Temporal Properties of Smart Contracts

Ilya Sergey[1], Amrit Kumar[2], and Aquinas Hobor[3]

[1] University College London, United Kingdom
i.sergey@ucl.ac.uk
[2] Zilliqa Research, Singapore
amrit@zilliqa.com
[3] Yale-NUS College and School of Computing, NUS, Singapore
hobor@comp.nus.edu.sg

Abstract. Smart contracts—shared stateful reactive objects stored on a blockchain—are widely employed nowadays for mediating exchanges of crypto-currency between multiple untrusted parties. Despite a lot of attention given by the formal methods community to the notion of smart contract correctness, only a few efforts targeted their *lifetime* properties. In this paper, we focus on reasoning about execution traces of smart contracts. We report on our preliminary results of mechanically verifying some of such properties by embedding a smart contract language into the Coq proof assistant. We also discuss several common scenarios, all of which require multi-step blockchain-based arbitration and thus must be implemented via stateful contracts, and discuss possible temporal specifications of the corresponding smart contract implementations.

1 Introduction

Smart contracts are stateful reactive objects that are stored on a blockchain and serve as mediators for multi-party fund-transferring computations. The last three years have seen a proliferation of smart contracts implementing various decentralised applications (Dapps) on top of the Ethereum blockchain [26]. During this period of ongoing early adoption, the smart contract technology provided by Ethereum has witnessed a number of serious hurdles, manifested by various safety and security vulnerabilities in the deployed implementations and resulting in the losses of USD millions' worth of cryptocurrency [4, 10]. Since, once deployed to the blockchain, a contract's implementation cannot be amended, the challenge of identifying the contracts' "good" and "bad" behaviours at the stage of development becomes particularly acute.

In order to ensure the absence of unwelcome outcomes, it is important to be able to reason about safety and liveness of contract executions across multiple transactions and about its possible interactions with other contracts or users. One representative high-level *safety* issue, manifested in multi-transactional contract executions with oracles, is a presence of race conditions, that might leave a contract in an inconsistent state due to unaccounted multiple parties interacting with it in different moments of time, commonly happening while communicating with external oracles [22]. Improperly incentivizing the parties taking different roles in a contract's execution might lead to denial-of-service leaving funds permanently blocked—a violation of an implicitly assumed liveness property (meaning, informally, that *eventually* the funds can be retrieved by a well-behaved party) [5, 19]. Detecting such contract instances for the sake of informing the developers, *before* they are deployed, requires techniques for specifying what is considered to be *correct* contract behaviours, and whether a given implementation always adheres to this specification.

In this paper, we make an observation that many behavioural properties of smart contracts that are considered "natural" can be only captured in reference to their *multi-step executions*, by defining relations on a contract's state in different moments of time, thus, corresponding to well-studied temporal properties of programs and state-transition systems [15,21]. We substantiate this claim and demonstrate the utility of temporal reasoning in application to smart contracts by using SCILLA, a recently proposed principled programming model for representing stateful contracts as communicating state-transition systems [23], to express simplified implementations of several classes of popular Dapps. We then sketch the execution semantics of SCILLA smart contracts and use it to define the notion of contract execution traces. Using this trace-based semantics, we then state a number of temporal properties, capturing the notion of particular classes of "well-behaved" smart contracts. Finally, we report on some preliminary results of mechanising the temporal reasoning by encoding SCILLA and its semantics into Coq proof assistant [8].

In this manuscript, we do *not* attempt to design a new set of *temporal logic* connectives for specifying contract properties. Instead, we demonstrate how the natural properties of execution traces can be encoded and proved by means of shallow embedding into Coq's higher-order logic [9], leaving the formal description of the standalone temporal logic for smart contracts as our future work.

2 Overview and Motivation

Let us consider a fragment of the infamous BlockKing contract [1], taken directly from the Ethereum mainnet.[4] Its code in SOLIDITY [25] is presented in in Figure 1. This contract has been a popular testbed for several analyses for smart contracts recently, due to its flawed implementations, prone to concurrency errors [22], commutativity violations [6], and dynamically-determined resource consumption [7]. The defining feature of this contract is interaction with an off-chain oracle service Oraclize by means of calling the `oraclize_query()` function in line 303, so that an oracle can return an expected result by calling the `__callback()` function in line 306. The crux of the problematic behaviour is in the three mutable fields of the BlockKing contract: `warrior`, `warriorGold`, and `warriorBlock`, all of which, after having been set by call to `enter()` in a transaction tx_1, can be later *overriden* by a transaction tx_2 of a competing client of the same contract when executed concurrently.

In this scenario, an oracle's response via `__callback()` might return the value for the value "meant" for the values of the fields set by tx_1 that are no longer

[4] At the moment of this writing, the contract still holds approximately 0.043 ETH.

```
function enter() {
 // 100 finney = .05 ether minimum payment otherwise refund payment and stop contract
 if (msg.value < 50 finney) {
  msg.sender.send(msg.value);
  return;
 }
 warrior = msg.sender;
 warriorGold = msg.value;
 warriorBlock = block.number;
 bytes32 myid =
  oraclize_query(0,"WolframAlpha","random number between 1 and 9");
}

function __callback(bytes32 myid, string result) {
 if (msg.sender != oraclize_cbAddress()) throw;
 randomNumber = uint(bytes(result)[0]) - 48;
 process_payment();
}

function process_payment() {

...

 if (singleDigitBlock == randomNumber) {
  rewardPercent = 50;
  // If the payment was more than .999 ether then increase reward percentage
  if (warriorGold > 999 finney) {
   rewardPercent = 75;
  }
  king = warrior;
  kingBlock = warriorBlock;
 }
```

Figure 1. Fragments of the smart contract implementing the BlockKing game.

present (since they are overriden by tx_2), whereas the sender of tx_2 will enjoy the double reward, "cashing out" both the results of its own game and also when doing so "on behalf" of tx_1 sender's.

While multiple ways to identify this problem exist, by employing either concurrency [22], resource [7] or commutativity reasoning [6], we consider this example as an opportunity to provide a "morally correct" specification to the functionality of this game-implementing contract that has to do with identifying the reward by means of taking a random input from an oracle, and transferring this reward to the corresponding player. One way to state the desired property semiformally in the style of Lamport [16] is by means of demanding certain causality between the two *events* in the contract's execution history: entering a game and executing a callback. This can be done as follows:

Property 1 (Correctness of BlockKing *payment processing).* Any call to `enter()` from a *sender* account a sets the value of the field `warrior` to a, so when the next call to `__callback()` by an oracle takes place, the value of `warrior` is still a.

Obviously, for the given implementation in Figure 1 does not hold, as they can be violated in the presence of the concurrent transactions. In order to ensure this property, the contract can be fixed by, for instance, enhancing it with a *locking* discipline, prohibiting other players to enter the game before the callback is executed, with the obvious drawback of such a solution that would make the contract prone to DoS attacks. A more clever approach would require one to engineer a register of the players who currently have entered the game but have not got their payments processed.

While fixing the BlockKing contract is not the topic of this paper, this example should make apparent the importance of *temporal* properties of smart contract implementations, relating the effects of events (such as receiving requests and sending funds) taking place at certain *moments of time*, as well as the contract's state at those moments. However, even writing such temporal specification formally for Solidity or EVM contracts is far from trivial, due to (a) intricate control-flow patterns, (b) dependence of one contract's logic on another contract's state and (c) the presence of the implicit execution stack.

To address this specification challenge, we designed of a programming framework for smart contracts and an accompanying semantic formalism that separate and streamline the computation/communication aspects of contracts and allow for natural specifications and verification of safety and liveness properties.

3 The Language and Semantic Model

In order to enable formal reasoning about complex behaviour of stateful smart contracts, we designed SCILLA: a novel intermediate-level programming language for smart contracts [23]. By "intermediate" we mean that we do not expect most programmers to write in SCILLA directly, any more than most programmers write in `x86` assembly directly. Instead, the typical path will be to compile a higher-level language to SCILLA and then further to an executable bytecode, very much in a tradition of optimising [20] and verified compilers [17]. SCILLA aims to achieve both *expressivity* and *tractability*, while enabling rigorous formal reasoning about contract behavior, by adopting the following fundamental design principles, based on separation of programming concerns:

Separation between computation and communication. Contracts in SCILLA are structured as *communicating automata*: every in-contract computation (*e.g.*, changing its balance or computing a value of a function) is implemented as a standalone, atomic *transition*, *i.e.*, without involving any other parties. Whenever such involvement is required (*e.g.*, for transferring control to another party), a transition would end, with an explicit communication, by means of sending and receiving messages. The automata-based structure makes it possible to disentangle the contract-specific effects (*i.e.*, transitions) from blockchain-wide interactions (*i.e.*, sending/receiving funds and messages), thus providing a clean reasoning mechanism about contract composition and invariants.

Separation between effectful and pure computations. Any in-contract computation happening within a transition has to terminate, and have a predictable effect on the state of the contract and the execution. In order to achieve this, we draw inspiration from *functional programming* with effects, drawing a distinction between pure expressions (*e.g.*, expressions with primitive data types and maps), impure local state manipulations (*i.e.*, reading/writing into contract fields) and blockchain reflection (*e.g.*, reading current block number). By carefully designing semantics of interaction between pure and impure language aspects, we ensure a

```
contract Crowdfunding
 (owner     : address,
  max_block : uint,
  goal      : uint)

(* Mutable state description *)
{
  backers : address ⇒ uint = [];
  funded  : boolean = false;
}

(* Transition 1: Donating money *)
transition Donate
  (sender : address, value : uint,
   tag : string)
  (* Identifying this transition *)
  if tag == "donate" ⇒
  bs ← & backers;
  blk ← && block_number;
  let nxt_block = blk + 1 in
  if max_block ≤ nxt_block
  then send ⟨to → sender, amount → 0,
             tag → "main",
             msg → "deadline_passed"⟩
  else
    if not (contains(bs, sender))
    then
      let bs1 =
          put(bs, sender, value) in
      backers := bs1;
      send ⟨to → sender,
            amount → 0,
            tag → "main",
            msg → "ok"⟩
    else
      send ⟨to → sender,
            amount → 0,
            tag → "main",
            msg → "already_donated"⟩

(*  Transition 2: Sending the funds to the owner *)
transition GetFunds
  (sender : address, value : uint, tag : string)
  (* Only the owner can get the money back *)
  if (tag == "getfunds") && (sender == owner) ⇒
  blk ← && block_number;
  bal ← & balance;
  if max_block < blk
  then if goal ≤ bal
       then
         funded := true;
         send ⟨to → owner, amount → bal,
               tag → "main", msg → "funded"⟩
       else send ⟨to → owner, amount → 0,
                  tag → "main", msg → "failed"⟩
  else send ⟨to → owner, amount → 0, tag → "main",
             msg → "too_early_to_claim_funds"⟩

(* Transition 3: Reclaim funds by a backer *)
transition Claim
  (sender : address, value : uint, tag : string)
  if tag == "claim" ⇒
  blk ← && block_number;
  if blk ≤ max_block
  then send ⟨to → sender, amount → 0, tag → "main",
             msg → "too_early_to_reclaim"⟩
  else bs ← & backers;
       bal ← & balance;
       if (not (contains(bs, sender))) || funded ||
          goal ≤ bal
       then send ⟨to → sender, amount → 0,
                  tag → "main",
                  msg → "cannot_refund"⟩
       else
       let v = get(bs, sender) in
       backers := remove(bs, sender);
       send ⟨to → sender, amount → v, tag → "main",
             msg → "here_is_your_money"⟩
```

Figure 2. Crowdfunding contract in Scilla: state and transitions.

number of foundational properties about contract transitions, such as progress and type preservation, while also making them amenable to interactive and/or automatic verification with standalone tools.

Structuring contracts as communicating automata provides a computational model, known as *continuation-passing style* (CPS), in which every call to an external function (*i.e.*, another contract) can be done as the absolutely last instruction. That is, programming in Scilla naturally forces the programmer to express the computations with the contract as standalone transitions, performed *atomically*, *i.e.*, without the intermediate interaction with other contracts and relying only on the received messages.

3.1 Syntax of Scilla

Figure 2 shows a Scilla implementation of a crowdfunding campaign à la Kickstarter. In a crowdfunding campaign, a project owner wishes to raise funds through donations from the community. In the specific example modelled here, we assume that the owner wishes to run the campaign for a certain pre-determined period of time. The owner also wishes to raise a minimum amount of funds without which the project can not be started. The campaign is deemed successful if the owner can raise the minimum goal. In case the campaign is unsuccessful, the donations are returned to the project backers who contributed during the campaign. The design of the Crowdfunding contract is intentionally simplistic (for example, it does not allow the backers to change the amount of their donation), yet it shows the important features of SCILLA, which we elaborate upon.

The contract is parameterised with three values that will remain immutable during its lifetime (lines 2–4): an owner account address `owner` of type `address`, a maximal block number `max_block` (of type `uint`), indicating a deadline, after which no more donations will be accepted from backers, and a `goal` (also of type `uint`) indicating the amount of funds the owner plans to raise. The `goal` is not a hard cap but rather the minimum amount that the owner wishes to raise. What follows is the block of mutable *field declarations* (lines 7–10). The mutable fields of the contract are the mapping `backers` (of type `address` ⇒ `uint`), which will be used to keep track of the incoming donations and is initialised with an empty map literal `[]`, and a mutable boolean flag `funded` that indicates whether the owner has already transferred the funds after the end of the campaign (initialised with `false`). In addition to these fields, any contract in SCILLA has an implicitly declared mutable field `balance` (initialised upon the contract's creation), which keeps the amount of funds held by the contract.

The logic of the contract is implemented by three *transitions*: `Donate`, `GetFunds`, and `Claim`. The first one serves for donating funds to a campaign by external backers; the second allows the owner to transfer the funds to its account once the campaign is ended and the goal is reached; the final one makes it possible for the backers to reclaim their funds in the case the campaign was not successful.

One can think of transitions as methods or functions in Solidity contracts. What makes them different from functions, though, is the atomicity of the computation enforced at the language level. Specifically, each transition manipulates *only* with the state of the contract itself, without involving any other contracts or parties. All interaction with the external world, with respect to the contract, happens either at the very start of a transition, when it is initiated by an external message, or at the end, when a message (or messages), possibly carrying some amount of funds, can be emitted and sent to other parties.

Each transition can be invoked by a suitable message, which should provide a corresponding *tag* as its component to identify which transition is triggered. It is enforced at the compile time that tags define transitions unambiguously. All other components of the message, relevant for the transition to be executed, are declared as the transition's parameters. For instance, the transition `Donate` expects the incoming message to have at least the fields `sender`, `value`, and `tag`. What follows in each transition's definition is the *filter*—an optional clause `if e` ⇒, where `e` is a boolean-returning computation that can involve reading from the components of the incoming message and the contract's state, deciding whether the corresponding transition can be taken.

Every transition's last command, in each of the execution branches, is either sending a set of messages, or simply returning. Messages are encoded as vectors

⟨...⟩ of name → value entries, including at least the destination address (`to`), an amount of funds transferred (`amount`) and a default tag of the function to be invoked (`tag`). All transitions of the Crowdfunding end by sending a message to either the sender of the initial request or the contract's owner. For example, depending on the state of the contract and the blockchain, the transition `GetFund` might end up in either sending a message with its balance to the contract's owner, if the campaign has succeeded and the deadline has passed, or zero funds with a corresponding text otherwise.

The state of the contract, represented by its fields, is mutable: it can be changed by the contract's transitions. A body of a transition can *read* from the fields, assigning the result to immutable stack variables using the specialised syntax `x ←& f;`, where `f` is a field name and `x` is a fresh name of a local variable (*e.g.*, lines 18 and 46). In a similar vein, a body of transition can *store* a result of a pure expression `e` into a contract field `f` using the syntax `f := e;` (as in lines 30 and 50). The dichotomy between pure expressions (coming with corresponding binding form `let-in`) and impure ("effectful") commands manipulating the field values, is introduced on purpose to facilitate logic-based verification of contracts, reasoning about the effect of a transition to the contract's state, while abstracting away from evaluation of pure expressions.

In addition to reading/writing contract state, each transition implementation can use read-only introspection on the current state of the blockchain using the "deep read" operation `x ←&& g;`, where `g` is a name of the corresponding aspect of the underlying blockchain state. For example, the Crowdfunding contract reads the number of a current block in lines 19 and 45.

3.2 Semantics

We are developing Scilla hand-in-hand with the formalisation of its semantics and its embedding into the Coq proof assistant [8].[5] We now briefly outline the key components of our formalisation of the trace semantics of Scilla contracts. We will not explain the entire syntax of our Coq encoding, for which we refer the reader to the accompanying technical report [23].

Figure 3 provides Coq definitions of a small-step operational semantics `step_prot` of a contract `C` by means of executing, for the contract pre-state `pre`, in the blockchain state `bc`, an applicable transition, which is uniquely determined by an incoming message `m`, via `apply_transition`, and changing the contract's state and balance accordingly. The sequence of such changes contributes for a particular *schedule* `sc` of incoming messages contributes an execution traces, as defined by the function `execute`.

3.3 Higher-order trace predicates

With the operational semantics and the definition of traces at hand, we can now proceed to defining trace predicates for specifying relevant contract properties.

We first define a predicate `I` on a contract state (denoted, in Coq terms, by a "function type" `cstate S →Prop` from the type of states `cstate S` to propositions

[5] The mechanised embedding of a subset of Scilla into Coq is publicly available for downloads and experiments: https://github.com/ilyasergey/scilla-coq.

```
(* In the following definition, a contract automata C is implicit and fixed. *)
Definition step_prot (pre : cstate S) (bc : bstate) (m : message) : step :=
  let CState id bal s := pre in
  let (s', out) := apply_transition C id bal s m bc in
  let bal' := if out is Some m'
              then (bal + val m) - val m' else bal in
  let post := CState id bal' s' in
  Step pre post out.

(* Map a schedule into a trace *)
Fixpoint execute (pre : cstate S) (sc: schedule) : trace :=
  if sc is (bc, m) :: sc'
  then let stp := step_prot pre bc m in stp :: execute (post stp) sc'
  else [::].

Definition state0 := CState (acc C) (init_bal C) (init_state C).
Definition execute0 sc := if sc is _ :: _ then execute state0 sc else [:: Step state0 state0 None].
```

Figure 3. Contract traces and semantics.

Prop) to be a *safety property* if it holds at any state of a contract, that can be obtained as a result of interaction between the contract and its environment, starting from the initial state. The following Coq definition states this formally:

```
Definition safe (I : cstate S → Prop) : Prop :=
  (* For any schedule sc, pre/post states and out... *)
  ∀ sc pre post out,
  (* s.t. triple Step (pre, post, out) is in the sc-induced trace *)
  Step pre post out ∈ execute0 sc →
  (* both pre and post satisfy I *)
  I pre ∧ I post.
```

A safety property means some universally true correctness condition holds at any contract's state, which is reachable from its initial configuration via *any* schedule `sc`. Typical examples of safety properties of interest include: "a contract's balance is always positive", "a contract's balance equals the sum of balances of its contributors", or "at any moment no money is blocked on the contract". The definition above thus defines safety by universally quantifying over *all* schedules `sc`, as well as step-triples `Step pre post out` that occur in a trace, obtained by following `sc`.

As the next example, let us consider a temporal connective `since_as_long p q r`, which means the following: once the contract is in a state `st`, in which (*i*) the property `p` is satisfied, each state `st'` reachable from `st` (*ii*) satisfies a binary property `q st st'` (with respect to `st`), as long as (*iii*) every element of the schedule `sc`, "leading" from `st` to `st'` satisfies a predicate `r`.

The corresponding Coq encoding of the `since_as_long` connective is given below. We first specify reachability between states `st` and `st'` via a schedule `sc` as the state `st'` being the *last* post-state in a trace obtained by executing the contract from `st` via `sc`:

```
Definition reachable (st st' : cstate S) sc :=
     st' = post (last (Step st st None) (execute st sc)).
```

We next employ the definition of reachability to define the `since` connective, which is parameterised by predicates `p`, `q` and `r`. The premises (*i*)–(*iii*) are outlined in the corresponding comments in the following Coq code:

```
(* q holds since p, as long as schedule bits satisfy r. *)
Definition since_as_long (p : cstate S → Prop)
                         (q : cstate S → cstate S → Prop)
                         (r : bstate * message → Prop) := ∀ sc st st',
    (* (i) st satisfies p *)
    p st →
    (* (ii) st' is reachable from st via sc *)
    reachable st st' sc →
    (* (iii) any element b of sc satisfies r *)
    (∀ b, b ∈ sc → r b) →
    (* (conclusion) q holds over st and st' *)
    q st st'.
```

Why this logical connective is useful for reasoning about contract correctness? As we will show further, it makes it possible to concisely express "preservation" properties relating contract balance and state, so that they hold as long as certain actions do not get triggered by some of the contract's users.

4 Specifying and Verifying Trace Properties

We now show how the combination of notions of safety and temporal properties presented in Section 3.3 allows us to verify a contract, proving that all its behaviours satisfy a certain complex interaction scenario.[6] Specifically, for our Crowdfunding example, let us prove that, once a donation `d` has been made by a backer with an account address `b`, given that the campaign eventually fails, the backer `b` will be always able to get their donation `d` back. This can be obtained as the conjunction of the following three properties embodying both safety and temporal reasoning.

Property 2 (No leaking funds). The contract's accounted funds do not decrease unless the campaign has been funded or the deadline has expired.

In our Coq formalisation, this property can be captured via the following definition `balance_backed` and the accompanying safety theorem, stating that is *always* holds:

```
Definition balance_backed st : Prop :=
  (* If the campaign has not been funded... *)
  ¬ funded (state st) →
  (* the contract has enough funds to reimburse all. *)
  sumn (map snd (backers (state st))) <= balance st.
```

For an arbitrary contract state `st`, it asserts that if the `funded` flag is still `false` in `st` (*i.e.*, `¬funded (state st)`), then the balance of the contract (`balance st`) is at least as large as the sum of all donations made by the recorded backers (`sumn (map snd (backers (state st)))`).

[6] All definitions, theorems and proofs are in the accompanying Coq development.

```
Theorem no_leaking_funds : safe balance_backed.
```

The second property, which is temporal and it relates several states during the contract's lifetime is informally stated as follows:

Property 3 (Donation record preservation). The contract preserves records of individual donations by backers, unless they interact with it.

To specify this property and state the corresponding theorem we rely on the temporal connective `since_as_long` defined above and state that, once a backer made a donation, the record of it is not going to be lost by the contract, *as long as* the backer makes no attempt to withdraw its donation.

```
(* Contribution d of a backer b is recorded in the field 'backers'. *)
Definition donated b (d : value) st := get (backers (state st)), b) == d.

(* b doesn't claim its funding back *)
Definition no_claims_from b (q : bstate * message) := sender q.2 != b.

Theorem donation_record_preservation (b : address) (d : value):
  since_as_long c (donated b d)
                  (fun _ s' ⇒ donated b d s')
                  (no_claims_from b).
```

By now we know that the contract does not lose the donated funds and keeps the backer records intact. Now we need the last piece: the proof that if a contract is not funded, and the campaign has failed (deadline has passed and the goal has not been reached), then any backer with the corresponding record can *eventually* get the donation back, hence the following property:

Property 4 (Can get refund). If the campaign fails, the backers can eventually get their refund.

We state the property of interest as theorem `can_get_refund` in Figure 4. As its premises (a)–(d), the theorem lists all the assumptions about the state of the contract that are necessary for getting the reimbursement. The conclusion is somewhat peculiar: it expresses the *possibility* to claim back the funds by postulating the *existence* of a message `m`, such that it can be sent by a backer `b`, and the response will be a message with precisely `d` funds in it, sent back to `b`. The theorem, whose proof is only 10 lines of Coq, formulates the property as one single-step, yet its statement can be easily shown to be a safety property, as it is, indeed, preserved by the transitions, and, after the funds are successfully claimed for the first time, the premise (a) of the statement is going to be false, hence the property will trivially hold.

Properties 2–4 deliver the desired correctness condition of a contract: *once donated money can be claimed back in the case of a failed campaign.* It is indeed not the only notion of correctness that intuitively should hold over this particular contract, and by proving it we did not ensure that the contract is "bug-free". For instance, in our study we focused on backers only, while another legit concern would be to formally verify that the contract's owner will be able transfer the cumulative donation to their account in the case if the campaign is *successful*.

```
Theorem can_get_refund id b d st bc:
  (* (a) The backer b has donated d, so the contract holds
         that record in its state *)
  donated b d st →
  (* (b) The campaign has not been funded. *)
  ¬ funded (state st) →
  (* (c) Balance is small: not reached the goal. *)
  balance st < (get_goal (state st)) →
  (* (d) Block number exceeds the deadline. *)
  get_max_block (state st) < block_num bc →
  (* (conclusion) Backer b can get their donation back. *)
  ∃ (m : message),
    sender m == b ∧
    out (step_prot c st bc m) = Some (Msg d id b 0 ok_msg).
```

Figure 4. A backer can claim back her funds if the campaign fails.

5 More Temporal Properties of Common Contracts

We now show two more stateful smart contracts, which commonly occur on Ethereum blockchain, implemented in SCILLA, informally outlining temporal properties of interest one should aim to prove over their implementations.

5.1 Properties of Auctions

Figure 5 shows an implementation of a simple auction in SCILLA. Its parameters include the starting block `auctionStart`, a number of blocks `biddingTime` for which it is open for bidding, as well as the address of the `beneficiary`, to which the funds are going to be transferred once the bidding is closed. The mutable fields record the fact whether the auction has `ended`, the latest `highestBidder`, their `highestBid` as well as a mapping of the pending returns, to be reclaimed by bidders who no longer offer the highest bid, but have not yet been reimbursed.

The contract features three transitions. The first one, `Bid` allows anyone to bid for winning in the auction. In case of a higher new bid, the previous `highestBidder` is replaced, simultaneously getting a record in `pendingReturns`, so they could claim their overall bid amount later. The second transition `Withdraw` makes it possible for any previous bidder (who is no longer the highest one) to reclaim the amount of all their previous bids in one transfer. Finally, the transition `AuctionEnd` allows the beneficiary to receive the amount of the highest big, once the auction has finished.

Notice, that even though we encoded this contract in SCILLA, we have *not* formalised and verified any of its properties as we did for `Crowdfunding` in the previous section.[7] The goal of this smart contract programming exercise is, thus, to *formulate* the desired properties and assess their adequacy. We suggest the following temporal properties for the simple auction contract:

[7] *I.e.*, there might be bugs in the code, and we invite the reader to point them out!

```
contract SimpleAuction(
  auctionStart: uint,
  biddingTime: uint,
  beneficiary: address
)
{
  ended: bool = false;
  highestBidder: address = 0;
  highestBid: uint = 0;
  pendingReturns : address ⇒ uint = [ ]
}

(* Transition 1: bidding *)
transition Bid
 (sender : address, value : uint, tag : string)
  if tag == "bid" ⇒
  blk ← && block_number;
  let end = auctionStart + biddingTime in
  let after_end = end + 1 in
  e ← & ended;
  if after_end ≤ blk || e
  then
    send ⟨to → sender, amount → 0,
          tag → "main", msg → "late_to_bid"⟩
  else
    hb ← & highestBid;
    if value ≤ hb
    then
      send ⟨to → sender, amount → 0,
            tag → "main", msg → "bid_too_low"⟩
    else
      hbPrev ← & highestBidder;
      prs ← & pendingReturns;
      let b = contains(prs, hbPrev) in
      let prs1 =
          b ?
          let pr = get(prs, hbPrev) in
          let hs1 = pr + highestBid in
              put(prs, hbPrev, hs1) :
          put(prs, hbPrev, highestBid) in
       pendingReturns := prs1;
       highestBidder := sender;
       highestBid := value;
       send ⟨to → sender, amount → 0,
             tag → "main",
             msg → "bid_accepted"⟩

(* Transition 2: claiming money back *)
transition Withdraw
  (sender : address,
   value : uint,
   tag : string)
  if tag == "withdraw" ⇒
  prs ← & pendingReturns;
  let b = contains(prs, hbsender) in
  if b
  then
    let pr = get(prs, sender) in
    let prs1 = remove(prs, sender) in
    pendingReturns := prs1;
    send ⟨to → sender, amount → pr,
          tag → "main", msg → "here_is_your_money"⟩
   else
     send ⟨to → sender, amount → 0, tag → "main",
           msg → "nothing_to_withdraw"⟩

(* Transition 3: auction ends *)
transition AuctionEnd
  (sender : address,
   value : uint,
   tag : string)
  if tag == "end" ⇒
  blk ← && block_number;
  e ← & ended;
  let t1 = auctionStart + biddingTime in
  let t2 = blk ≤ t1 in
  let t3 = not e in
  let t4 = t2 || t3 in
  if t4
  then
    send ⟨to → sender, amount → 0,
          tag → "main",
          msg → "auction_is_still_going_on"⟩
  else
    ended := true;
    hb ← & highestBid;
    send ⟨to → beneficiary, amount → hb,
          tag → "main",
          msg → "here_is_the_highest_bid"⟩
```

Figure 5. An Auction contract in Scilla.

P1. The balance of `SimpleAuction` should be greater or equal than the sum of the `highestBid` and values of all entries in `pendingReturns`.

P2. For any account a, the value of the corresponding entry in `pendingReturns` (if present) should be equal to the sum of values of all transfers a has made during its interaction with the contract.

P3. An account a, which is not the higher bidder, should be able to retrieve the full amount of their bids from the contract, and do it *exactly* once.

Together, a combination of these properties ensure that the contract is not "prodigal" funds, *i.e.*, does not dispense them to the parties who have no right to claim them, neither that it is "greedy", *i.e.*, it does not lock funds forever, so they can be always retrieved [19].

```
contract RockPaperScissors (
  player1: address,
  player2: address,
  owner: address
)
{
 p1Choice : string = "";
 p2Choice : string = "";
 payoffMatrix :
   string ⇒ string ⇒ uint = [
   "rock"   → ["rock" → 0; "paper" → 2;
              "scissors" → 1];
   "paper" → ["rock" → 1; "paper" → 0;
              "scissors" → 2];
   "scissors" → ["rock" → 2;
                 "paper" → 1;
                 "scissors" → 0];
 ]
}

transition choicePlayer1 (
    sender: address,
    value: uint,
    tag: string,
    choice: string)
 if let b1 = tag == "pp1" in
    let b2 = sender == player1 in
    b1 && b2 ⇒
 pc ← & p1Choice;
 pm ← & payoffMatrix;
 if pc == "" && contains(pm, pc)
 then
  p1Choice := choice;
  send ⟨to → sender, amount → 0,
        tag → "main", msg → "true"⟩
 else
  send ⟨to → sender, amount → 0,
        tag → "main", msg → "false"⟩
(* choicePlayer2 is similar  *)

transition determineWinner (
    sender: address,
    value: uint,
    tag: string)
  if tag == "detWinner" ⇒
  pm ← & payoffMatrix;
  pc1 ← & p1Choice;
  pc2 ← & p2Choice;
  if not ((pc1 == "") || (pc2 == ""))
  then
    let p1cm = get(pm, pc1) in
    let winner = get(p1cm, pc2) in
    bal ← && balance;
    if winner == 1
    then
      send ⟨sender → player1, amount → bal,
            tag → "main", msg → "Congrats, P1"⟩
    else
      if winner == 2
      then
        send ⟨sender → player2, amount → bal,
              tag → "main", msg → "Congrats, P2"⟩
      else
        send ⟨sender → owner, amount → bal,
              tag → "main", msg → "Congrats, Owner"⟩
   else
     send ⟨to → sender, amount → 0,
           tag → "main", msg → "Not determined"⟩
```

Figure 6. A simplistic Rock-Paper-Scissors contract in SCILLA.

5.2 Properties of Multi-Party Games

The last contract we consider implements a version of the Rock-Paper-Scissors game and is adapted from the experience report by Delmolino *et al.* [11]. In this implementation, we intentionally do not address a known vulnerability allowing one of the parties to cheat, once they see a result submitted by the competition. The contract implementation is parameterised with identities of `player1` and `player2`, as well as the contract's `owner`. The `payoffMatrix` encodes the outcome of the game depending on the results submitted by both `player1` and `player2`, allowing to unambiguously determine the winner. The transition `choicePlayer1` allows Player 1 to submit their value; `choicePlayer2` is similar and is, therefore, omitted. The transition `determineWinner` can be invoked by anyone and determines a winner based on the payoff matrix with a twist: if the players submitted equal values, the award goes to the contract's owner.

What can we specify about this game? We suggest the following properties:

P1. No other party besides `player1`, `player2`, or `owner` can be awarded the prize, which is equal to the contract's balance remaining constant before then.

P2. Each player can only submit their non-trivial choice once, and this choice will have to be a key from `payoffMatrix` in order to be recorded in the corresponding contract field.

As noticed before, we cannot express a property that would prevent either player from cheating, given that the values of the fields are public, since this property would not hold for this implementation. However, we can envision that in a fixed version of the contract [11], one can state it using a knowledge argument over the prefix of an execution history observed so far [13].

6 Related Work

Temporal reasoning about smart contracts has not received much attention to date, but we expect it some to become a popular research direction in the formal methods community. Our proposal on Scilla [23] was amongst the first one to emphasize the state transition system-like nature of smart contract in order to facilitate reasoning about their behaviours, safety and temporal properties. Other programming language proposals along the same lines of thinking are Bamboo [2] and Obsidian [3]. That said, none of those languages has been used to provide a framework for formal reasoning about contract executions.

The recently presented tool FSolidM [18] proposes a high-level modelling framework for smart contracts based on state automata, targeting verification of automata properties at the level of a model, rather than executable code.

The importance of being able to detect smart vulnerabilities, arising in from violating safety and trace properties, has been realised in the blockchain community, and several automated tools have been recently released to tackle this challenge. Amongst the most related to the ideas we discussed here, the tool by Grossman *et al.* [12] implements a dynamic analysis of execution traces of smart contracts with the goal to detect DAO-like vulnerabilities [10], manifested by ill-formed reentrancy patterns [24]. Zeus by Kalra *et al.* [14] checks contract source for user-defined safety properties; it does not address temporal properties, though. The closest to our proposal is Maian by Nikolic *et al.* [19]. The tool provides a static analysis for detecting bugs, violating certain trace properties, which are expressed as instance of our predicate `since_as_long` (*cf.* Section 3.3) for specific precondition `p`, side-condition `r`, and a postcondition `q`.

7 Conclusion

In this position paper we outlined some new avenues for applications of formal methods for reasoning about *temporal properties* of smart contracts. We presented a verification framework, based on the Scilla smart contract programming language, and sketched a number of critical properties for commonly used smart contracts. We believe that our observations will stimulate research, and allow effective reuse of existing results, tools, and insights for formally specifying and verifying applications built on top of a distributed ledger .

References

1. BlockKing contract, 2016. Source code available at https://etherscan.io/address/0x3ad14db4e5a658d8d20f8836deabe9d5286f79e1.

2. Bamboo, 2017. https://github.com/pirapira/bamboo.
3. Obsidian, 2018. https://mcoblenz.github.io/Obsidian.
4. J. Alois. Ethereum Parity Hack May Impact ETH 500,000 or $146 Million, 2017. https://www.crowdfundinsider.com/2017/11/124200-ethereum-parity-hack-may-impact-eth-500000-146-million/.
5. N. Atzei, M. Bartoletti, and T. Cimoli. A Survey of Attacks on Ethereum Smart Contracts (SoK). In *POST*, volume 10204 of *LNCS*, pages 164–186. Springer, 2017.
6. K. Bansal, E. Koskinen, and O. Tripp. Automatic generation of precise and useful commutativity conditions. In *TACAS (Part I)*, volume 10805 of *LNCS*, pages 115–132. Springer, 2018.
7. T. Chen, X. Li, X. Luo, and X. Zhang. Under-optimized smart contracts devour your money. In *SANER*, pages 442–446. IEEE, 2017.
8. Coq Development Team. *The Coq Proof Assistant Reference Manual - Version 8.8*, 2018. http://coq.inria.fr/.
9. T. Coquand and G. P. Huet. The calculus of constructions. *Information and Computation*, 76(2/3):95–120, 1988.
10. M. del Castillo. The dao attack, 2016. 16 June 2016.
11. K. Delmolino, M. Arnett, A. E. Kosba, A. Miller, and E. Shi. Step by step towards creating a safe smart contract: Lessons and insights from a cryptocurrency lab. In *FC 2016 Workshops*, volume 9604 of *LNCS*, pages 79–94. Springer, 2016.
12. S. Grossman, I. Abraham, G. Golan-Gueta, Y. Michalevsky, N. Rinetzky, M. Sagiv, and Y. Zohar. Online detection of effectively callback free objects with applications to smart contracts. *PACMPL*, 2(POPL):48:1–48:28, 2018.
13. J. Y. Halpern and Y. Moses. Knowledge and common knowledge in a distributed environment. *J. ACM*, 37(3):549–587, 1990.
14. S. Kalra, S. Goel, M. Dhawan, and S. Sharma. Zeus: Analyzing safety of smart contracts. In *NDSS*, 2018.
15. L. Lamport. "Sometime" is Sometimes "Not Never" - On the Temporal Logic of Programs. In *POPL*, pages 174–185. ACM Press, 1980.
16. L. Lamport. The part-time parliament. *ACM TOPLAS*, 16(2):133–169, 1998.
17. X. Leroy. Formal certification of a compiler back-end or: programming a compiler with a proof assistant. In *POPL*, pages 42–54. ACM, 2006.
18. A. Mavridou and A. Laszka. Tool Demonstration: FSolidM for Designing Secure Ethereum Smart Contracts. In *POST*, volume 10804 of *LNCS*, pages 270–277. Springer, 2018.
19. I. Nikolic, A. Kolluri, I. Sergey, P. Saxena, and A. Hobor. Finding The Greedy, Prodigal, and Suicidal Contracts at Scale. *CoRR*, abs/1802.06038, 2018.
20. S. L. Peyton Jones. *The Implementation of Functional Programming Languages*. Prentice-Hall, 1987.
21. A. Pnueli. The temporal logic of programs. In *FOCS*, pages 46–57. IEEE Computer Society, 1977.
22. I. Sergey and A. Hobor. A Concurrent Perspective on Smart Contracts. In *1st Workshop on Trusted Smart Contracts*, 2017.
23. I. Sergey, A. Kumar, and A. Hobor. SCILLA: a Smart Contract Intermediate-Level LAnguage, 2018. https://arxiv.org/abs/1801.00687.
24. E. G. Sirer. Reentrancy Woes in Smart Contracts, 2016. 13 July 2016.
25. Solidity: A contract-oriented, high-level language for implementing smart contracts, 2018.
26. G. Wood. Ethereum: A Secure Decentralised Generalised Transaction Ledger, 2014. https://ethereum.github.io/yellowpaper/paper.pdf.