer is decided in the ConsensusClient, the corresponding node executes ExecutionClient to call the smart contract function in the node's transaction pool. We assume there is only one transaction in a block to ensure the occurrence of *Multi-Transactional* behaviour. Extending to multiple transaction executions is easy by making them recursive. The calling of the transaction is implemented in the corresponding interface. Once called, the interface sends a message to trigger the actual smart contract function defined in the smart contract model (the business logic in Figure 5), thus linking the control logic (the blockchain) with the business logic.

Note that the events are renamed to provide meaningful information, compared to the replicated model where the same event names are used as in the [16], but they are semantically equivalent. In addition, to enable the interface to work correctly, the process of each smart contract function added a receiving message event at the beginning to model receiving signals from the interface *i.e.*, line 14, 23, 37 and 51 in Figure 5.

Verification Results. The verification results produced by running our CSP# model in the PAT model checker are shown in Table 2. Given that the assertions inherently convey their semantics, we refrain from reiterating them for the sake of brevity. The results prove *non-existence* of any vulnerabilities pertaining to the interleaved executions of contract functions and *infeasibility* of launching the attack described in Section 4.2 in practice to steal the funds deposited by the buyer, via demonstrating the validity of the last three assertions and the invalidity of the first assertion contradictory to the conclusion drawn from Table 1. We therefore conclude that the original contract design is in fact robust to any finely crafted attacks in the form of well planned sequences of function invocations and the result reported in the original literature [16] is challenged and shown to be a *false positive* alert based on our model built under the correct assumption of the execution model of Ethereum blockchain.

Modelling Strategy Comparisons. Essentially, integrating the blockchain framework into the modelling process consistently enforces mutual exclusive access to the execution of each constituent process that represents its respective contract function. This is achieved through cooperatively running execution clients and consensus clients of all participating nodes, as represented by the process BlockchainNetwork in our framework. Despite the semantic implications of the actual construct used — either interleaving operators or general choice operators — in defining the process, this integration characterises all possible interactions with the contract.

We have also effectively tackled the oversight of the potential scenario of successive attempts to invoke the function Abort, ConfirmPurchased, and ConfirmReceived multiple times in unanticipated order, independent of the current state of the contract (*e.g.*, even after the finalisation of the purchase order), so as to expand the derived state space to encompass these non-terminating be-

Assertion	**Validity**
BlockchainSystem() reaches non_constant_balances	NOT VALID
BlockchainSystem() |= [](item_received && buyer_is_SELLER_ADDRESS -> <>(buyer_balance_remains_unchanged && seller_balance_remains_unchanged))	VALID
BlockchainSystem() |= [](item_received && buyer_is_BUYER_ADDRESS -> <>(buyer_balance_deducted_by_item_price_value && seller_balance_added_by_item_price_value))	VALID
BlockchainSystem() |= [](aborted -> <>(buyer_balance_remains_unchanged && seller_balance_remains_unchanged))	VALID

Table 2: Verification Result of Our Correct CSP# Model.

haviours, ensuring comprehensive coverage of their side effects towards the overall correctness of the contract. In addition, to ensure it works correctly, their soundness criteria must also be properly model-checked during the verification phase, leading to the extra assertions and verification results in Table 2.

Correctness. The correctness of this CSP# model mostly lies in the accuracy of our framework with respect to the correspondence between the actual execution model of Ethereum and constructs used for modelling their abstracted behaviours (*e.g.*, scheduling of smart contract function executions), in particular, the use of those synchronous channels for both inter-node and intra-node communications, the use of the general choice operator in defining the process of the execution client, as well as the way in which a simplified blockchain node is defined in terms of cooperatively running execution/consensus clients. We have provided detailed justification in our Github report; for the sake of brevity, no further explanations are given here.

Limitations. In the current model, intricacies of network data transmission have been fully abstracted away from our framework and replaced by reliable channels for the sake of simplicity, and all possible interactions initiated by a third-party proxy contract have been intentionally omitted due to possible combinations of such interactions being infinite. We modelled an ideal consensus omitting failure in consensus mechanism *e.g.*, the actual order in which transactions are processed may differ from the expected order of executions when assuming no failures during the consensus procedure if dismissing this aspect of the environment when model checking any smart contract of interest.

5.2 A Generalized Formal Framework in CSP#

To generalise the approach beyond the Safe Remote Purchase case study, we formalised it into a framework. In particular, we implemented a script (see Figure 8) to generate the control logic automatically. Since the "business logic" of a smart contract is application-specific, it needs to be modelled manually.

To test the model generator and evaluate that the approach would not miss out on concurrency attacks, we verified the smart contract in Example 1 that truly suffers from concurrency issues to confirm the ability of our approach to detect concurrency vulnerabilities. We manually modelled the business logic of the smart contract[5]. The generated model of the "control logic" smart contract is shown in Figure 9, where the main difference lies in the generated interface (line 9-16) and their invocation (line 19) in the process of the blockchain nodes *i.e.*, ExecutionClient. We then integrated the smart contract model with the generated model. The verification result confirms the expected concurrency vulnerability.

These two case studies demonstrate that our framework empowers users to model with greater precision and accuracy, the *non-determinism* stemming from transaction races *i.e.*, invocation requests to contract functions, which are often mishandled by average model engineers, leading to the inclusion of unlikely

[5] The modelling is straightforward and thus we do not explain them here. For details, refer to our Github.

```
const fs = require("fs");
fs.readFile("input.json", "utf8", (err, data) => {
  if (err) {console.error(err); return; }
  try {
    const capitalize = (string) => string.charAt(0)
          .toUpperCase() + string.slice(1).
          toLowerCase();
    const jsonData = JSON.parse(data);
    const outputString =
        "#define N " + jsonData.n + ";\n" +
        "#define INITIAL_BALANCE " + jsonData.
              initialBalance + ";\n" +
        "#define EXIT_CODE_SUCCESS 0;\n" +
        "#define EXIT_CODE_ERROR 1;\n\n" +
        "var balances = [0, " + (new Array(jsonData
              .n - 1).fill("INITIAL_BALANCE")).join
              (", ") + "];\n" +
        "hvar counter = 0;\n\n" +
        jsonData.functionNames.map((functionName)
              => "channel " + functionName + "
              _invocation 0;\n" +
          "channel " + functionName + " 0;\n"
        ).join('') +
        "channel release 0;\n\n" +
        jsonData.functionNames.map((functionName)
              =>
          functionName.split('_').map(capitalize).
                join('') + "Executor() = \n" +
          "  " + functionName + "_invocation?
                msg_sender.msg_value -> \n" +
          "  " + functionName + "!msg_sender.
                msg_value -> \n" +
          "  release?exit_code -> \n" + " Skip;\n"
        ).join('\n') + '\n' +
        "ExecutionClient(i) = \n" +
        "  (start_execution_client.i -> Skip);\n" +
        "  (" + jsonData.functionNames.map((functionName)
               => functionName.split('_').map(capitalize)
              .join('') + "Executor()").join(" [] ") + ")
              ;\n" +
        "  (end_execution_client.i -> Skip);\n" +
        "  ExecutionClient(i);\n\n" +
        "ConsensusClient(i) = \n" +
        "  [counter == i]\n" +
        "  start_execution_client.i -> \n" +
        "  end_execution_client.i -> \n" +
        "  tau{counter = (counter + 1) % N} -> \n" +
        "  ConsensusClient(i);\n\n" +
        "BlockchainNode(i) = ExecutionClient(i) ||
              ConsensusClient(i);\n\n" +
        "BlockchainNetwork() = BlockchainNode(0) ||
              BlockchainNode(1) || BlockchainNode(2);\n\n
              \n" +
        "#define non_constant_balances (" + Array.from(
              Array(jsonData.n).keys()).map((index) => "
              balances[" + index + "]").join(' + ') + "
              != 0 + " + (new Array(jsonData.n - 1).fill(
              "INITIAL_BALANCE")).join(" + ") + ");\n";
    fs.writeFile("./blockchain_framework.csp",
          outputString, (err) => {
        if (err) { console.error(err); return; }
        console.log("The blockchain framework to be
              included in your CSP model has been
              successfully generated under the same
              directory of this generator script.");});
        }  catch (parseError) {
        console.error(parseError); } });
```

Fig. 8: Blockchain Model Generator.

```
#define N 3; #define INITIAL_BALANCE 100;
#define EXIT_CODE_SUCCESS 0;
#define EXIT_CODE_ERROR 1;
var balances = [0, INITIAL_BALANCE,
      INITIAL_BALANCE];
hvar counter = 0;
channel get_invocation 0; channel get 0;
channel set_invocation 0;
channel set 0; channel release 0;
GetExecutor() =
   get_invocation?msg_sender.msg_value ->
   get!msg_sender.msg_value ->
   release?exit_code -> Skip;
SetExecutor() =
   set_invocation?msg_sender.msg_value ->
   set!msg_sender.msg_value ->
   release?exit_code -> Skip;
ExecutionClient(i) =
   (start_execution_client.i -> Skip);
   (GetExecutor() [] SetExecutor());
   (end_execution_client.i -> Skip); ExecutionClient(i);
ConsensusClient(i) =
   [counter == i] start_execution_client.i ->
   end_execution_client.i ->
   tau{counter = (counter + 1) % N} ->
   ConsensusClient(i);
BlockchainNode(i) = ExecutionClient(i) ||
      ConsensusClient(i);
BlockchainNetwork() = BlockchainNode(0) ||
      BlockchainNode(1) || BlockchainNode(2);
#define non_constant_balances (balances[0] + balances[1]
        + balances[2] != 0 + INITIAL_BALANCE +
      INITIAL_BALANCE);
```

Fig. 9: CSP# Blockchain Framework for Example 1 Smart Contract.

execution sequences in the state space for verification, resulting in false attack alarm and unnecessary labour for rectifying them. Our primary contribution lies in reducing *false positive* cases by correctly modelling all possible transaction orders and reducing *false negative* cases by including potential conflicting transaction executions in the state space. Leveraging our generator script, which separates *control logic* from *business logic*, enables more diverse modelling strategies: Users can use the interleaving operator to compose the smart contract functions without considering how exactly the underlying blockchain schedules the actual execution sequence; The common application-agnostic details observed in all types of smart contract are abstracted into a separate model component *i.e.*, the blockchain framework, for automatic generation. This framework frees the model engineers from the need to possess this knowledge, thereby enhancing the *generalizability* of our framework.

6 Related Works

Security of smart contracts and blockchain vulnerabilities have been extensively surveyed in [13,4,9]. Concurrency issues discussed in this work spans over several surveyed attacks, as shown in Section 2.

To address smart contract vulnerabilities, various verification techniques (surveyed in [2]) and tools (surveyed in [3]) have been developed, including testing-based approaches (like [11]) and static analysis based approaches (*e.g.*, symbolic execution [13]). These approaches heavily rely on known patterns and cannot guarantee correctness and security. On the other hand, formal verification approaches that overcome such limitations, have also been developed (surveyed in [24]) including theorem proving based approaches *e.g.*, [20], model checking approaches *e.g.*, [12] and abstract state machines based approaches *e.g.*., [6,5]. However, these approaches verify smart contracts independently. Exceptions that consider other participants in specific attacks have been discussed in Section 1. In contrast, there are much less works on formal verification of blockchain due to its complexity, with exceptions including [23,7,14].

The concurrency issue in this work involves the specification and verification of both smart contracts and their blockchain environment. Notable techniques for addressing concurrency issues have been introduced in Section 1.

7 Conclusions and Future Work

Aiming to assist model engineers in constructing correct and precise models of smart contracts that involve *non-determinism* arising from the races between transactions, which are often overlooked or mistakenly handled by an average model engineer, this work proposes a formal framework that facilitates a holistic inclusion of transaction executions to reduce the false attacks in verifying concurrency of smart contracts, achieved by recognising the commonality of the separation of the control logic from the business logic of the smart contracts.

Recognising the limitations of the framework (as mentioned earlier), the primary focus of future work is to address them by offering more detailed consensus and network integration. Additionally, another area for future exploration involves integrating the other types of concurrency in smart contract execution into the framework, where the challenge is the necessary manual translation.

References

1. Post-mortem investigation. `https://www.kingoftheether.com/postmortem.html` (2016), retrived at 11 April 2024
2. Almakhour, M., Sliman, L., Samhat, A.E., Mellouk, A.: Verification of smart contracts: A survey. Pervasive and Mobile Computing **67**, 101227 (2020)
3. Angelo, M.D., Salzer, G.: A survey of tools for analyzing ethereum smart contracts. In: DAPPS. pp. 69–78 (2019)
4. Atzei, N., Bartoletti, M., Cimoli, T.: A survey of attacks on ethereum smart contracts (sok). In: POST. pp. 164–186 (2017)

5. Braghin, C., Riccobene, E., Valentini, S.: An asm-based approach for security assessment of ethereum smart contracts. In: SECRYPT. pp. 334–344 (2024)
6. Braghin, C., Riccobene, E., Valentini, S.: Modeling and verification of smart contracts with abstract state machines. In: SAC. pp. 1425–1432 (2024)
7. Braithwaite, S., Buchman, E., Konnov, I., Milosevic, Z., Stoilkovska, I., Widder, J., Zamfir, A.: Formal specification and model checking of the tendermint blockchain synchronization protocol (short paper). In: FMBC (2020)
8. Breidenbach, L., Juels, P.D.A., Sirer, E.G.: An in-depth look at the parity multisig bug. `https://hackingdistributed.com/2017/07/22/deep-dive-parity-bug/` (2017), retrived at 11 April 2024
9. Chen, H., Pendleton, M., Njilla, L., Xu, S.: A survey on ethereum systems security: Vulnerabilities, attacks, and defenses. ACM Comput. Surv. **53**(3) (2021)
10. Daian, P., Goldfeder, S., Kell, T., Li, Y., Zhao, X., Bentov, I., Breidenbach, L., Juels, A.: Flash boys 2.0: Frontrunning in decentralized exchanges, miner extractable value, and consensus instability. In: SP. pp. 910–927 (2020)
11. Grieco, G., Song, W., Cygan, A., Feist, J., Groce, A.: Echidna: effective, usable, and fast fuzzing for smart contracts. In: ISSTA. pp. 557–560 (2020)
12. Kalra, S., Goel, S., Dhawan, M., Sharma, S.: Zeus: analyzing safety of smart contracts. In: Ndss. pp. 1–12 (2018)
13. Luu, L., Chu, D., Olickel, H., Saxena, P., Hobor, A.: Making smart contracts smarter. In: CCS. pp. 254–269 (2016)
14. Maung Maung Thin, W.Y., Dong, N., Bai, G., Dong, J.S.: Formal analysis of a proof-of-stake blockchain. In: ICECCS. pp. 197–200 (2018)
15. Munir, S., Reichenbach, C.: Todler: A transaction ordering dependency analyzer - for ethereum smart contracts. In: WETSEB. pp. 9–16 (2023)
16. Qu, M., Huang, X., Chen, X., Wang, Y., Ma, X., Liu, D.: Formal verification of smart contracts from the perspective of concurrency. In: SmartBlock. LNCS, vol. 11373, pp. 32–43 (2018)
17. Sergey, I., Hobor, A.: A concurrent perspective on smart contracts. In: FC. LNCS, vol. 10323, pp. 478–493 (2017)
18. Siegel, D.: Understanding the DAO attack. `https://www.coindesk.com/learn/understanding-the-dao-attack/`, retrived at 11 April 2024
19. Solidity: Solidity by example safe remote purchase. `https://docs.soliditylang.org/en/latest/solidity-by-example.html`, retrived on 16 April 2024
20. Sotnichek, M.: Formal verification of smart contracts with the k framework. `https://www.apriorit.com/dev-blog/592-formal-verification-with-k-framework` (2019), retrived at 23 Spet. 2024
21. Sun, J., Liu, Y., Dong, J.S., Pang, J.: PAT: towards flexible verification under fairness. In: CAV. LNCS, vol. 5643, pp. 709–714 (2009)
22. Thin, W.Y.M.M., Dong, N., Bai, G., Dong, J.S.: Formal analysis of a proof-of-stake blockchain. In: ICECCS. pp. 197–200 (2018)
23. Tholoniat, P., Gramoli, V.: Formal verification of blockchain byzantine fault tolerance. In: Handbook on Blockchain, pp. 389–412 (2022)
24. Tolmach, P., Li, Y., Lin, S., Liu, Y., Li, Z.: A survey of smart contract formal specification and verification. ACM Comput. Surv. **54**(7), 148:1–148:38 (2022)
25. Wang, Y., Li, J., Liu, W., Tan, A., Derhab, A.: Efficient concurrent execution of smart contracts in blockchain sharding. Sec. and Commun. Netw. (2021)
26. Wood, G.: Ethereum: A secure decentralised generalised transaction ledger. `https://ethereum.github.io/yellowpaper/paper.pdf` (2019)
27. Xu, X., Pautasso, C., Zhu, L., Lu, Q., Weber, I.: A pattern collection for blockchain-based applications. In: EuroPLoP. pp. 3:1–3:20 (2018)