Communication Directed Locality for Compositional Verification of Smart Contracts

Scott Wesley[1][0000−0002−6708−2122], Maria Christakis[2], Jorge A. Navas[3], Richard Trefler[1], Valentin Wüstholz[4], and Arie Gurfinkel[1]

[1] University of Waterloo, Canada
[2] MPI-SWS, Germany
[3] SRI International, USA
[4] ConsenSys, Germany

Abstract. Smart contracts are programs that manage cryptocurrency accounts on a blockchain. Smart contracts offer exciting new challenges to the formal-methods community, due to their novel execution environment and problem domain. In this paper, we present an automated and scalable technique to obtain compositional proofs for global safety properties over all sets of k users (k-*universal safety*). First, we present a parameterized model for smart contract networks in which users are *user processes* that manipulate shared memory through a *control process*. We reframe this communication as message passing, and implement a static analysis algorithm to overapproximate the possible communications. Using this overapproximation, we then present a *local model* which can simulate networks of arbitrary size. Using this local model, we then introduce a compositional model checking rule that extends the applicability of parameterized compositional model checking to real-world systems. To evaluate our technique, we compare against state-of-the-art smart contract verification tools and show that ref our numbers once added in.

1 Introduction

Smart contracts were proposed in 1996 [24], as reactive state machines that managed legal assets. The first large-scale smart contract system was realized in 2008 with the creation of the Bitcoin peer-to-peer blockchain protocol [15]. Bitcoin has since inspired other financial smart contract systems, such as Ethereum, Algorand [6], Zilliqa [22], and Facebook Libra [3]. Ethereum has seen widespread acceptance, resulting in a capitalization of 1 billion dollars (UDS) within its first 2 years of existence [2]. Ethereum is an open, decentralized compute framework which allows developers to deploy Turing-complete smart contracts [26]. This framework is backed by a cryptocurrency to incentivize cooperation. As a result, many attackers have taken an interest in Ethereum, and have managed to compromise up to 60 million dollars (USD) in a single attack [2]. Evidently, the verification of Ethereum smart contracts is of the upmost importance to both developers and investors.

Smart contract and blockchain technologies present new and exciting research challenges for the formal methods community. Aside from testing (e.g., [10,27])

```
contract Auction {
    address payable owner;
    address highestBidder;
    uint endTime;
    bool isClosed;

    mapping(address => uint) bids;

    constructor(uint duration) public {
        owner = msg.sender;
        endTime = block.timestamp + duration;
    }

    function bid() public payable {
        require(block.timestamp < endTime);
        require(msg.sender != owner);

        uint maxBid = bids[highestBidder];
        uint newBid = bids[msg.sender] + msg.value;
        require(maxBid < newBind);

        bids[msg.sender] = newBid;
        highestBidder = msg.sender;
    }

    function withdraw() public {
        require(block.timestamp >= endTime);
        require(msg.sender != highestBidder);
        uint bid = bids[msg.sender];
        if (bid > 0) {
            bids[msg.sender] = 0;
            msg.sender.transfer(bid);
        }
    }

    function close() public {
        require(block.timestamp >= endTime);
        require(msg.sender == owner);
        require(isClosed);

        uint highestBid = bids[highestBidder];
        owner.transfer(highestBid);
        isClosed = false;
    }
}
```

Fig. 1: An example smart contract which implements an open-bid auction. Make this look nicer.

and static analysis (e.g. [4]), many approaches focus on semi- (e.g., [25,9]) and fully-automated (e.g., [11,19,23]) verification. A more limited selection of research has focused on the distributed aspects of smart contracts (i.e., [13,20,12]). The majority of smart contract research has focused on programs written in Solidity, one of the most popular languages for Ethereum smart contract development.

Despite current research trends, smart contracts are parameterized and distributed in nature. For example, the `Auction` smart contract in Fig. 1 allows for an arbitrary number of users to `bid()` until an `endTime` is reached. Due to limitations in Solidity (i.e., gas constraints) the `Auction` must recall the `highestBidder` at any point of execution. For this reason, `Auction` must ensure, among other requirements, Prop. 1: Every non-zero bid is of a distinct amount. Prop. 1 must hold for any pair of users, irrespective of the number of users interacting with `Auction`. This motivates local reasoning for smart contracts.

What we do.

Contributions

2 Background

In this section, we briefly recall Parameterized Compositional Model Checking (PCMC) [16].

Notation. We write $\mathbf{u} = \langle u_0, \ldots u_{n-1} \rangle$ for a vector of n elements, and $\mathbf{u}_i$ for the ith element of $\mathbf{u}$. For a natural number $n \in \mathbb{N}$, we write $[n]$ for $\{0, \ldots, n-1\}$.

Parmeterized Labeled Transition System. A *labeled transition* system P, is a tuple (S, A, T, s_0), where S is a set of states, A is a set of actions, $T : S \times A \to 2^S$ is a transition function, and $s_0 \in S$ is an initial state. P is *deterministic* if T is a function, $T : S \times A \to S$. A (finite) *trace* of P is an alternating sequence of states and actions, $(s_0, a_1, s_1, \ldots, a_k, s_k)$, such that $\forall i \in [k] \cdot s_{i+1} \in T(s_i, a_{i+1})$. A state s_i is *reachable* in P if it appears in some trace $(s_0, a_1, \ldots, s_k)$ of P. A *safety property* for P is a subset of states (or a predicate[5]) $\varphi \subseteq S$. P satisfies φ, written $P \models \varphi$, if every state s reachable in P is in φ.

Many transition systems are parameterized. For instance, a client-server application is parameterized in the number of clients, while an array manipulating program is parameterized in the number of array cells. In both of these examples, there is a single *control process* that interacts with a number of *user processes*. In [16], such systems are called synchronized control-user networks (SCUNs). By convention, we use N to denote the number of processes, and use the set $[N]$ to denote process identifiers. In this paper, we consider a special case of SCUNs, in which users only synchrhonize with the control process and are not executing any code on their own.

An SCUN [16] $\mathcal{N}$ is a tuple (S, A, T, c_0, u_0), where $S = S_C \cup S_U$, S_C is a set of control states, S_U a set of user states, $A = A_C \cup A_S$, A_C a set of control actions, A_S a set of synchronized actions, $T = T_I \cup T_S$, $T_I : S_C \times A_C \to S_C$ an internal transition relation, $T_S : S_C \times S_U \times A_S \to S_C \times S_U$ a synchronized transition relation, $c_0 \in S_C$ and $u_0 \in S_U$ are the initial control and user states, respectively. The semantics of $\mathcal{N}$ is given by a parameterized LTS, $P(N) = (S, A, T, s_0)$, where $S = S_C \times (S_U)^N$, $A = A_U \cup (A_S \times [N])$, $s_0 = (c_0, u_0, \ldots, u_0)$, and $T : S \times A \to S$ such that: (1) if $a \in A_C$, then $T((c, \mathbf{u}), a) = (c', \mathbf{u})$, where $c' = T_C(c, a)$; (2) if $(a, i) \in A_S \times [N]$, then $T((c, \mathbf{u}), (a, i)) = (c', \mathbf{u}')$ where $(c', \mathbf{u}'_i) = T_C(c, \mathbf{u}_i)$, and $\forall j \in [N] \cdot i \neq j \Rightarrow \mathbf{u}'_j = \mathbf{u}_j$.

Parameterized Compositional Verification. Proving properties of parameterized systems requires properties to be parametrized as well [8,16]. A *k-universal safety property* [8] is a predicate $\varphi \subseteq S_C \times (S_U)^k$. A state $(c, \mathbf{u})$ satisfies φ if $\forall \{i_1, \ldots, i_k\} \subseteq [N] \cdot \varphi(c, \mathbf{u}_{i_1}, \ldots, \mathbf{u}_{i_k})$. We say that a parameterized system $P(N)$ satisfies φ if $\forall N \in \mathbb{N} \cdot P(N) \models \varphi$.

For example, Prop. 1 (Sec. 1) of `Auction` (Fig. 1) is 2-universal: for any pair (c_1, c_2) of distinct users, either one bid is 0 or the bids of c_1 and c_2 are unequal.

Proofs of k-universal safety use compositional reasoning, e.g., [1,8,16,18]. In this paper, we use *Parameterized Compositional Model Checking* (PCMC) [16]. A key to PCMC is a *compositional invariant* – a predicate that summarizes the reachable states of some process, while being closed under the actions of any other process [16]. In the case of a SCUN, the compositional invariant is given by two predicates $\theta_C \subseteq S_C$ and $\theta_U \subseteq S_C \times S_U$ satisfying:

[5] Abusing notation, we refer to a subset of states φ as a *predicate* and do not distinguish between the syntactic form of φ and the set of states that satisfy it.

Initialization $c_0 \in \theta_C$ and $(c_0, u_0) \in \theta_U$;
Consecution 1 If $c \in \theta_C$, $(c, u) \in \theta_U$, $a \in A_S$, and $(c', u') \in T_S((c, u), a)$, then $c' \in \theta_C$ and $(c', u') \in \theta_U$;
Consecution 2 If $c \in \theta_C$, $(c, u) \in \theta_U$, $a \in A_C$, and $c' = T_C(c, a)$, then $c' \in \theta_C$ and $(c', u') \in \theta_U$;
Non-Interference If $c \in \theta_C$, $(c, u) \in \theta_U$, $(c, v) \in \theta_U$, $a \in A_S$, and $(c', u') = T_S((c, u), a)$ then $(c', v) \in \theta_C$.

By PCMC [16], if $\forall u \in \theta_C \cdot \forall\{(s, u_1), \ldots, (s, u_k)\} \subseteq \theta_U \cdot \varphi(s, u_1, \ldots, u_k)$ then $P \models \varphi$. This is as an extension of Owicki-Gries style proofs where θ_C is preserved under a step of the control process, and θ_C is preserved under interference [8,18]. For this reason, we call θ_C the *interference invariant*. SW: Add uniformity.

3 MiniSol: Syntax and Semantics

We work with MiniSol – a subset of the Solidity language that simplifies parameterized reasoning about smart contracts. Like Solidity, MiniSol is an imperative object-oriented language with built-in communication operations. The complete syntax for MiniSol is given in Appendix A. MiniSol is designed to overcome two challenges (a) complex semantics, and (b) complex language features. Restricting MiniSol to a core subset of Solidity significantly simplifies the formal semantics of MiniSol. Reducing Solidity features outside the scope of our analysis (e.g., inheritance and cryptographic operations) simplifies the presentation. We illustrate our presentation using `Auction` in Fig. 1.

A MiniSol *smart contract* is a process on the Ethereum virtual machine. Each smart contract exposes transactions for users to execute. A transaction is a deterministic sequence of operations. Each user in the network is allocated a unique identifier, known as an *address*, and maintains a *balance* to track the cryptocurrency (called *Ether*) it owns. We view a smart contract as a control process, and each user as a user process of a synchronized control-user network [16]. As will be shown in the semantics, the user process is the smart contract state that is owned by the respective user, whereas the control process is the shared state. We view transactions as ordered sequences of internal (to update control state) and synchronized (to retrieve user data) actions. Note that in MiniSol, the terms *client* and *user* are not interchangeable. A *user* is any user process within the control-user network. A *client* is defined relative to a transaction f, and refers to a user that is given as an input to f.

MiniSol represents transactions using functions. Each function takes zero or more inputs. A contract declaration is a collection of variables and functions (similar to a class in object-oriented programming). A special transaction is the constructor that is executed before any other transaction (similar to a class constructor). The `Auction` in Fig. 1 is a contract. At line 9, there is a constructor that takes a single argument. The functions at lines 14, 26, and 36 take no arguments. Lines 2 to 7 declare state variables of various types.

MiniSol supports three types: *primitive* (e.g., `int256`), *mapping* (e.g., `mapping`), *contract reference* (e.g, `Auction`). Primitive type is further partitioned into *ad-*

dress (i.e., `address`) and *non-address*. Each typed variable is further classified as either *state*, *input*, or *local*. An address and non-address typed state variables are called *role* and *datum*, respectively. A mapping typed state variable is called a *record*. An address and non-address typed inputs are called *client* and *argument*, respectively. A contact reference typed variable is called *tightly-coupled* if it is assigned only in the constructor. A MiniSol program is *tightly-coupled* whenever all of it contract reference typed variables are tightly-coupled. In `Auction` there are: 2 roles – `owner` and `highestBidder`, 2 contract data – `endTime` and `isClosed`, 1 record – `bids`, 1 user – `msg.sender` common across all transactions, at most 2 arguments in any transaction – `block.timestamp` and `msg.value`, or `block.timestamp` and `duration`. Note that `msg.sender`, `msg.value`, and `block.timestamp` are implicit arguments as defined by the semantics.

Semantics of MiniSol. Let $\mathcal{C}$ be a MiniSol program. An N-user *bundle* is an N-user network of several (possibly identical) MiniSol programs [19]. We require that each MiniSol program is tightly coupled (in particular, no dynamic dispatch). The transactions of a bundle are determined by the number of users in the network. Intuitively, users alternate and execute transactions sequentially. Each transaction is atomic, that is, it is applied all at once. For simplicity, we assume there is a single transaction.

The semantics of an N-user MiniSol bundle, is a transition system $\mathsf{lts}(\mathcal{C}, N) = (S, P, f, s_0)$, where $S \subseteq S_C \times (S_U)^N$ is the set of states, $\mathsf{control}(\mathcal{C}) = S_C$ is the set of control states, $\mathsf{user}(\mathcal{C}) = S_U$ is the set of user states, $\mathsf{config}(\mathcal{C}, N) = (S_U)^N$ is the set of *N-user configurations*, P is the set of *transaction inputs* (labels), $f : S \times P \to S$ is the *transaction function*, and s_0 is the initial state. We assume, without loss of generality, that there is a single control process[6]. We let $\mathbb{A}$ denote the set of address values, and $\mathbb{D}$ denote the set of non-address values. Both $\mathbb{A}$ and $\mathbb{D}$ are finite by definition. For simplicity of presentation, we assume that each record maps from $\mathbb{A}$ to $\mathbb{D}$, and that the implicit balance of each user is a record. AG: We should stay away from notation that differs by font only. We should not have A, $\mathbb{A}$, and $\mathcal{A}$ stand for significantly different things. This is where using macros helps to fine tune the notation later.

The states of $\mathcal{C}$ are determined by its state variables. Let n, m, and k be the numbers of roles, data, and records, respectively. Then, the control and user states of $\mathcal{C}$ are $S_C \subseteq \mathbb{A}^n \times \mathbb{D}^m$ and $S_U \subseteq \mathbb{A} \times \mathbb{D}^k$, respectively. For $c = (\mathbf{x}, \mathbf{y}) \in S_C$, $\mathsf{role}(c, i) = \mathbf{x}_i$ is the i-th role of $\mathcal{C}$ and $\mathsf{data}(s, i) = \mathbf{y}_i$ is the i-th datum of $\mathcal{C}$. For $u = (x, \mathbf{y}) \in S_C$, $\mathsf{id}(u) = x$ is the address of u and $\mathsf{rec}(u) = \mathbf{y}$ is the records of u.

The transactions of $\mathcal{C}$ are determined by its functions. Let q and r be the maximum number of clients and arguments, respectively. Then, the inputs of $\mathcal{C}$ are $\mathsf{inputs}(\mathcal{C}) = P \subseteq \mathbb{A}^q \times \mathbb{D}^r$. For $p = (\mathbf{x}, \mathbf{y}) \in P$, $\mathsf{client}(p, i) = \mathbf{x}_i$ is the i-th client in p and $\mathsf{arg}(p, i) = \mathbf{y}_i$ is the i-th argument in p. We assume standard imperative semantics to the body of each MiniSol function. This semantic interpretation is given by $f = [\![\mathcal{C}]\!]_N$. By convention, we write $f_p(s, \mathbf{u}) = f(s, \mathbf{u}, p)$ and say that f_p is an instance of f *bound* to p.

[6] All smart contracts are tightly coupled, so multiple contracts can be combined.

For any $\mathcal{A} = \{a_0, \ldots, a_{N-1}\} \in \mathbb{A}$, the state $\mathsf{init}(\mathcal{C}, \mathcal{A}) = (c, \mathbf{u}) \in S$ is *zero-initialized state* if $c = (\mathbf{0}, \mathbf{0})$ and $\forall i \in [N] \cdot \mathsf{rec}(\mathbf{u}_i) = \mathbf{0}$. To ensure that each user has a unique address, we further require that $\forall i \in [N] \cdot \mathsf{id}(\mathbf{u}_i) = a_i$. The initial state of an N-user bundle is $s_0 = \mathsf{init}(\mathcal{A}, [N])$. That is, s_0 is zero-initialized and uses addresses from 0 to $N - 1$.

For example, let $\mathsf{lts}(\mathcal{C}, 4) = (S, P, f, s_0)$ be the 4-user bundle of `Auction`. Consider a state $(c, \mathbf{u}) \in S$ and an input $p \in P$ such that the `highestBidder` is `address(2)`, the `endTime` is 100, the bid of the user at `address(2)` is 5, and the next transaction is `bid()` executed by the user at `address(1)` with a `msg.value` of 10. As `highestBidder` is the second address state variable, it is the second role, and therefore $\mathsf{role}(s, 1) =$ `address(2)`. Similarly, as `endTime` is the first non-address state variable, it is the first datum, and therefore $\mathsf{data}(s, 0) = 100$. To inspect the user's balance, we recall that users are ordered from 0 to 3, and that the first record is reserved for balances. Therefore, the bid of the user at `address(2)` is $(\mathsf{rec}(\mathbf{u}_2))_1 = 5$. As for the input, $\mathsf{client}(p, 0) =$ `address(1)` and $\mathsf{arg}(p, 0) = 10$ for similar reasons. We give the full LTS in Appendix B.

Note, however, that MiniSol abstracts away or does not support many important features of Solidity. Some of these features are syntactic sugar (i.e., inheritance, enums, structs, bounded arrays) and are supported by our implementation. Other features are non-trivial but outside of the scope of local reasoning (i.e., cryptographic and hasing primitives, bit-precise arithmetic, gas constraints, inline assembly). Some features (i.e., dynamic dispatch to other contracts, mappings to address variables, unbounded arrays, address inequalities) are not supported by the local reasoning techniques in this paper, and are open challenges.

4 Characterizing Communication

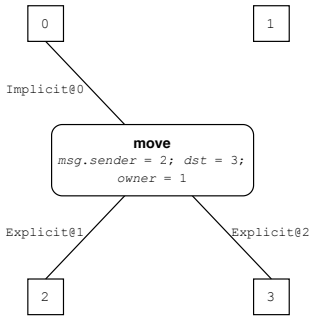
The core functionality of any smart contract is communication between users. Usually, the users communicate by reading from and writing to designated mapping entries. That is, the communication paradigm is that of shared memory. However, it is convenient for analysis to re-imagine this communication as rendezvous synchronization in which users explicitly participate in message passing. In this section, we formally re-frame smart contracts with explicit communication by defining a (semantic) participation topology and its abstractions.

A user c *influences* a transaction f if a change in the state of c affects the outcome of f. Likewise, a transaction f influences a user c if f can change the state of c. For example, in Fig. 2a, the `msg.sender` influences `move` at line 7. Similarly, `move` influences `msg.sender` at line 8, and `dst` at line 9. In all these cases, the influence is *witnessed* by the state of the contract and the configuration of users that exhibit the influence.

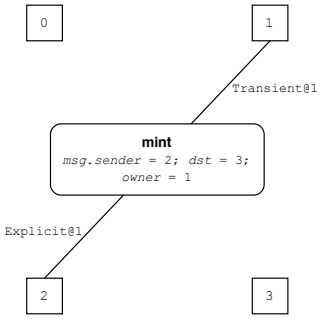
SW: Perhaps I've been looking at this too long but I'm convinced these definitions are backwards (they seem to have always been that way). I've swapped them. I'm leaving this here in case anyone objects.

```
contract SimpleToken {
    address owner;
    mapping (address => int) funds;

    function move(address dst) public {
        require (dst != address(0));
        require (funds[msg.sender] > 0);
        require (funds[dst] + 1 > funds[
            dst]);
        funds[msg.sender] = 1;
        funds[dst] += 1;
    }

    function mint(address dst) public {
        require (msg.sender == owner);
        funds[dst] += 1;
    }
}
```

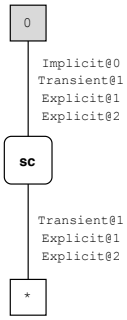
(a) Contract source text.



(b) A participation topology for `move`.



(c) A participation topology for `mint`.



(d) A full participation graph.

Fig. 2: The participation topology of a smart contract, contrasted with its participation topology graph, as generated by `PTGBuilder`.

Throughout the rest of this section, Let $\mathcal{C}$ be a contract, $N \in \mathbb{N}$ be the network size, $\langle S, P, f, s_0 \rangle = \mathsf{lts}(\mathcal{C}, N)$, and $p \in P$. A user at address $x \in \mathbb{A}$ *influences* transaction f_p if there exists an $s \in \mathsf{control}(\mathcal{C})$, $\mathbf{u}, \mathbf{v} \in \mathsf{config}(\mathcal{C}, N)$, and $i \in [N]$ such that, (1) $\mathsf{id}(\mathbf{u}_i) = x$, (2) $\forall j \in [N] \cdot \mathbf{u}_j = \mathbf{v}_j \iff i \neq j$, (3) $(\langle s'_u, \mathbf{u}' \rangle = f_p(s, \mathbf{u}) \wedge \langle s'_v, \mathbf{v}' \rangle = f_p(s, \mathbf{v})) \Rightarrow \left(s'_u \neq s'_v \vee \exists j \in [N] \setminus \{i\} \cdot \mathbf{u}'_j \neq \mathbf{v}'_j\right)$ The tuple $\langle s, \mathbf{u}, \mathbf{v} \rangle$ is a *witness* to the influence of f_p over user x.

A user at index $x \in \mathbb{A}$ is *influenced* by transaction f_p if there exists an $s \in \mathsf{control}(\mathcal{C})$, $\mathbf{u} \in \mathsf{config}(\mathcal{C}, N)$, and $i \in [N]$ such that $\langle s', \mathbf{u}' \rangle = f_p(s, \mathbf{u})$ with $\mathsf{id}(\mathbf{u}_i) = x$ and $\mathbf{u}'_i \neq \mathbf{u}_i$. The tuple $\langle s, \mathbf{u} \rangle$ is called a *witness* to the influence of user x over f_p.

These influences form the notion of explicit participation:

Definition 1 (Participation). *A user at address $x \in \mathbb{A}$ participates in transaction f_p whenever either x influences f_p (witnessed by some $\langle s, \mathbf{u}, \mathbf{v} \rangle$) or f_p influences x (witnessed by some $\langle s, \mathbf{u} \rangle$). In both cases, s is a* witness state *to the participation of x in f_p.*

Often a smart contract facilitates communication between multiple users over multiple transactions. We need to know the set of all possible participants, and the cause of the participation — we call this the *participation topology (PT)*. A PT associates each act of message passing (sending or receiving) with one or more participation classes, called *explicit*, *transient*, and *implicit*. The participation is *explicit* whenever the participant is a client of the transaction. It is *transient* if the participant is assigned to at least one role during the transaction. Finally, it is *implicit* whenever there is at least one state in which the participant is neither a client nor holds any roles. An example of an implicit participation is a reference to a constant address, such as `address(0)` in line 6 of Fig. 2a.

Definition 2 (Participation Topology). *A participation topology of a transaction f_p is a tuple $Topology(\mathcal{C}, N, p) := \langle Explicit, Transient, Implicit \rangle$, where*

1. *$Explicit \subseteq \mathbb{N} \times \mathbb{A}$ s.t. $Explicit(i, x)$ is true iff x participates during f_p, with $\mathsf{client}(p, i) = x$.*
2. *$Transient \subseteq \mathbb{N} \times \mathbb{A}$ s.t. $Transient(i, x)$ is true iff x participates during f_p, as witnessed by some state $s \in \mathcal{S}$, where $\mathsf{role}(s, i) = x$.*
3. *$Implicit \subseteq \mathbb{A}$ s.t. $Implicit(x)$ is true iff x participates during f_p, as witnessed by some state $s \in \mathsf{control}(\mathcal{C})$, where $\forall i \in \mathbb{N}$, $\mathsf{role}(s, i) \neq x$ and $\mathsf{client}(p, i) \neq x$.*

For example, Fig. 2b and Fig. 2c, show PT for `move` and `mint` using four user, an identical input set, and an identical pre-state. Note that in Fig. 2c the user at `address(3)` is not a client, as line 15 is not reachable.

Definition 2 is semantic. Next, we define a *Participation Topology Graph* (TPG) that is a syntactic over-approximation of the PT of a transaction independent of the network size. A PTG has a vertex for each user and each transaction such that edges between vertices represent different classes of participation.

We call each entry in a PT a *participation class*. Note that for a contract $\mathcal{C}$, the number of participation classes is finite. There are m explicit classes, n transient classes, and at most $|\mathbb{A}|$ implicit classes, where n is the number of roles, and m is the maximum number of clients taken by any transaction in $\mathcal{C}$. Thus, the label set for a PTG of $\mathcal{C}$ is $\mathrm{AP}(\mathcal{C}) := \{explicit@i \mid i \in [n]\} \cup \{transient@i \mid i \in [m]\} \cup \{implicit@x \mid x \in \mathbb{A}\}$.

Definition 3 (Participation Topology Graph). *A* participation topology graph *for a contract $\mathcal{C}$ is a tuple $\langle G, \rho, \tau \rangle$, where $G = (E, V, L)$ is a graph labeled by $L \subseteq E \times AP(\mathcal{C})$, $\rho \subseteq \mathsf{inputs}(\mathcal{C}) \times V$, and $\tau \subseteq \mathsf{inputs}(\mathcal{C}) \times \mathbb{A} \times V$, such that for every $Topology(\mathcal{C}, N, p) = \langle Explicit, Transient, Implicit \rangle$,*

1. *If $Explicit(i, x)$ then there exists some $\langle p, u \rangle \in \rho$ and $\langle p, x, v \rangle \in \tau$ such that $\langle u, v \rangle \in E$ and $explicit@i \in L(u, v)$.*
2. *If $Transient(i, x)$ then there exists some $\langle p, u \rangle \in \rho$ and $\langle p, x, v \rangle \in \tau$ such that $\langle u, v \rangle \in E$ and $trasient@i \in L(u, v)$.*
3. *If $Implict(x)$ then there exists some $\langle p, u \rangle \in \rho$ and $\langle p, x, v \rangle \in \tau$ such that $\langle u, v \rangle \in E$ and $implicit@x \in L(u, v)$.*

Theorem 1. *Let $\mathcal{C}$ be a contract with a PTG $\langle G, \rho, \tau\rangle$ with $G = (V, E, L)$. Then, for all $N \in \mathbb{N}$, and for all $p \in \mathsf{inputs}(\mathcal{C})$, Topology$(\mathcal{C}, N, p) = \langle$Explicit, Transient, Implicit$\rangle$ is over-approximated by $\langle G, \rho, \tau\rangle$ as follows:*

1. *If Explicit(i, x) then $\exists\langle u, v\rangle \in E \cdot \rho(p, u) \wedge \tau(p, x, v) \wedge$ explicit@$i \in L(u, v)$;*
2. *If Transient(i, x) then $\exists\langle u, v\rangle \in E \cdot \rho(p, u) \wedge \tau(p, x, v) \wedge$ transient@$i \in L(u, v)$;*
3. *If Implicit(x) then $\exists\langle u, v\rangle \in E \cdot \rho(p, u) \wedge \tau(p, x, v) \wedge$ implicit@$x \in L(u, v)$.*

For any PT, there are many potential PTGs. The weakest PTG joins every user to every transaction using all possible labels. In Fig. 2d, we present a simple, yet stronger, PTG for Fig. 2a. First, note that there is a single implicit participant, identified by `address(0)`. Next, observe that any arbitrary user could take on the role of `owner`. Finally, the differences in participation patterns between `move` and `mint` are ignored. Thus, there are two user vertices, one which is mapped uniquely to `address(0)`, and another – mapped to by all other user. A PTG as in this example are constructed automatically using an algorithm `PTGBuilder`.

`PTGBuilder` takes a contract $\mathcal{C}$ and returns a PTG. It works similarly to taint analysis. Input and state address variables, literal addresses, and each address cast are tainted sources. A memory write, a comparison expression, and each mapping access are sinks. `PTGBuilder` returns $\langle Args, Roles, Lits, Ops\rangle$, where (1) *Args* is the set of indices of input variables that propagate to a sink; (2) *Roles* is the set of indices of state variables that propagate to a sink; (3) *Lits* is the set of literal addresses that propagate a sink; (4) *Ops* is the set of address cast expressions that propagate to a sink.

Finally, a PTG is constructed as $\langle G, \rho, \tau\rangle$, where $G = (V, E, L)$, $\rho \subseteq \mathsf{inputs}(\mathcal{C}) \times V$, and $\tau \subseteq \mathsf{inputs}(\mathcal{C}) \times \mathbb{A} \times V$ s.t. (1) If $Ops = \varnothing$, then $V := \{sc; \star\} \cup Lits$, else $V := \{sc\} \cup \mathbb{A}$; (2) $E := \{\langle sc, v\rangle \mid v \in V \backslash \{sc\}\}$; (3) $\forall e \in E$, $\forall i \in \mathbb{N}$, *explicit*@$i \in L(e) \iff i \in Args$; (4) $\forall e \in E$, $\forall i \in \mathbb{N}$, *transient*@$i \in L(e) \iff i \in Roles$; (5) $\forall e = \langle sc, v\rangle \in E$, $\forall x \in \mathbb{A}$, *explicit*@$x \in L(e) \iff v = x$; (6) $\rho := \{\langle p, sc\rangle \mid p \in \mathsf{inputs}(\mathcal{C})\}$; (7) If $Ops = \varnothing$, then $\tau := \{\langle p, x, \star\rangle \mid p \in \mathsf{inputs}(\mathcal{C}), x \in \mathbb{A} \backslash Lits\} \cup \{\langle p, x, x\rangle \mid p \in \mathsf{inputs}(\mathcal{C}), x \in Lits\}$, else $\tau := \{\langle p, x, x\rangle \mid p \in \mathrm{Inputs}(\mathcal{C}), x \in \mathbb{A}\}$; where sc and $\star$ are unique vertices.

Theorem 2. *Let $\mathcal{C}$ be a contract. If $\langle G, \rho, \tau\rangle$ is the PTGBuilder graph for $\mathcal{C}$, then $\langle G, \rho, \tau\rangle$ is a PTG for $\mathcal{C}$.*

By re-framing smart contracts with rendezvous synchronization, each transaction is re-imagined as a communication of several users. Their communication patterns are captured by the corresponding PT. A PTG over-approximates PTs of all transactions, and is automatically constructed using `PTGBuilder`. This is crucial for local reasoning (Sec. 6) by providing an upper bound on the number of and the classes of the users required for the completeness of local reasoning.

5 Order-Oblivious Communication

A participation topology summarizes all communicating users. However, smart contract communication is not always this simple. For instance, *Election* smart

contract in Solidity tutorial organizers users in a forest-like data structure[7]. This is achieved by explicitly ordering of users by their addresses. In general, users can be ordered explicitly or implicitly. For instance, a linear order can be constructed over the users of a smart contract by comparing their addresses using an inequality operator. In this paper, we restrict ourselves to contracts that do not impose any ordering on the address. This section formalizes this concept as an order-oblivious communication.

Intuitively, a transaction is unordered if the addresses of any two clients or transient users can be swapped without changing its outcome. Of course, this requires a notion of swapping clients. Let $\mathcal{C}$ be a contract with user configurations $\mathbf{u}, \mathbf{u}' \in \mathsf{config}(\mathcal{C}, N)$, and fix $x, y \in \mathbb{A}$. Then $\mathbf{u}'$ is an *address swap* of $\mathbf{u}$ w.r.t. x and y, written $\mathsf{swap}(\mathbf{u}, x, y)$, if $\forall i \in [N]$, (1) $\mathsf{id}(\mathbf{u}_i) = y \Rightarrow \mathsf{id}(\mathbf{u}'_i) = x$, (2) $\mathsf{id}(\mathbf{u}_i) = x \Rightarrow \mathsf{id}(\mathbf{u}'_i) = y$, (3) $\mathsf{id}(\mathbf{u}_i) \notin \{x, y\} \Rightarrow \mathbf{u}'_i = \mathbf{u}_i$, and (4) $\mathsf{rec}(\mathbf{u}'_i) = \mathsf{rec}(\mathbf{u}_i)$. A similar notion is defined for the roles of a control process. Specifically, let $s, s' \in \mathsf{control}(\mathcal{C})$. Then, s' is an address swap of s, if $\forall i \in [N]$, (1) $\mathsf{role}(s, i) = y \Rightarrow \mathsf{role}(s', i) = x$, (2) $\mathsf{role}(s, i) = x \Rightarrow \mathsf{role}(s', i) = y$, and (3) $\mathsf{role}(s, i) \notin \{x, y\} \Rightarrow \mathsf{role}(s', i) = \mathsf{role}(s, i)$. The extension to $\mathsf{inputs}(\mathcal{C})$ and $\mathsf{control}(\mathcal{C}) \times \mathsf{config}(\mathcal{C}, N)$ is trivial.

Definition 4 (Order-Oblivious Transaction). *Let $\mathcal{C}$ be a contract, $N \in \mathbb{N}$ be the network size, $\langle S, P, f, s_0 \rangle = \mathit{lts}(\mathcal{C}, N)$, $p \in P$, and Topology$(\mathcal{C}, N, p) = \langle$Explicit, Transient, Implicit$\rangle$. Then, the transaction f_p is* order-oblivious *if for every pair of states $s, t \in S$ and every $x, y \in \mathbb{A} \setminus$ Implicit s.t. $t = \mathit{swap}(s, x, y)$ and $p' = \mathit{swap}(p, x, y)$, then $f(t, p') = \mathit{swap}(f(s, p), x, y)$.*

The contract $\mathcal{C}$ is *order-oblivious* if every transaction of $\mathcal{C}$ is order-oblivious. It is not hard to see that MiniSol allows only order-oblivious transactions. MiniSol forbids address inequalities and mappings from addresses to addresses. As a result, addresses can only appear in equality, which does not result in an order. This result is important, as it guarantees that users are interchangeable, and consequently, yields a uniform network (as defined in Sec. 2).

AG: I don't think we need this theorem in the paper. It is restating the previous paragraph without adding much. The previous paragraph then provides a conclusion. This whole section is sort-of a remark. It is possible that we can move it into an appendix if needed for space.

Theorem 3. *All MiniSol programs are order-oblivious.*

6 Local Reasoning in Smart Contracts

In this section, we present a proof rule for the parameterized safety of order-oblivious smart contracts. Our proof rule extends the existing theory of PCMC. The section is structured as follows. In Sec. 6.1, we introduce restrictions, on properties and compositional invariants, that expose address dependencies as syntactic patterns. In Sec. 6.2, we define local bundle reductions, which reduce

[7] `https://solidity.readthedocs.io/en/v0.7.3/solidity-by-example.html`

a parameterized smart contract model to a finite-state model. We show that for the correct choice of local bundle reduction, the safety of the corresponding finite model implies the safety of the original parameterized model.

6.1 Properties and Invariants

Many properties and compositional invariants depend on user addresses. However, our local reasoning technique requires that all address dependencies are explicit. To resolve this, we introduce two syntactic forms: the guarded universal safety property and the split compositional invariant. These forms allow address dependencies to be identified through syntactic inspection. We build both forms from predicates that do not depend on user addresses. We call these predicates *address oblivious*.

For any contract $\mathcal{C}$, a predicate $\xi \subseteq \mathsf{control}(\mathcal{C}) \times \mathsf{user}(\mathcal{C})^k$ is address oblivious if it cannot distinguish between lists of *k-address similar* users. Formally, two length-k user lists, $\mathbf{u}$ and $\mathbf{v}$, are k-address similar if $\forall i \in [k] \cdot \mathsf{rec}(\mathbf{u}_i) = \mathsf{rec}(\mathbf{v}_i)$. The predicate ξ is address oblivious if for any choice of $s \in \mathsf{control}(\mathcal{C})$ and any pair of k-address similar lists, $\mathbf{u}$ and $\mathbf{v}$, $\xi(s, \mathbf{u}) \iff \xi(s, \mathbf{v})$. Prop. 2 in Sec. 2 is address oblivious.

A *guarded k-universal safety property* is built from a single k-user address oblivious predicate. The predicate is guarded by constraints over k user addresses. Each constraint compares a single user's address to either a literal address or a user's role. We formalize this in the following definition.

Definition 5 (Guarded Universal Safety). *Let $\mathcal{C}$ be a contract and $k \in \mathbb{N}$. A* guarded k-universal safety property *is a k-universal safety property φ, given by a tuple $\langle L, R, \xi \rangle$, where $L \subseteq \mathbb{A} \times [k]$, $R \subseteq \mathbb{N} \times [k]$, and ξ is an address oblivious k-user predicate, such that*

$$\varphi\,(s, \mathbf{u}) := \psi(s, \mathit{id}(\mathbf{u}_1), \ldots, \mathit{id}(\mathbf{u}_k)) \Rightarrow \xi(s, \mathbf{u})$$

$$\psi(s, \mathbf{x}) := \left[\bigwedge_{\langle a,i\rangle \in L} a = \mathbf{x}_i\right] \wedge \left[\bigwedge_{\langle i,j\rangle \in R} \mathit{role}(s, i) = \mathbf{x}_j\right]$$

Not that $\mathcal{A} = \{a \mid \langle a, i\rangle \in L\}$ defines the literal guards *of φ.*

Definition 5 may appear complex. However, the properties it describes are far less daunting. We illustrate this in Example 1.

Example 1. Consider the claim that in `Auction` of Fig. 2a, the user at `address(0)` is unable to transfer funds. This claim can be phrased, somewhat unintuitively, as Prop. 3: for every user process $\mathbf{u}$, if $\mathsf{id}(\mathbf{u}_0) = \texttt{address(0)}$, then $(\mathsf{rec}(\mathbf{u}_0))_1 = 0$. In this form, Prop. 3 corresponds to the guarded 1-universal safety property φ_1 defined by $\langle L_1, \varnothing, \xi_1 \rangle$, where $L_1 = \{(0, 1)\}$ and $\xi_1(s, \mathbf{u}) := (\mathsf{rec}(\mathbf{u}_0))_1 = 0$. The second set is $\varnothing$ as there are no role constraints in Prop. 3. Following Definition 5, $\varphi_1(s, \mathbf{u}) := (0 = \mathsf{id}(\mathbf{u}_0)) \Rightarrow ((\mathsf{rec}(\mathbf{u}_0))_1 = 0)$. Note that $\mathbf{u}$ is a singleton vector, and that φ_1 has 1 literal guard, given by $\{0\}$. □

The syntax of a *split compositional invariant* is similar to the guarded safety property. We construct the invariant from a list of address oblivious predicates, each guarded by a single constraint. The final predicate is guarded by the negation of all literal address constraints. Intuitively, each clause is meant to summarize the users that satisfy its guard. The split compositional invariant is the conjunction of each of these clauses. We proceed with the formal definition.

Definition 6 (Split Compositional Invariant). *Let $\mathcal{C}$ be a contract. A* split compositional invaraint *is a compositional invariant θ, given by a tuple $\langle L, R, \zeta, \mu, \xi \rangle$, where $L = \{l_1, \ldots, l_m\} \subseteq \mathbb{A}$, $R = \{r_1, \ldots, r_n\} \subseteq \mathbb{N}$, ζ is a list of m address oblivious 1-user predicates, μ is a list of n address oblivious 1-user predicates, and ξ is a single address oblivious 1-user predicate, such that,*

$$
\begin{aligned}
\theta(s, \mathbf{u}) &:= \psi_{Roles}(s, \mathbf{u}) \wedge \psi_{Lits}(s, \mathbf{u}) \wedge \psi_{Else}(s, \mathbf{u}) \\
\psi_{Lits}(s, \mathbf{u}) &:= \bigwedge_{i=0}^{m-1} [\mathit{id}(\mathbf{u}_0) = l_i] \Rightarrow \zeta_i(s, \mathbf{u}) \\
\psi_{Roles}(s, \mathbf{u}) &:= \bigwedge_{i=0}^{n-1} [\mathit{id}(\mathbf{u}_0) = \mathit{role}(s, r_i)] \Rightarrow \mu_i(s, \mathbf{u}) \\
\psi_{Else}(s, \mathbf{u}) &:= \left[\bigwedge_{i=0}^{m-1} \mathit{id}(\mathbf{u}_0) \neq l_i\right] \wedge \left[\bigwedge_{i=0}^{n-1} \mathit{id}(\mathbf{u}_0) \neq \mathit{role}(s, r_i)\right] \Rightarrow \xi(s, \mathbf{u})
\end{aligned}
$$

Note that $\mathbf{u}$ is a singleton vector and that L defines the literal guards *of θ.*

In Example 2, we illustrate how Definition 6 is applied in practice.

Example 2. To establish φ_1 from Example 1, we require a split compositional invariant θ_1, defined by the tuple $\mathit{Inv} = \langle L_2, \varnothing, \varnothing, \langle \xi_1 \rangle, \xi_2 \rangle$, where $L_2 = \{0\}$, ξ_1 is as defined in Example 1, and $\xi_2(s, \mathbf{u}) := (\mathsf{rec}(\mathbf{u}_0))_1 \geq 0$. The two instances of $\varnothing$ in *Inv* correspond to the role constraints and their associated user summaries. If *Inv* is related back to Definition 6, it follows that that $\psi_{\text{Roles}}(s, \mathbf{u}) := \top$, $\psi_{\text{Lits}}(s, \mathbf{u}) := (\mathsf{id}(\mathbf{u}_0) = 0) \Rightarrow ((\mathsf{rec}(\mathbf{u}_0))_1 = 0)$, and $\psi_{\text{Else}}(s, \mathbf{u}) := (\mathsf{id}(\mathbf{u}_0) \neq 0) \Rightarrow ((\mathsf{rec}(\mathbf{u}_0))_1 \geq 0)$. Expanding θ_1 yields,

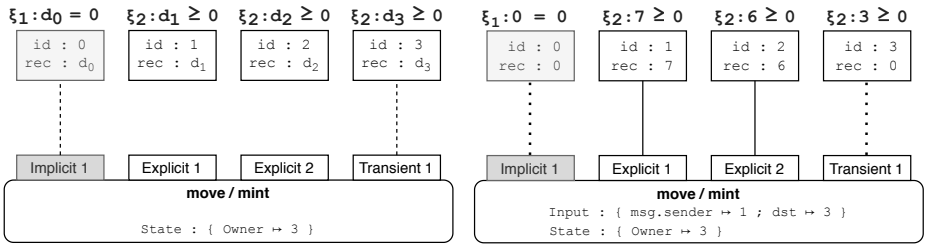
$$
\theta_1(s, \mathbf{u}) := \mathsf{id}(\mathbf{u}_0) = 0 \;\Rightarrow\; (\mathsf{rec}(\mathbf{u}_0))_1 = 0 \;\wedge\; \mathsf{id}(\mathbf{u}_0) \neq 0 \;\Rightarrow\; (\mathsf{rec}(\mathbf{u}_0))_1 \geq 0
$$

Predicate θ_1 is read as: the user at address `address(0)` has balance zero (by ξ_1), while all other users have non-negative balances (by ξ_2). □

6.2 Localizing a Smart Contract Bundle

A local (smart contract) bundle is a finite-state abstraction of a smart contract bundle. This abstraction reduces smart contract verification to finite-state model checking[8]. At a high level, each local bundle is a non-deterministic LTS, and is

[8] Recall that MiniSol is limited to bounded arrays and address-indexed mappings.



(a) A local 4-user configuration.

(b) $\mathcal{A}_1$ saturating a transaction.

Fig. 3: The local model of Fig. 2a, defined by $\mathcal{A}_1$ and θ_1 in Example 3.

constructed from three components: a smart contract, a candidate compositional invariant, and a neighbourhood. We use the term *candidate compositional invariant* to describe any predicate that has the structure of a compositional invariant, regardless of its semantic interpretation. The term *neighbourhood* describes a set of users, and is represented by the set of user addresses in the set.

Let $\mathcal{A}$ be an N-user neighbourhood and θ be a candidate compositional invariant. The local bundle corresponding to $\mathcal{A}$ and θ is defined using special user configurations called *local user configurations*. We say that an N-user network configuration $\mathbf{u}$ is local with respect to a control state s and a neighbourhood $\mathcal{A}$ whenever for any user $\mathbf{u}_i$ in $\mathbf{u}$, $\mathsf{id}(\mathbf{u}_i) \in \mathcal{A}$ and $\theta(s, \mathbf{u}_i)$. In other words, a user configuration is local if it contains precisely the user in $\mathcal{A}$, and if each user satisfies the invariant θ.

Definition 7 (Local User Configuration). *Let $\mathcal{C}$ be a contract, $\mathcal{A} \subseteq \mathbb{A}$ be an N-user neighbourhood, θ be a candidate compositional invariant for $\mathcal{C}$, $s \in \mathsf{control}(\mathcal{C})$, and $\mathbf{u} \in \mathsf{config}(\mathcal{C}, N)$. A user configuration $\mathbf{u}$ is* local *w.r.t. $\mathcal{A}$, θ, and s, written $\mathit{IsLocal}(\mathcal{A}, \theta, s, \mathbf{u})$, if $\forall a \in \mathcal{A} \cdot \exists i \in [N] \cdot \mathsf{id}(\mathbf{u}_i) = a \wedge (s, \mathbf{u}_i) \in \theta$.*

Each state of the *local bundle* for $\mathcal{A}$ and θ is a tuple $\langle s, \mathbf{u} \rangle$, where s is a control state and $\mathbf{u}$ is a N-user configuration. The transition relation is defined in terms of the transaction function f. However, the transition relation does not take $\mathbf{u}$ into account. Instead, $\langle s', \mathbf{u}' \rangle$ is a successor of $\langle s, \mathbf{u} \rangle$ if there exists a *local* user configuration $\mathbf{v}$ of s such that $\langle s', \mathbf{u}' \rangle = f(s, \mathbf{v})$.

Definition 8 (Local Bundle Reduction). *Let $\mathcal{C}$ be a contract, $\mathcal{A} \subseteq \mathbb{A}$ be an N-user neighbourhood, and θ be a candidate compositional invariant for $\mathcal{C}$. A* local bundle *is an LTS $\mathsf{local}(\mathcal{C}, \mathcal{A}, \theta) = \langle S, P, \hat{f}, s_0 \rangle$, where $S \subseteq \mathsf{control}(\mathcal{C}) \times \mathcal{N}$, $\mathcal{N} = \mathsf{config}(\mathcal{C}, N)$, $P = \mathsf{inputs}(\mathcal{C})$, $s_0 = \mathit{init}(\mathcal{C}, \mathcal{A})$, and $\hat{f}$ is*

$$\hat{f}(s, \mathbf{u}, p) = \{[\![\mathcal{C}]\!]_N(s, \mathbf{v}, p) \mid \mathbf{v} \in \mathcal{N} \wedge \mathit{IsLocal}(\mathcal{A}, \theta, s, \mathbf{v})\}$$

Example 3. We briefly illustrate $\hat{f}$ of Definition 8. Let $\mathcal{A}_1 = \{0, 1, 2, 3\}$ be a neighbourhood, and θ_1 be as in Example 2, with $\langle S, P, \hat{f}, s_0 \rangle = \mathsf{local}(\mathcal{C}, \mathcal{A}_1, \theta_1)$.

Consider a state $\langle s, \mathbf{u}\rangle \in S$, where $s = \{\texttt{owner} \mapsto \texttt{address(3)}\}$ and $\forall i \in [4] \cdot \mathsf{rec}(\mathbf{u}_i) = 0$. Now assume that `move` has been excuted with clients `address(1)` and `address(2)`. A successor state of $\langle s, \mathbf{u}\rangle$ must be determined. By definition, $(s', \mathbf{u}') \in \hat{f}(s, \mathbf{u}, p) = [\![\mathcal{C}]\!]_N(s, \mathbf{v}, p)$, where $\mathbf{v}$ is local to $\mathcal{A}_1$, θ_1 and s.

The key step is to select an appropriate $\mathbf{v}$, as depicted in Fig. 3a. This is done by first selecting an arbitrary 4-user network using all addresses in $\mathcal{A}_1$. Second, it must be assumed that each user satisfies θ_1. In Fig. 3a a network is selected in which $\forall i \in [4] \cdot \mathsf{id}(\mathbf{v}_i) = i$. As depicted in Fig. 3a, θ_1 expands such that the user at `address(0)` must satisfy ξ_1 while all other users must satisfy ξ_2.

In Fig. 3b, a satisfying assignment is selected for the funds of each user. The choice for d_1 was fixed as $\xi_1(s, \mathbf{v}_0)$ entails $d_1 = 0]$. In the case of d_2 to d_4 any non-negative value could have been selected. After the transaction is executed, $\mathsf{rec}(\mathbf{u}'_0) = 0$, $\mathsf{rec}(\mathbf{u}'_1) = 6$, $\mathsf{rec}(\mathbf{u}'_2) = 7$ and $\mathsf{rec}(\mathbf{u}'_3) = 0$, whereas the control state remains unchanged. This is a successor state, as desired. □

SW: The first result was moved to an earlier section. The second result should be easier to follow. Example 3 motivates an important result regarding compositional invariants and local bundles. Observe that $\mathbf{u}'$ is local. This is not by chance. First, by the compositionality of θ_1, all user configurations reached by $\mathsf{lts}(\mathcal{C}, \mathcal{A}_1, \theta_1)$ must be local. Second, and far less obviously, by the choice of $\mathcal{A}_1$, if a reachable user configuration is not local to θ_1, then θ_1 is not compositional. The proof of this result relies on a saturating property of $\mathcal{A}_1$.

A neighbourhood $\mathcal{A}$ is *saturating* if $\mathcal{A}$ contains an address for each participation class in the PTG of a contract. Furthermore, in the case of an implicit participation class *implicit*@x, it is required that the address in the neighbourhood is x. By construction, the saturating neighbourhood can designate a unique user to each transaction participant. The saturating property of $\mathcal{A}_1$ is depicted in Fig. 3b by the correspondence between users and participation classes.

Definition 9 (Saturting Neighbourhood). *Let $\mathcal{C}$ be a contract, $\langle G, \rho, \tau\rangle$ be a PTG of $\mathcal{C}$, and $G = (V, E, L)$. A* saturating neighbourhood *for $\langle G, \rho, \tau\rangle$ is a tuple $\langle \mathcal{A}_{Persist}, \mathcal{A}_{Trans}, \mathcal{A}_{Reps}\rangle$ s.t. $\mathcal{A}_{Persist}, \mathcal{A}_{Trans}, \mathcal{A}_{Reps} \subseteq \mathbb{A}$ and*

1. $|\mathcal{A}_{Reps}| = |\{i \in \mathbb{N} \mid i@explicit \in L(E)\}|$,
2. $|\mathcal{A}_{Trans}| = |\{i \in \mathbb{N} \mid i@trasient \in L(E)\}|$,
3. $\mathcal{A}_{Persist} = \{x \in \mathbb{A} \mid x@implicit \in L(E)\}$,
4. $\mathcal{A}_{Persist}$, $\mathcal{A}_{Trans}$, *and* $\mathcal{A}_{Reps}$ *are pairwise disjoint.*

We can use transaction saturating neighbourhoods to reduce compositionality and parameterized safety proofs to the safety of a local bundle. We start with compositionality. By definition, every candidate compositional invariant θ is also a 1-universal safety property. We first claim that if θ is compositional, then any local bundle constructed from θ must be safe with respect to θ (as in Example 3). For users that participate in a transaction, this follows directly from **Initialization** and **Consecution** (as defined in Sec. 2). For users that do not participate in a transaction, this follows directly from **Non-Interference** (as defined in Sec. 2). We also claim that for a sufficiently large neighbourhood, say

$\mathcal{A}^+$, that the converse is also true. Informally, we require a unique user for each participation class. This is satisfied whenever $\mathcal{A}^+$ contains a transaction saturating neighbourhood. We also require one additional user to ensure there is always a Non-Interference check. To ensure that θ is well-behaved, we require that $\mathcal{A}^+$ contains the literal guards of θ. We formalize this argument in Theorem 4.

Theorem 4. *Let $\mathcal{C}$ be an order-oblivious contract, θ be a candidate split compositional invariant for $\mathcal{C}$, $\langle G, \rho, \tau\rangle$ be a PTG for $\mathcal{C}$, $\langle \mathcal{A}_{Persist}, \mathcal{A}_{Trans}, \mathcal{A}_{Reps}\rangle$ be a saturating neighbourhood of $\langle G, \rho, \tau\rangle$, $\mathcal{A}_\theta$ be the literal guards of θ, $\mathcal{A} := \mathcal{A}_{Persist} \cup \mathcal{A}_{Trans} \cup \mathcal{A}_{Reps} \cup \mathcal{A}_\theta$ and $a \in \mathbb{A}\backslash\mathcal{A}$. Define $\mathcal{A}^+ := \{a\} \cup \mathcal{A}$. Then, $\mathsf{local}(\mathcal{C}, \mathcal{A}^+, \theta) \models \theta$ if and only if θ is a compositional invariant for $\mathcal{C}$.*

Proof. ...

Next we present a sound proof rule for k-universal safety. Our proof rule is an application of PCMC, and, therefore, requires a split compositional invariant θ [16]. We then proceed similarly to Theorem 4 by first computing a sufficiently large neighbourhood $\mathcal{A}^+$, and then verifying the k-universal safety of $\mathsf{local}(\mathcal{C}, \mathcal{A}^+, \theta)$. A key difference is that we are now considering k-universal safety rather than 1-universal safety. To be absolutely certain that the parameterized system is safe, we require at least k interchangeable users in $\mathcal{A}^+$. AG: Too many "we". The reader knows that everything in the paper is written by us already. Our proof rule is complete relative to θ.

Theorem 5. *Let $\mathcal{C}$ be an order-oblivious contract, θ be a split compositional invariant for $\mathcal{C}$, $\langle G, \rho, \tau\rangle$ be a PTG for $\mathcal{C}$, $\langle \mathcal{A}_{Persist}, \mathcal{A}_{Trans}, \mathcal{A}_{Reps}\rangle$ be a saturating neighbourhood of $\langle G, \rho, \tau\rangle$, and φ be a guarded universal k-safety property for $\mathcal{C}$ with literal guards $\mathcal{A}_\varphi$. Define $\mathcal{A} := \mathcal{A}_{Persist} \cup \mathcal{A}_{Trans} \cup \mathcal{A}_{Reps} \cup \mathcal{A}_\varphi$ and $\mathcal{A}^+ \subseteq \mathbb{A}$ such that $\mathcal{A} \subseteq \mathcal{A}^+$ and $|\mathcal{A}^+| = |\mathcal{A}| + \max(0, |\mathcal{A}_{Reps}| - k)$. If $\mathsf{local}(\mathcal{C}, \mathcal{A}^+, \theta) \models \varphi$, then $\forall N \in \mathbb{N} \cdot \mathsf{lts}(\mathcal{C}, N) \models \varphi$.*

Proof. ...

Remark 1. For those familiar with parameterized model-checking, it may seem surprising that Theorem 5 makes no mention of a cutoff. In PCMC, we would expect to find a *compositional cutoff* n for which safety in the non-local depends on the safety of all local-bundles smaller than n [16]. In fact, the k-universal safety of $\mathrm{Local}(\mathcal{C}, \mathcal{A}^+, \theta)$ subsumes the k-universal safety of all neighbourhoods contained within $\mathcal{A}^+$. In this way, verifying $\mathrm{Local}(\mathcal{C}, \mathcal{A}^+, \theta)$ is equivalent to verifying all neighbourhoods up to size m.

7 Implementation and Evaluation

We have implemented the technique described in this paper in an open source tool called SeaHorn. SmartACE is built upon version 0.5.9 of the Solidity compiler. It works in the following steps: (1) consumes a Solidity smart contract and validates its conformance to MiniSol, (2) performs source-code analysis and transformation (i.e., inlining of inheritance, resolving dynamic dispatch,

`CTGBuilder`), (3) generates a local bundle and models it using LLVM IR, and (4) verifies the model using SeaHorn [7]. In this section, we report on the effectiveness of SmartACE to analysis of real-world smart-contracts. A detailed description of the SmartACE architecture and of the case studies is outside of scope of this paper. Here, we only report on the key findings. Both the case studies and SmartACE are publicly available at `https://github.com/contract-ace`.

In specific, this evaluation investigates at four research questions:

RQ1. For smart contracts that do conform to MiniSol, is local reasoning effective?
RQ2. Are the restrictions of MiniSol reasonable for real-world smart contracts?
RQ3. Can SmartACE scale to real-world smart contract instances?
RQ4. Is SmartACE appropriate for real-world smart contract development?
RQ5. Is the performance of SmartACE competitive with other techniques?

SW: Moving solidity-to-cmodel onto organization. SW: Diagram here of SmartACE workflow.

To answer these research questions, we have conducted three case studies and one experiment, *Fund-Auction*, *Melon-Alchemist*, *QuantStamp*, and *VerX*, respectively. In the Fund-Auction case study, SmartACE is used to verify simple (parameterized) safety properties on two simplified smart contracts (including `Auction` in Fig. 1). Some of these properties require additional techniques that are beyond the scope of this paper (i.e., instrumenting temporal properties as in [19]). This case study addresses RQ1. An informal report on this study is available through the SeaHorn development blog[9]. The Melon-Alchemist case study investigates the PTG produced by `PTGBuilder`. By design, `PTGBuilder` is coarse-grained, and therefore, should leave space for optimizations on a per-method basis. The Melon-Alchemist smart contract bundle is used in this case study, as it is a real-world smart contract that exhibits many instances of transient and implicit participation. This case study addresses RQ2 and RQ3. The QuantStamp case study evaluates the feasibility of applying SmartACE to real-world smart contract development. SmartACE is used to verify a subset of specifications on the QuantStamp Assurance Protocol, under development by QuantStamp. This case study addresses RQ2 and RQ4. Finally, the VerX experiment compares SmartACE to VerX: a state-of-the-art smart contract verification tool [19]. This experiment is carried out against the VerX benchmark suite, and addresses both RQ2 and RQ5 on a wide scale. All evaluations were performed on an Intel® Core i7® CPU @ 2.8GHz 4-core machine with 16GB of memory running Ubuntu 18.04.

VW: Maybe we can even introduce the questions at the beginning of the section since we presumably also use the case studies to answer them.

Question: VerX gives average times. For some contracts they split the properties. It is not given "per property". I'm not sure how to combine these. VW: Not sure what their average refers to. Average per property or average out of

[9] Available at: `http://seahorn.github.io/blog/`

Experiments		Analysis Results		VerX Comparison	
Project	Properties	Time (s)	Clauses/Interference	Supported	Time (s)
Fund (Blog)	2	?	?	N/A	N/A
Auction (Blog)	2	?	?	N/A	N/A
QSPStaking	4	?	?	N/A	N/A
Overview	4	?	?	Yes	?
Alchemist	3	?	?	Yes	?
Brickblock	6	?	?	?	?
Crowdsale	9	?	?	?	?
ERC20	9	?	?	Yes	?
ICO	8	?	?	?	?
Mana	4	?	?	?	?
Melon	16	?	?	Yes	?
MRV	5	?	?	Yes	?
PolicyPal	4	?	?	?	?
VUToken	5	?	?	?	?
Zebi	2	?	?	?	?
Ziliqa	5	?	?	?	?

Table 1: Experimental results for SmartACE on various benchmarks. VW: I would drop the benchmarks that we don't support. SW: Good point. We can mention the percent we handled in the discussion.

several runs... SW: I will do a closer review of their evaluation section to see if it's alluded to.

Question: Are there any "unspoken assumptions" about how experiments I executed in this field? Number of trials? Etc? VW: Given that there shouldn't be any randomness in the tools, (I assume you always use the same seed for Z3...) I don't think we need multiple tries.

The Fund-Action Case Study. To evaluate the effectiveness of local reasoning (RQ1) we analyzed two simplified smart contracts inspired by real-world use cases (i.e., a token and an auction). For each smart contract, SmartACE verified 2 pLTL [19] properties inspired by real-world smart contract requirements (e.g., access control [4]) in timings. The inductive invariants produced by SmartACE were validated by hand. This demonstrates that local reasoning is an effective technique for smart contract analysis.

The Melon-Alchemist Case Study. Report on scalability. We reduce from n to m clients in the local bundle. Suggests future work on refining PTG.

The QuantStamp Case Study. Report on difficulty to integrate. How many hours it took to do manually. Number of clauses. Most of the work was mechanical (instrumentation) or finding interference. We (by hand) extended SmartACE to arrays, and explored NLIA. This suggests future work.

The VerX Experiment: We performed n magnitudes faster. We could handle m precent of contracts. We observed aggregates (Dejan and to Shuvendu) + suggest future work. Major difficulties were arrays, loosely coupled bundles, etc.

8 Related Works

- **Compositional Reasoning:** PCMCP [16]; View abstraction [1]; Owicki-Gries [18]; Symmetry in synchronized control-user networks [5]; Scalarsets (less general than our addresses) [17]
- **Communication in Distributed Systems:** ?
- **Smart Contract Verification:** VerX (our transactional semantics are similar to delayed predicate abstraction) [19]; VeriSmart [23]; VeriSol [25]; solc-verify [9]; Zeus (also using CHC) [11]; Txn traces and temporal properties [21]; Static analysis for access policies [4]; VeriSolid's semantics are similar to the small step semantics in our proofs [14]
- **Smart Contract Concurrency:** Smart contracts as concurrent systems [20]; EO bugs and POR [12]

9 Conclusions

...

References

1. Abdulla, P.A., Haziza, F., Holík, L.: Parameterized verification through view abstraction. STTT **18**(5), 495–516 (2016)
2. Atzei, N., Bartoletti, M., Cimoli, T.: A survey of attacks on Ethereum smart contracts. In: POST. LNCS, vol. 10204, pp. 164–186. Springer (2017)
3. Blackshear, S., Cheng, E., Dill, D.L., Gao, V., Maurer, B., Nowacki, T., Pott, A., Qadeer, S., Rain, Russi, D., Sezer, S., Zakian, T., Zhou, R.: Move: A language with programmable resources (2019), https://developers.libra.org/docs/move-paper
4. Brent, L., Grech, N., Lagouvardos, S., Scholz, B., Smaragdakis, Y.: Ethainter: A smart contract security analyzer for composite vulnerabilities. In: Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation. p. 454–469. Association for Computing Machinery, New York, NY, USA (2020). https://doi.org/10.1145/3385412.3385990, https://doi.org/10.1145/3385412.3385990

5. German, S.M., Sistla, A.P.: Reasoning about systems with many processes. J. ACM **39**(3), 675–735 (Jul 1992). https://doi.org/10.1145/146637.146681, https://doi.org/10.1145/146637.146681
6. Gilad, Y., Hemo, R., Micali, S., Vlachos, G., Zeldovich, N.: Algorand: Scaling byzantine agreements for cryptocurrencies. In: Proceedings of the 26th Symposium on Operating Systems Principles. p. 51–68. SOSP '17, Association for Computing Machinery, New York, NY, USA (2017). https://doi.org/10.1145/3132747.3132757, https://doi.org/10.1145/3132747.3132757
7. Gurfinkel, A., Kahsai, T., Komuravelli, A., Navas, J.A.: The SeaHorn verification framework. In: CAV. LNCS, vol. 9206, pp. 343–361. Springer (2015)
8. Gurfinkel, A., Shoham, S., Meshman, Y.: SMT-based verification of parameterized systems. In: FSE. pp. 338–348. ACM (2016)
9. Hajdu, Á., Jovanovic, D.: solc-verify: A modular verifier for Solidity smart contracts. In: Verified Software. Theories, Tools, and Experiments - 11th International Conference, VSTTE 2019, New York City, NY, USA, July 13-14, 2019, Revised Selected Papers. Lecture Notes in Computer Science, vol. 12031, pp. 161–179. Springer (2019). https://doi.org/10.1007/978-3-030-41600-3_11
10. Jiang, B., Liu, Y., Chan, W.K.: ContractFuzzer: Fuzzing smart contracts for vulnerability detection. In: ASE. pp. 259–269. ACM (2018)
11. Kalra, S., Goel, S., Dhawan, M., Sharma, S.: ZEUS: Analyzing safety of smart contracts. In: NDSS. The Internet Society (2018)
12. Kolluri, A., Nikolic, I., Sergey, I., Hobor, A., Saxena, P.: Exploiting the laws of order in smart contracts. In: ISSTA. pp. 363–373. ACM (2019)
13. Luu, L., Chu, D., Olickel, H., Saxena, P., Hobor, A.: Making smart contracts smarter. In: CCS. pp. 254–269. ACM (2016)
14. Mavridou, A., Laszka, A., Stachtiari, E., Dubey, A.: Verisolid: Correct-by-design smart contracts for ethereum. In: Financial Cryptography and Data Security. pp. 446–465. Springer International Publishing (2019)
15. Nakamoto, S.: Bitcoin: A peer-to-peer electronic cash system (2008), https://bitcoin.org/bitcoin.pdf
16. Namjoshi, K.S., Trefler, R.J.: Parameterized compositional model checking. In: Proceedings of the 22nd International Conference on Tools and Algorithms for the Construction and Analysis of Systems - Volume 9636. p. 589–606. Springer-Verlag, Berlin, Heidelberg (2016), https://doi.org/10.1007/978-3-662-49674-9_39
17. Norris Ip, C., Dill, D.L.: Better verification through symmetry. In: Computer Hardware Description Languages and their Applications, pp. 97 – 111. IFIP Transactions A: Computer Science and Technology, North-Holland, Amsterdam (1993). https://doi.org/https://doi.org/10.1016/B978-0-444-81641-2.50012-5, http://www.sciencedirect.com/science/article/pii/B9780444816412500125
18. Owicki, S., Gries, D.: An axiomatic proof technique for parallel programs i. Acta Inf. **6**(4), 319–340 (Dec 1976), https://doi.org/10.1007/BF00268134
19. Permenev, A., Dimitrov, D., Tsankov, P., Drachsler-Cohen, D., Vechev, M.: Verx: Safety verification of smart contracts. In: 2020 IEEE Symposium on Security and Privacy, SP 2020, San Francisco, CA, USA, May 18-21, 2020. pp. 1661–1677. IEEE (2020). https://doi.org/10.1109/SP40000.2020.00024, https://doi.org/10.1109/SP40000.2020.00024
20. Sergey, I., Hobor, A.: A concurrent perspective on smart contracts. In: FC. LNCS, vol. 10323, pp. 478–493. Springer (2017)
21. Sergey, I., Kumar, A., Hobor, A.: Temporal properties of smart contracts. In: ISoLA. LNCS, vol. 11247, pp. 323–338. Springer (2018)

22. Sergey, I., Nagaraj, V., Johannsen, J., Kumar, A., Trunov, A., Hao, K.C.G.: Safer smart contract programming with scilla. Proc. ACM Program. Lang. **3**(OOPSLA) (2019). https://doi.org/10.1145/3360611, https://doi.org/10.1145/3360611
23. So, S., Lee, M., Park, J., Lee, H., Oh, H.: VeriSmart: A highly precise safety verifier for Ethereum smart contracts. In: S&P. IEEE Computer Society (2020), to appear.
24. Szabo, N.: Smart contracts: Building blocks for digital markets (1996), http://www.fon.hum.uva.nl/rob/Courses/InformationInSpeech/CDROM/Literature/LOTwinterschool2006/szabo.best.vwh.net/smart_contracts_2.html
25. Wang, Y., Lahiri, S.K., Chen, S., Pan, R., Dillig, I., Born, C., Naseer, I., Ferles, K.: Formal verification of workflow policies for smart contracts in Azure blockchain. In: VSTTE. LNCS, Springer (2019), to appear.
26. Wood, G.: Ethereum: A secure decentralised generalised transaction ledger (2014), http://gavwood.com/paper.pdf
27. Wüstholz, V., Christakis, M.: Harvey: A greybox fuzzer for smart contracts. In: FSE. ACM (2020), to appear

A The Syntax of MiniSol

⟨FName⟩ ::= *a valid function name*
⟨VName⟩ ::= *a valid variable name*
⟨CName⟩ ::= *a valid contract name*
⟨PrimitiveT⟩ ::= `int` | `uint` | `bool` | `address`
⟨MappingT⟩ ::= `mapping(` `address` `=>` ⟨ComplexT⟩ `)`
⟨ComplexT⟩ ::= ⟨PrimitiveT⟩ | ⟨MappingT⟩ | ⟨CName⟩
⟨Literal⟩ ::= $\mathbb{B}$ | $\mathbb{N}_8$ | ... | $\mathbb{N}_{256}$ | $\mathbb{Z}_8$ | ... | $\mathbb{Z}_{256}$ | $\mathbb{A}$
⟨Global⟩ ::= `block.number` | `block.timestamp` | `tx.origin` | `msg.sender` | `msg.value`
⟨Operator⟩ ::= `==` | `!=` | `<` | `>` | `+` | `-` | `*` | `/` | `&&` | `||` | `!`
⟨Expr⟩ ::= ⟨Literal⟩ | ⟨VName⟩ | ⟨Global⟩ | ⟨Expr⟩ ⟨Operator⟩ ⟨Expr⟩
| `this` | ⟨Expr⟩`.`⟨FName⟩`.value(` ⟨Expr⟩ `)` `(` ⟨Expr⟩`, ...` `)`
| ⟨FName⟩ `(` ⟨Expr⟩`, ...` `)` | `address(` ⟨CName⟩ `)`
| ⟨Expr⟩`.balance` | ⟨Expr⟩ `[` ⟨Expr⟩ `]` ... `[` ⟨Expr⟩ `]`
⟨Decl⟩ ::= ⟨ComplexT⟩ ⟨VName⟩
⟨Assign⟩ ::= ⟨VName⟩ `=` ⟨Expr⟩ | ⟨Expr⟩ `=` `new` ⟨CName⟩`(` ⟨Expr⟩`, ...` `)`
| ⟨Expr⟩ `[` ⟨Expr⟩ `]` ... `[` ⟨Expr⟩ `]` `=` ⟨Expr⟩
⟨Stmt⟩ ::= ⟨Decl⟩ | ⟨Assign⟩ | `require(` ⟨Expr⟩ `)` | `assert(` ⟨Expr⟩ `)` | `return`
| `return` ⟨Expr⟩ | `if(` ⟨Expr⟩ `)` `{` ⟨Stmt⟩ `}` | ⟨Stmt⟩`;` ⟨Stmt⟩
| ⟨Expr⟩`.transfer(` ⟨Expr⟩ `)`
⟨Payable⟩ ::= `payable` | ϵ
⟨Vis⟩ ::= `public` | `external` | `internal`
⟨Args⟩ ::= `()` | `(` ⟨Decl⟩`, ...` `)`
⟨RetVal⟩ ::= `returns` `(` ⟨PrimitiveT⟩ `)` | ϵ
⟨Ctor⟩ ::= `constructor` ⟨Args⟩ `public` ⟨Payable⟩ `{` ⟨Stmt⟩ `}`
⟨Func⟩ ::= `function` ⟨FName⟩ ⟨Args⟩ ⟨Vis⟩ ⟨Payable⟩ ⟨RetVal⟩ `{` ⟨Stmt⟩ `}`
⟨Contract⟩ ::= `contract` ⟨CName⟩ `{` ⟨Decl⟩`;` ...`;` ⟨Ctor⟩ ⟨Func⟩ ... `}`
⟨Bundle⟩ ::= ⟨Contract⟩ ⟨Contract⟩ ...

Fig. 4: The formal grammar of the MiniSol language.

In Sec. 3 we introduced contracts, functions, inputs, constructors, variable declarations, primitive types, mappings, and contract references. Formally, these are defined w.r.t. the MiniSol syntax in Fig. 4. Contracts, functions, and variables are any expressions derived from ⟨Contract⟩, ⟨Func⟩, and ⟨Decl⟩, respectively. An input is any non-terminal child of ⟨Args⟩. A constructor is any function derived from ⟨Ctor⟩. Primitive, mapping, and contract reference types are any non-terminals derived from ⟨PrimitiveT⟩, ⟨MappingT⟩, ⟨CName⟩, respectively.

We also introduced two primitive domains: $\mathbb{A}$ and $\mathbb{D}$. In reality, $\mathbb{D}$ consists of serveral sets that appear in Sec. 3. Formally

$$\mathbb{D} := \mathbb{B} \cup \mathbb{N}_8 \cup \mathbb{N}_{16} \cup \cdots \cup \mathbb{N}_{256} \cup \mathbb{Z}_8 \cup \mathbb{Z}_{16} \cup \cdots \cup \mathbb{Z}_{256}$$

where $\mathbb{B}$ is the set of Boolean literals, $\mathbb{N}_n$ is the set of n-bit unsigned integer literals, and $\mathbb{Z}_m$ is the set of m-bit signed integer literals.

B The Open-Bid LTS

In this section we formally define the bundle of $\mathcal{C} := \texttt{Auction}$, from Fig. 1. We highlight practicle issues that were overlooked in Sec. 3. Note that in this example, we no longer assume that Fig. 1 consists of a single transaction type. This means that the transaction input must also include a transaction type. For this we define the set $\mathrm{Tx} := \{\texttt{constructor}, \texttt{bid}, \texttt{withdraw}, \texttt{close}\}$.

The control states of $\mathcal{C}$ are $\mathsf{control}(\mathcal{C}) \subseteq (\mathbb{A} \times \mathbb{A}) \times (\mathbb{N}_{256} \times \mathbb{B}) \times (\mathbb{N}_{256} \times \mathbb{B})$. For each state $(a_1, a_2, d_1, d_2, aux_1, aux_2) \in \mathsf{control}(\mathcal{C})$, a_1, a_2, d_1, and d_2 correspond to `owner`, `highestBidder`, `endTime`, and `isClosed`, respectively. aux_1 is an auxiliary variable used to store the timestamp of the last transaction. This allows us to enforce that time is monotonic. aux_2 is also an auxiliary variable, used to signal whether the constructor has been called. The constructor is the next transaction if and only if $aux_1 = \texttt{false}$.

The user states of $\mathcal{C}$ are $\mathsf{control}(\mathcal{C}) \subseteq \mathbb{A} \times \mathbb{N}_{256} \times \mathbb{N}_{256}$. For each state $(x, y_1, y_2) \in \mathsf{control}(\mathcal{C})$, x is the user's address, y_1 is the user's balance, and y_2 is the user's bid.

The transaction inputs are $\mathsf{inputs}(\mathcal{C}) \subseteq \mathrm{Tx} \times \mathbb{A} \times \mathbb{N}_{256} \times \mathbb{N}_{256}$. For any $p = (t, x, y_1, y_2) \in \mathsf{inputs}(\mathcal{C})$, our interpretation of p depends on t. In all cases, x is `msg.sender` and y_1 is `block.timestamp`. If $t = \texttt{constructor}$, then y_2 represents `duration`. If $t = \texttt{bid}$, then y_2 represents `msg.value`. In all other cases, d_2 is unused.

For readability, we give transactional semantics for $f = [\![\mathcal{C}]\!]_N$. In Sec. 3, it was mentioned that $[\![\mathcal{C}]\!]_N$ can be reduced to internal and synchronized transitions. However, the existance of such a reduction is only important for proofs. SW: This might be an appendix?

$$f(s, \mathbf{u}, \langle t, x, y_1, y_2 \rangle) = \begin{cases} g_1(s, \mathbf{u}, x, y_1, y_2) & \text{if } t = \texttt{constructor} \\ g_2(s, \mathbf{u}, x, y_1, y_2) & \text{if } t = \texttt{bid} \\ g_2(s, \mathbf{u}, x, y_1) & \text{if } t = \texttt{withdraw} \\ g_4(s, \mathbf{u}, x, y_1) & \text{if } t = \texttt{close} \end{cases}$$

Full transition function