Communication-Directed Locality for Compositional Verification of Smart Contracts

Scott Wesley[1], 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 re-frame this communication as message passing, and implement a static-analysis algorithm to over-approximate the possible communications. Using this over-approximation, we then present a *local model* that can simulate networks of arbitrary size. Based on this local model, we finally 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 a state-of-the-art smart-contract verification tool and show order-of-magnitude improvements in time.

1 Introduction

Smart contracts present new challenges for the formal-methods community. Aside from testing (e.g., [18,38]) and static analysis (e.g. [5]), many approaches focus either on semi- (e.g., [37,16]) or fully-automated (e.g., [19,29,34]) verification. A more limited selection of research has focused on the distributed aspects of smart contracts [23,31,21].

At their core, smart contracts are distributed programs that facilitate flow of information between many distributed participants. For example, the `Auction` smart contract in Fig. 1 allows an arbitrary number of participants to `bid` until an `endTime` is reached. A key property for `Auction` is **Prop. 1**, "every non-zero bid is distinct", that must hold for any pair of participants. However, most automated verification techniques shy from the distributed nature of contracts by either focusing on properties of individual functions (e.g., integer overflow [19]), fixing number of participants [38], or relying on quantified reasoning [29,37,16,39].

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

  mapping(address => uint) bids;

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

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

    uint max = bids[topBidder];
    uint newBid = bids[msg.sender] +
         msg.value;
    require(max < newBid);

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

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

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

    uint max = bids[topBidder];
    owner.transfer(max);
    isClosed = false;
  }
}
```

Fig. 1: An example smart contract that implements an open-bid auction.

In this paper, we propose a radically different approach based on the local reasoning of Parameterized Compositional Model Checking (PCMC) [26]. In essence, we show how to reduce a smart-contract verification problem from arbitrary participants to a select few, while preserving completeness—if a property holds on a reduced system, it holds on the original one. This enables using existing powerful software model checkers (we use SeaHorn) to verify parameterized properties of complex contracts. Our approach is orders of magnitude faster than VerX [29] (on their benchmarks), requires only modest manual guidance and no quantified reasoning.

PCMC [26] is a form of compositional reasoning that breaks verification into two tasks: (1) showing that the system is "uniform" (i.e., the number of participants is bounded, and participants are interchangeable under symmetry); and (2) finding a split compositional inductive invariant. In its simplest form, PCMC is a variant of the Owicki-Gries [28] proof rule.

In this paper, we instantiate PCMC for Solidity to reduce parameterized safety checking to finite-state model checking. This is highly non-trivial. First, we show that contracts can be re-framed in terms of explicit communication, and prove that their communication patterns are over-approximated by *Participation Topology Graphs (PTG)* (see Sec. 4). A PTG is computable for every contract, and provides an upper bound on the number of participants in a transaction. Second, using the PTG, a finite model, called a *local bundle*, is constructed with sufficiently many participants to exercise all communication patterns (see Sec. 5). Our key result is that if the participants of the local bundle are abstracted by an interference invariant, then the safety certificate of the local bundle is an inductive invariant of the contract. We implement and validate this technique in SmartACE, using SeaHorn [14] as our model checker (see Sec. 6).

2 Background

In this section, we briefly recall Parameterized Compositional Model Checking (PCMC) [26]. We write $\mathbf{u} = (u_0, \dots u_{n-1})$ for a vector of n elements, and $\mathbf{u}_i$ for the i-th element of $\mathbf{u}$. For a natural number $n \in \mathbb{N}$, we write $[n]$ for $\{0, \dots, n-1\}$.

Labeled Transition Systems. 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 relation, 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, $(s_0, a_1, s_1, \dots, a_k, s_k)$, such that $\forall i \in [k] \cdot s_{i+1} \in T(s_i, a_{i+1})$. A state s is *reachable* in P if it appears in some trace $(s_0, a_1, \dots, s_k)$ of P; that is, $\exists i \in [k] \cdot s_i = s$. 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 [26], such systems are called synchronized control-user networks (SCUNs). We use N to denote the number of processes, and the set $[N]$ to denote process identifiers. We consider SCUNs in which processes only synchronize with the control process and do not execute any code on their own.

An SCUN $\mathcal{N}$ is a tuple $(S_C, S_U, A, T_I, T_S, c_0, u_0)$, where S_C is a set of control states, S_U a set of user states, $A = A_I \cup A_S$, A_I a set of internal actions, A_S a set of synchronized actions, $T_I : S_C \times A_I \to S_C$ an internal transition function, $T_S : S_C \times S_U \times A_S \to S_C \times S_U$ a synchronized transition function, and $c_0 \in S_C$ 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 $S = S_C \times (S_U)^N$, $A = A_I \cup (A_S \times [N])$, $s_0 = (c_0, u_0, \dots, u_0)$, and $T : S \times A \to S$ such that: (1) if $a \in A_I$, then $T((c, \mathbf{u}), a) = (c', \mathbf{u})$, where $c' = T_I(c, a)$ and (2) if $(a, i) \in A_S \times [N]$, then $T((c, \mathbf{u}), (a, i)) = (c', \mathbf{u}')$ where $(c', \mathbf{u}'_i) = T_S(c, \mathbf{u}_i)$, and $\forall j \in [N] \cdot i \neq j \Rightarrow \mathbf{u}'_j = \mathbf{u}_j$.

Parameterized Compositional Model Checking (PCMC). Parameterized systems have parameterized properties [15,26]. A *k-universal safety property* [15] is a predicate $\varphi \subseteq S_C \times (S_U)^k$. A state $(c, \mathbf{u})$ satisfies predicate φ if $\forall \{i_1, \dots, i_k\} \subseteq [N] \cdot \varphi(c, \mathbf{u}_{i_1}, \dots, \mathbf{u}_{i_k})$. A parameterized system $P(N)$ satisfies predicate 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 often make use of compositional reasoning, e.g., [2,15,26,28]. Here, we use PCMC [26]. The keys to PCMC are *uniformity*—the property that interacting groups of users are finite and interchangeable—and a

[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.

compositional invariant—a predicate that summarizes the reachable states of some process, while being closed under the actions of any other process. For an SCUN, the compositional invariant is given by two predicates $\theta_C \subseteq S_C$ and $\theta_U \subseteq S_C \times S_U$ satisfying:

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 $c' = T_I(c, a)$, then $c' \in \theta_C$ and $(c', u) \in \theta_U$;
Non-Interference If $c \in \theta_C$, $(c, u) \in \theta_U$, $(c, v) \in \theta_U$, $u \neq v$, $a \in A_S$, and $(c', u') = T_S((c, u), a)$ then $(c', v) \in \theta_C$.

By PCMC [26], if $\forall u \in \theta_C \cdot \forall \{(s, u_1), \dots, (s, u_k)\} \subseteq \theta_U \cdot \varphi(s, u_1, \dots, u_k)$, then $P \models \varphi$. This is as an extension of Owicki-Gries [28], where θ_C summarizes the acting process and θ_U summarizes the interfering process. For this reason, we call θ_C the *inductive invariant* and θ_U the *interference invariant*.

3 MiniSol: Syntax and Semantics

We work with MiniSol, a subset of Solidity. 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 simplify the language semantics by reducing supported language features. In particular, MiniSol removes features that are outside the scope of our analysis including inheritance and cryptographic operations. Throughout the section, we illustrate MiniSol using `Auction` in Fig. 1.

A MiniSol *smart contract* is a collection of variables and transactions (i.e., functions) available for users (similar to a class in object-oriented programming). A transaction is a deterministic sequence of operations. Each smart-contract user has a unique identifier, known as an *address*, and maintains a *balance* of cryptocurrency it owns. We view a smart contract as operating in an SCUN, the control process executes a transaction sequentially, and user processes are contract users that communicate with the control process. Each transaction is an ordered sequence of internal (to update control state) and synchronized (to access user data) actions.

A constructor is a special transaction that is executed upon contract creation. `Auction` in Fig. 1 is a contract with a single argument constructor (line 9), argument-free functions (lines 14, 26, and 36), and state variables (lines 2–7).

MiniSol has three types: *primitive* (e.g., `int256`), *mapping* (e.g., `mapping`), and *contract reference* (e.g., `Auction`). Primitive type is further partitioned into *address* (i.e., `address`) and *non-address*. Each typed variable is further classified as either *state*, *input*, or *local*. An address- and a non-address-typed state variable are called *role* and *datum*, respectively. A mapping-typed state variable is called a *mapping*. An address- and a non-address-typed input are called *client* and *argument*, respectively. A contract-reference-typed variable is called *tightly-coupled* if it is assigned only in the constructor. A MiniSol program is *tightly-coupled* whenever all of its contract-reference-typed variables are tightly-coupled. In `Auction`, there are: 2 roles (`owner` and `topBidder`), 2 contract data (`endTime` and `isClosed`), 1 mapping (`bids`), 1 user (`msg.sender`) common across all transactions, at most 2 arguments in any transaction (`now` and `msg.value`, or `now` and `duration`).

Note that in MiniSol, *user* is used to denote any user process within the control-user network, and *client* is defined relative to a transaction f, and refers to a user that is given as an input to f.

Semantics of MiniSol. Let $\mathcal{C}$ be a MiniSol program. Assume that $\mathcal{C}$ is tightly-coupled (i.e., no dynamic dispatch), and has a single transaction tr[6]. An N-user *bundle* is an N-user network of several (possibly identical) MiniSol programs.

The semantics of a 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*, $\mathsf{inputs}(\mathcal{C}) = P$ is the set of *inputs*, $f : S \times P \to S$ is the *transition function*, and s_0 is the initial state. We assume, without loss of generality, that there is a single control process[7]. Let $\mathbb{A}$ and $\mathbb{D}$ denote the finite set of address and non-address values, respectively. To ensure that $\mathcal{C}$ is order-oblivious (see Appendix C), we require that mappings are from $\mathbb{A}$ to $\mathbb{D}$. We assume that the balance of each user is a mapping.

The state space of the $\mathsf{lts}(\mathcal{C}, N)$ is determined by the state variables of $\mathcal{C}$. Let n, m, and k be the numbers of roles, data, and mappings in $\mathcal{C}$, respectively. State variables are stored by their numeric index, i.e., variable 0, 1, etc. Thus, $S_C \subseteq \mathbb{A}^n \times \mathbb{D}^m$ and $S_U \subseteq \mathbb{A} \times \mathbb{D}^k$, and 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 $\mathsf{data}(c, i) = \mathbf{y}_i$ is the i-th datum of $\mathcal{C}$. For $u = (z, \mathbf{y}) \in S_U$, $\mathsf{id}(u) = z$ is the address of u and $\mathsf{map}(u) = \mathbf{y}$ are the mapping values of u.

The transition function of the $\mathsf{lts}(\mathcal{C}, N)$ is determined by the (usual) semantics of the single transaction tr of $\mathcal{C}$, i.e., $f = [\![tr]\!]_N$. That is, given input, f updates the state variables according to the source code of tr. Let q and r be the number of clients and arguments of tr, respectively. Then, the inputs of $\mathsf{lts}(\mathcal{C}, N)$ 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. For a fixed input p, 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.

The initial state of $\mathsf{lts}(\mathcal{C}, N)$ is $s_0 = (c, \mathbf{u}) \in S$, where $c = (\mathbf{0}, \mathbf{0})$, $\forall i \in [N] \cdot \mathsf{map}(\mathbf{u}_i) = \mathbf{0}$, and $\forall i \in [N] : \mathsf{id}(\mathbf{u}_i) = i$. That is, all variables are zero-initialized and each user has a unique address.

For example, $\mathsf{lts}(\text{Auction}, 4) = (S, P, f, s_0)$ is the 4-user bundle of `Auction`. Let $(c, \mathbf{u}) \in S$ and $p \in P$ be an input s.t. `topBidder` $=$ `address(2)`, `endTime` $= 100$, the bid of the user at `address(2)` is 5, and transaction `bid()` is executed by a user at `address(1)` with `msg.value` $= 10$. As `topBidder` is the second address state variable, it is the second role, and, therefore, $\mathsf{role}(s, 1) =$ `address(2)`. Similarly, `endTime` is the first non-address state variable and the first datum, and, therefore,

[6] If transaction names are taken as inputs, then this extends to multiple transactions.
[7] All smart contracts are tightly coupled, so multiple contracts can be combined.

$\mathsf{data}(s, 0) = 100$. Recall that users are ordered from 0 to 3, and that the first mapping (e.g., $(\mathsf{map}(\mathbf{u}_2))_1$) is reserved for balances. Therefore, the bid of the user at `address(2)` is $(\mathsf{map}(\mathbf{u}_2))_1 = 5$. As for the input, $\mathsf{client}(p, 0) =$ `address(1)` and $\mathsf{arg}(p, 0) = 10$. A complete LTS for this example is in Appendix B.

MiniSol misses important features of Solidity. Some are syntactic (i.e., inheritance, enums, structs, bounded arrays) and are supported by our implementation. Other are non-trivial, but orthogonal to parameterized verification (e.g., cryptographic functions, bit-precise arithmetic, gas constraints). The challenge of extending local reasoning from this paper to dynamic dispatch, unbounded arrays, address inequalities, and mappings to address variables remains open.

4 Participation Topology

The core functionality of any smart contract is communication between users. Usually, users communicate by reading from and writing to designated mapping entries. That is, the communication paradigm is shared memory. However, it is convenient for parameterized 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 u participates in communication during a transaction f whenever u affects execution of f or f affects a state of u. We call this *influence*. For example, in Fig. 2a, the `msg.sender` influences `move` on line 7. Similarly, `move` influences `msg.sender` on line 8, and `dst` on 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.

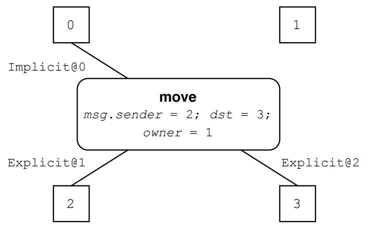
In this section, let $\mathcal{C}$ be a contract, $N \in \mathbb{N}$ the network size, $(S, P, f, s_0) = \mathsf{lts}(\mathcal{C}, N)$, and $p \in P$. A user $x \in \mathbb{A}$ *influences* transaction f_p if there exists an $s \in \mathsf{control}(\mathcal{C})$, $\mathbf{u}, \mathbf{v} \in \mathsf{config}(\mathcal{C},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

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

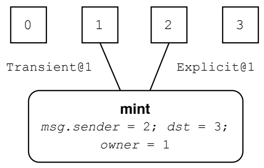
Smart contracts facilitate communication between multiple users over multiple transactions. We need to know the set of all possible participants, and the cause of the participation—we call this the *participation topology (PT)*. A PT associates each communication (sending or receiving) with one or more participation classes, called *explicit*, *transient*, and *implicit*. The participation is *explicit* whenever the participant is a client of the transaction; *transient* if the participant is assigned to at least one role during the transaction (as roles can change);

```
 1 contract SimpleToken {
 2     address owner;
 3     mapping (address => int) funds;
 4
 5     function move(address dst) public {
 6         require (dst != address(0));
 7         require (funds[msg.sender] > 0);
 8         require (funds[dst] + 1 > funds[
               dst]);
 9         funds[msg.sender]  = 1;
10         funds[dst] += 1;
11     }
12
13     function mint(address dst) public {
14         require (msg.sender == owner);
15         funds[dst] += 1;
16     }
17 }
```

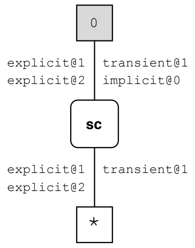
(a) Contract source text.



(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`.

implicit whenever there is at least one state s.t. 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)` on line 6 of Fig. 2a.

Definition 2 (Participation Topology). *A* participation topology *of a transaction f_p is a tuple $\mathsf{pt}(\mathcal{C}, N, p) := (Explicit, Transient, Implicit)$, 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 users, 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 blocked by a failed assertion on line 14.

Def. 2 is semantic and dependent on inputs. For analysis, a syntactic summary of the PT over all inputs is required. This is analogous to over-approximating control flow with a graph [3]. This motivates the *Participation Topology Graph* (PTG) that is a syntactic over-approximation of all possible PTs 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*. For any 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 $AP(\mathcal{C}) := \{\mathit{explicit}@i \mid i \in [n]\} \cup \{\mathit{transient}@i \mid i \in [m]\} \cup \{\mathit{implicit}@x \mid x \in \mathbb{A}\}$.

Definition 3 (Participation Topology Graph). *A* participation topology graph *for a contract $\mathcal{C}$ is a tuple (G, ρ, τ), 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 $\mathsf{pt}(\mathcal{C}, N, p) = (\mathit{Explicit}, \mathit{Transient}, \mathit{Implicit})$,*

1. *If $\mathit{Explicit}(i, x)$, then there exists some $(p, u) \in \rho$ and $(p, x, v) \in \tau$ such that $(u, v) \in E$ and $\mathit{explicit}@i \in L(u, v)$.*
2. *If $\mathit{Transient}(i, x)$, then there exists some $(p, u) \in \rho$ and $(p, x, v) \in \tau$ such that $(u, v) \in E$ and $\mathit{transient}@i \in L(u, v)$.*
3. *If $\mathit{Implicit}(x)$, then there exists some $(p, u) \in \rho$ and $(p, x, v) \in \tau$ such that $(u, v) \in E$ and $\mathit{implicit}@x \in L(u, v)$.*

In Def. 3, τ and ρ map inputs and users to vertices, respectively. An edge between an input and a user indicates the potential for participation. The labels describe the potential participation classes. As an example, Fig. 2d is a PTG for Fig. 2a, where all inputs map to sc, the user at `address(0)` maps to vertex 0, and all other users map to $\star$.

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

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

For any PT, there are many over-approximating PTGs. The weakest PTG joins every user to every transaction using all possible labels. Fig. 2d, shows 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 all other users. Such a PTG is constructed automatically using an algorithm, `PTGBuilder`.

`PTGBuilder` takes a contract $\mathcal{C}$ and returns a PTG. PTG construction is reduced 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` computes $(\mathit{Args}, \mathit{Roles}, \mathit{Lits}, \mathit{Ops})$, 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 to a sink; (4) *Ops* is the set of address cast expressions that propagate to a sink.

Finally, a PTG is constructed as (G, ρ, τ), where $G = (V, E, L)$, $\rho \subseteq \mathsf{inputs}(\mathcal{C}) \times V$, $\tau \subseteq \mathsf{inputs}(\mathcal{C}) \times \mathbb{A} \times V$, sc, and $\star$ are unique vertices:

1. If $Ops = \varnothing$, then $V := \{sc; \star\} \cup Lits$, else $V := \{sc\} \cup \mathbb{A}$;
2. $E := \{(sc, v) \mid v \in V \backslash \{sc\}\}$;
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. $\forall e = (sc, v) \in E, \forall x \in \mathbb{A}$, $implicit@x \in L(e) \iff v = x$;
6. $\rho := \{(p, sc) \mid p \in \mathsf{inputs}(\mathcal{C})\}$;
7. If $Ops = \varnothing$, then $\tau := \{(p, x, \star) \mid p \in \mathsf{inputs}(\mathcal{C}), x \in \mathbb{A} \backslash Lits\} \cup \{(p, x, x) \mid p \in \mathsf{inputs}(\mathcal{C}), x \in Lits\}$, else $\tau := \{(p, x, x) \mid p \in \mathsf{inputs}(\mathcal{C}), x \in \mathbb{A}\}$.

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. 5) by providing an upper bound on the number of and the classes of the users required for the completeness of local reasoning. Details are provided in Appendix D.

5 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. 5.1, we introduce restrictions, on properties and interference invariants, that expose address dependencies as syntactic patterns. In Sec. 5.2, we define local bundle reductions, which reduce 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.

5.1 Guarded Properties and Split Invariants

Properties and invariants might depend on addresses. However, local reasoning requires that all address dependencies are explicit. To resolve this, we introduce two syntactic forms that make addresses explicit: the guarded universal safety property and the split interference invariant. We build both forms from so called *address oblivious* predicates that do not depend on user addresses.

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 *address similar* users. Two length-k user lists, $\mathbf{u}$ and $\mathbf{v}$, are k-address similar if $\forall i \in [k] \cdot \mathsf{map}(\mathbf{u}_i) = \mathsf{map}(\mathbf{v}_i)$. The predicate ξ is address oblivious if, for every choice of $s \in \mathsf{control}(\mathcal{C})$ and every 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. This is formalized by the following definition.

Definition 4 (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 (L, R, ξ), where $L \subseteq \mathbb{A} \times [k]$, $R \subseteq \mathbb{N} \times [k]$ and ξ is an address oblivious k-user predicate, such that*

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

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

Note that $\mathcal{A} = \{a \mid (a, i) \in L\}$ defines the literal guards *of φ.*

Def. 4 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 $\mathsf{id}(\mathbf{u}_0) =$ `address(0)`, then $(\mathsf{map}(\mathbf{u}_0))_1 = 0$. In this form, **Prop. 3** corresponds to the guarded 1-universal safety property φ_1 defined by $(L_1, \varnothing, \xi_1)$, where $L_1 = \{(0, 1)\}$ and $\xi_1(s, \mathbf{u}) := (\mathsf{map}(\mathbf{u}_0))_1 = 0$. The second set is $\varnothing$ as there are no role constraints in **Prop. 3**. Following Def. 4, $\varphi_1(s, \mathbf{u}) := (0 = \mathsf{id}(\mathbf{u}_0)) \Rightarrow ((\mathsf{map}(\mathbf{u}_0))_1 = 0)$. Note that $\mathbf{u}$ is a singleton vector, and that φ_1 has 1 literal guard, given by $\{0\}$. □

The syntax of a *split interference invariant* is similar to the guarded safety property. The invariant is constructed 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 interference invariant is the conjunction of each of these clauses. We proceed with the formal definition.

Definition 5 (Split Interference Invariant). *Let $\mathcal{C}$ be a contract. A* split interference invariant *is an interference invariant θ, given by a tuple (L, R, ζ, μ, ξ), 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,*

$$\theta(s, \mathbf{u}) := \psi_{\text{Roles}}(s, \mathbf{u}) \wedge \psi_{\text{Lits}}(s, \mathbf{u}) \wedge \psi_{\text{Else}}(s, \mathbf{u})$$

$$\psi_{\text{Lits}}(s, \mathbf{u}) := \bigwedge_{i=0}^{m-1} [\mathit{id}(\mathbf{u}_0) = l_i] \Rightarrow \zeta_i(s, \mathbf{u})$$

$$\psi_{\text{Roles}}(s, \mathbf{u}) := \bigwedge_{i=0}^{n-1} [\mathit{id}(\mathbf{u}_0) = \mathit{role}(s, r_i)] \Rightarrow \mu_i(s, \mathbf{u})$$

$$\psi_{\text{Else}}(s, \mathbf{u}) := \left[\bigwedge_{i=0}^{m-1} \mathit{id}(\mathbf{u}_0) \neq l_i\right] \wedge \left[\bigwedge_{i=0}^{n-1} \mathit{id}(\mathbf{u}_0) \neq \mathit{role}(s, r_i)\right] \Rightarrow \xi(s, \mathbf{u})$$

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

In Example 2, we illustrate how Def. 5 is applied in practice.

Example 2. To establish φ_1 from Example 1, we construct a split interference invariant θ_1, defined by the tuple $\mathit{Inv} = (L_2, \varnothing, \varnothing, (\xi_1), \xi_2)$, where $L_2 = \{0\}$, ξ_1 is as defined in Example 1, and $\xi_2(s, \mathbf{u}) := (\mathsf{map}(\mathbf{u}_0))_1 \geq 0$. The two instances of $\varnothing$ in Inv correspond to the role constraints and their associated user summaries. If Inv is related back to Def. 5, it follows that $\psi_{\text{Roles}}(s, \mathbf{u}) := \top$, $\psi_{\text{Lits}}(s, \mathbf{u}) := (\mathsf{id}(\mathbf{u}_0) = 0) \Rightarrow ((\mathsf{map}(\mathbf{u}_0))_1 = 0)$, and $\psi_{\text{Else}}(s, \mathbf{u}) := (\mathsf{id}(\mathbf{u}_0) \neq 0) \Rightarrow ((\mathsf{map}(\mathbf{u}_0))_1 \geq 0)$. Expanding θ_1 yields,

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

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). $\square$

5.2 Localizing a Smart Contract Bundle

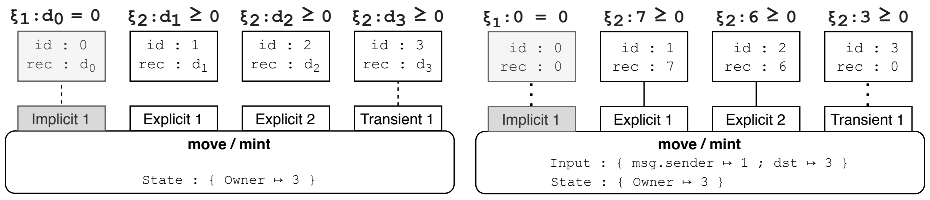
A local (smart contract) bundle is a finite-state abstraction of a smart contract bundle. This abstraction reduces smart-contract verification to finite-state model checking[8]. At a high level, each local bundle is a non-deterministic LTS and is constructed from three components: a smart contract, a candidate interference invariant, and a neighbourhood. We use the term *candidate interference invariant* to describe any predicate that has the structure of an interference invariant, regardless of its semantic interpretation. The term *neighbourhood* describes a set of users, where each user is represented by their respective address.

Let $\mathcal{A}$ be an N-user neighbourhood and θ be a candidate interference invariant. The local bundle corresponding to $\mathcal{A}$ and θ is defined using special user configurations called *local user configurations*. An N-user 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 6 (Local User Configuration). *Let* $\mathcal{C}$ *be a contract,* $\mathcal{A} \subseteq \mathbb{A}$ *be an* N*-user neighbourhood,* θ *be a candidate interference invariant for* $\mathcal{C}$*,* $s \in \mathit{control}(\mathcal{C})$*, and* $\mathbf{u} \in \mathit{config}(\mathcal{C}, N)$*. A user configuration* $\mathbf{u}$ *is* local *w.r.t.* $\mathcal{A}$*,* θ*, and* s*, written* $\mathit{IsLocal}(\mathbf{u}, \mathcal{A}, \theta, s)$*, if* $\forall a \in \mathcal{A} \cdot \exists i \in [N] \cdot \mathit{id}(\mathbf{u}_i) = a \wedge (s, \mathbf{u}_i) \in \theta$*.*

Each state of the *local bundle* for $\mathcal{A}$ and θ is a tuple $(s, \mathbf{u})$, where s is a control state and $\mathbf{u}$ is an 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, $(s', \mathbf{u}')$ is a successor of $(s, \mathbf{u})$ if there exists a *local* user configuration $\mathbf{v}$ of s such that $(s', \mathbf{u}') = f(s, \mathbf{v})$.

[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.

Definition 7 (Local Bundle). *Let $\mathcal{C}$ be a contract, $\mathcal{A} = \{a_0, \ldots, a_{N-1}\} \subseteq \mathbb{A}$ be an N-user neighbourhood and θ be a candidate interference invariant for $\mathcal{C}$. A* local bundle *is an LTS $\mathit{local}(\mathcal{C}, \mathcal{A}, \theta) = (S, P, \hat{f}, s_0)$, where $S \subseteq \mathit{control}(\mathcal{C}) \times \mathcal{N}$, $\mathcal{N} = \mathit{config}(\mathcal{C}, N)$, $P = \mathit{inputs}(\mathcal{C})$, $s_0 = (c_0, \mathbf{u})$, $c_0 = (\mathbf{0}, \mathbf{0})$, $\forall i \in [N] \cdot \mathit{id}(\mathbf{u}_i) = a_i \wedge \mathit{map}(\mathbf{u}_i) = \mathbf{0}$, and $\hat{f}$ is $\hat{f}(s, \mathbf{u}, p) = \{[\![tr]\!]_N(s, \mathbf{v}, p) \mid \mathbf{v} \in \mathcal{N} \wedge \mathit{IsLocal}(\mathbf{v}, \mathcal{A}, \theta, s)\}$.*

Example 3. We briefly illustrate $\hat{f}$ of Def. 7. Let $\mathcal{A}_1 = \{0, 1, 2, 3\}$ be a neighbourhood, and θ_1 be as in Example 2, with $(S, P, \hat{f}, s_0) = \mathsf{local}(\mathcal{C}, \mathcal{A}_1, \theta_1)$. Consider a state $(s, \mathbf{u}) \in S$, where $s = \{\texttt{owner} \mapsto \texttt{address(3)}\}$ and $\forall i \in [4] \cdot \mathsf{map}(\mathbf{u}_i) = 0$. Now assume that `move` has been executed with clients `address(1)` and `address(2)`. A successor state of $(s, \mathbf{u})$ must be determined. By definition, $(s', \mathbf{u}') \in \hat{f}(s, \mathbf{u}, p) = [\![tr]\!]_N(s, \mathbf{v}, p)$, 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 configuration 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 $\xi_1(s, \mathbf{v}_0)$, which 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, $\mathsf{map}(\mathbf{u}'_0) = 0$, $\mathsf{map}(\mathbf{u}'_1) = 6$, $\mathsf{map}(\mathbf{u}'_2) = 7$, and $\mathsf{map}(\mathbf{u}'_3) = 0$, whereas the control state remains unchanged. This is a successor state, as desired. □

Example 3 motivates an important insight regarding interference 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 8 (Saturating Neighbourhood). *Let $\mathcal{C}$ be a contract, (G, ρ, τ) be a PTG of $\mathcal{C}$, and $G = (V, E, L)$. A* saturating neighbourhood *for (G, ρ, τ) is a tuple $(\mathcal{A}_{\text{Persist}}, \mathcal{A}_{\text{Trans}}, \mathcal{A}_{\text{Reps}})$ s.t. $\mathcal{A}_{\text{Persist}}, \mathcal{A}_{\text{Trans}}, \mathcal{A}_{\text{Reps}} \subseteq \mathbb{A}$ and*

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

A saturating neighbourhood can be used to reduce compositionality and parameterized safety proofs to the safety of a local bundle. We start with compositionality. By definition, every candidate interference 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, in a sufficiently large neighbourhood—say $\mathcal{A}^+$—that the converse is also true. Informally, the proof requires a unique user for each participation class. This is satisfied provided $\mathcal{A}^+$ contains a saturating neighbourhood. The proof also requires 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 2.

Theorem 2. *Let $\mathcal{C}$ be an order-oblivious contract, θ be a candidate split interference invariant for $\mathcal{C}$, (G, ρ, τ) be a PTG for $\mathcal{C}$, $(\mathcal{A}_{\text{Persist}}, \mathcal{A}_{\text{Trans}}, \mathcal{A}_{\text{Reps}})$ be a saturating neighbourhood of (G, ρ, τ), $\mathcal{A}_\theta$ be the literal guards of θ, $\mathcal{A} := \mathcal{A}_{\text{Persist}} \cup \mathcal{A}_{\text{Trans}} \cup \mathcal{A}_{\text{Reps}} \cup \mathcal{A}_\theta$ 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 interference invariant for $\mathcal{C}$.*

Next, we present our main result: a sound proof rule for k-universal safety. As Theorem 2, it uses a saturating neighborhood $\mathcal{A}^+$. However, the established safety property is k-universal rather than 1-universal. Thus, $\mathcal{A}^+$ must have at least k interchangeable users.

Theorem 3. *Let $\mathcal{C}$ be an order-oblivious contract, θ be a split interference invariant for $\mathcal{C}$, (G, ρ, τ) be a PTG for $\mathcal{C}$, $(\mathcal{A}_{\text{Persist}}, \mathcal{A}_{\text{Trans}}, \mathcal{A}_{\text{Reps}})$ be a saturating neighbourhood of (G, ρ, τ), and φ be a guarded universal k-safety property for $\mathcal{C}$ with literal guards $\mathcal{A}_\varphi$. Define $\mathcal{A} := \mathcal{A}_{\text{Persist}} \cup \mathcal{A}_{\text{Trans}} \cup \mathcal{A}_{\text{Reps}} \cup \mathcal{A}_\varphi$ and $\mathcal{A}^+ \subseteq \mathbb{A}$ such that $\mathcal{A} \subseteq \mathcal{A}^+$ and $|\mathcal{A}^+| = |\mathcal{A}| + \max(0, |\mathcal{A}_{\text{Reps}}| - k)$. If $\mathsf{local}(\mathcal{C}, \mathcal{A}^+, \theta) \models \varphi$, then $\forall N \in \mathbb{N} \cdot \mathsf{lts}(\mathcal{C}, N) \models \varphi$.*

We use Theorem 3 to complete Example 2. Recall φ_1 (which is 1-universal), θ_1, and $\mathcal{A}_1$ from Example 3. To apply Theorem 3, $\mathcal{A}_1$ must be extended to $\mathcal{A}_1^+$, This is not hard as φ_1 is 1-universal, and as `address(0)` (the only literal guard) is already in $\mathcal{A}_1$. For example, $\mathcal{A}_1^+ = \mathcal{A} \cup \{5\}$ is sufficient. Using an unbounded, finite state model checker, $\mathsf{local}(\mathcal{C}, \mathcal{A}_1^+, \theta_1) \models \varphi_1$ can be proven and certified by an invariant θ_1^* that blocks all bad states. By Theorem 3, the parameterized safety of $\mathcal{C}$ must follow. Even more remarkably, θ_1^* must be the inductive invariant from Sec. 2, as it summarizes the safe control states that are closed under the interference of θ_1. Thus, *we reduced verification of parameterized smart contracts to finite state sequential verification!*

6 Implementation and Evaluation

We have implemented the technique described in this paper in an open-source tool called SmartACE, which 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, constructing the PTG), (3) generates a local bundle and models it using LLVM IR, and (4) verifies the model using SeaHorn [14]. In this section, we report on the effectiveness of SmartACE for analyzing real-world smart-contracts. A detailed description of the SmartACE architecture and of the case studies is outside the scope of this paper. Both the case studies and SmartACE are publicly available at `https://github.com/contract-ace`. Our evaluation focuses on the following three research questions:

RQ1: Compliance. Does MiniSol represent real-world smart contracts?
RQ2: Effectiveness. Is SmartACE effective for MiniSol smart contracts?
RQ3: Performance. Is SmartACE competitive with other techniques?

Benchmarks and Setup. To answer the above research questions, we used a benchmark of 53 properties across 10 smart contracts (shown in Tab. 1). Contracts `Alchemist` to `Ziliqa` are from VerX [29]. Contracts `Fund` and `Auction` were added to offset the lack of parameterized properties in existing benchmarks. The `QSPStaking` contract comprises the QuantStamp Assurance Protocol[9] for which we checked properties provided by QuantStamp. Some properties require additional instrumentation techniques (i.e., temporal [29] and aggregate [16] properties). In Tab. 1, *Inv. Size* is the clause size of an interference invariant manually provided to SmartACE and *Users* is the number of users computed by `PTGBuilder`. All experiments were done on an Intel® Core i7® CPU @ 2.8GHz 4-core machine with 16GB of RAM on Ubuntu 18.04.

RQ1: Compliance. To assess whether the restrictions of MiniSol are reasonable, we determined the percentage of *compliant* VerX benchmarks. We found that 54% of the contracts were compliant after removing dead code (e.g., unused library imports). From the unsupported contracts, 5 were loosely-coupled, 1 made use of dynamic arrays, and none were order-aware.

[9] `https://github.com/quantstamp/qsp-staking-protocol`

Contracts			SmartACE			VerX
Name	Prop.	LOC	Time (s)	Inv. Size	Users	Time (s)
Alchemist	3	401	7	0	7	71
ERC20	9	599	12	1	5	159
Melon	16	462	30	0	7	408
MRV	5	868	2	0	7	887
Overview	4	66	3	0	8	212
PolicyPal	4	815	8	0	8	20,773
Ziliqa	5	377	8	0	7	377
Fund	2	38	1	0	6	N/A
Auction	1	42	1	1	5	N/A
QSPStaking	4	1,550	3	7	8	N/A

Table 1: Experimental results for SmartACE on various benchmarks.

RQ2: Effectiveness. To assess the effectiveness of SmartACE, we determine the percentage of properties that could be verified from (compliant) VerX contracts. We found that 100% of these properties can be verified, but also discovered that many of these properties were not parameterized. To validate SmartACE against parameterized properties, an additional case study was carried out using `Fund` and `Auction`, as informally described on the SeaHorn development blog[10]. To validate SmartACE in the context of large-scale smart-contract development, we performed another case study to verify properties of `QSPStaking`. In this study, 4 QuantStamp specifications were selected (at random from a large specification provided by QuantStamp) and validated. It required two person days to model the environment, and one person day to discover the interference invariant. The major overhead in modeling the environment came from manual abstraction of unbounded arrays. The discovery of the interference invariant was semi-automatic, and aided by counter-examples produced by SeaHorn. We conclude that SmartACE is suitable for smart contracts requiring high-assurance, and with proper automation, can be integrated into smart-contract development.

RQ3: Performance. To evaluate the performance of SmartACE, we compared its verification time to the reported times[11] of VerX, a state-of-the-art verification tool. In each case, SmartACE significantly outperformed VerX, achieving a speedup of at least 10x. We hypothesize that one bottleneck is the number of users (which requires more state). A more fine-grained participation analysis could further reduce the number of users. Upon manual inspection of `Melon` and `Alchemist` (in a single bundle), it was found that user state could be reduced by 28%. We conclude that SmartACE has the potential to scale.

[10] `http://seahorn.github.io/blog/`
[11] VerX is closed source, so a direct comparison is impossible.

7 Related Work

In recent years, the program-analysis community has developed many tools for smart-contract analysis: these range from dynamic analysis [18,38] to static analysis [23,25,1,22,35,12,27,21,36,5] and verification [19,16,37,24,29,34]. The latter are most related to SmartACE since their focus is on verifying functional correctness, as opposed to generic rules (e.g., the absence of reentrancy [13] and arithmetic overflows). Existing techniques for functional correctness are either deductive, and require that most invariants be provided manually (i.e., [16,37]), or are automated but neglect the parameterized nature of smart contracts (i.e., SMTChecker in the Solidity compiler and [24,29,34]). The tools that do acknowledge parameterization provide static analysis [21,5]. In contrast, SmartACE introduces a novel local-reasoning technique that verifies parameterized safety properties with higher automation than deductive techniques.

More generally, parameterized systems form a rich field of research, as outlined in [4]. The use of SCUNs was first proposed in [11], and many other models exist for both synchronous and asynchronous systems (e.g., [9,32,33]). The approach of PCMC is not the only compositional solution for parameterized verification. For instance, environmental abstraction [6] considers a process and its environment, similar to the inductive and interference invariants of SmartACE. Other approaches [30,10] generalize from small instances through the use of ranking functions. The use of abstract interpretation has also been shown useful in finding parameterized invariants [2]. The addresses used in our analysis are similar to the scalarsets of [17]. Most compositional techniques require cutoff analysis—considering network instances up to a given size [7,20]. Local bundles avoid explicit cutoff analysis by simulating all smaller instances, which is similar to existing work on bounded parameterized model checking [8]. SmartACE is the first application of local reasoning in the context of smart contracts.

8 Conclusions

In this paper, we present the first study of local reasoning for smart contracts. We show that, for many smart contracts, parameterized safety can be reduced to finite-state safety. We have implemented SmartACE to evaluate the technique. SmartACE is built upon a novel model for smart contracts, in which users are processes and communication is explicit. In this model, communication can be over-approximated by static analysis, and the results are sufficient to satisfy the uniformity condition in the local-reasoning proof rule.

There are several directions for future work. First, we are interested in automating the discovery of interference invariants. We believe that the techniques used by SeaHorn (i.e. Constrained Horn Clause solving) can be extended to compute such invariants. Second, we intend to relax the restrictions of MiniSol, as motivated by some of the VerX benchmarks (i.e., to loosely coupled contracts and unbounded arrays). Third, our experience using SmartACE to prove aggregate properties indicates potential for a richer theory of local aggregates.

References

1. Mythril, `https://github.com/ConsenSys/mythril`
2. Abdulla, P.A., Haziza, F., Holík, L.: Parameterized verification through view abstraction. STTT **18**(5), 495–516 (2016)
3. Allen, F.E.: Control flow analysis. In: Symposium on Compiler Optimization. pp. 1–19. ACM (1970)
4. Bloem, R., Jacobs, S., Khalimov, A.: Decidability of Parameterized Verification. Morgan & Claypool Publishers (2015)
5. Brent, L., Grech, N., Lagouvardos, S., Scholz, B., Smaragdakis, Y.: Ethainter: A smart contract security analyzer for composite vulnerabilities. In: PLDI. pp. 454–469. ACM (2020)
6. Clarke, E.M., Talupur, M., Veith, H.: Environment abstraction for parameterized verification. In: VMCAI. LNCS, vol. 3855, pp. 126–141. Springer (2006)
7. Emerson, E.A., Namjoshi, K.S.: Reasoning about rings. In: POPL. p. 85–94. ACM (1995)
8. Emerson, E.A., Trefler, R.J., Wahl, T.: Reducing model checking of the few to the one. In: ICFEM. LNCS, vol. 4260, pp. 94–113. Springer (2006)
9. Esparza, J., Ganty, P., Majumdar, R.: Parameterized verification of asynchronous shared-memory systems. In: CAV. LNCS, vol. 8044, pp. 124–140. Springer (2013)
10. Fang, Y., Piterman, N., Pnueli, A., Zuck, L.D.: Liveness with invisible ranking. In: VMCAI. LNCS, vol. 2937, pp. 223–238. Springer (2004)
11. German, S.M., Sistla, A.P.: Reasoning about systems with many processes. J. ACM **39**(3), 675–735 (Jul 1992)
12. Grech, N., Kong, M., Jurisevic, A., Brent, L., Scholz, B., Smaragdakis, Y.: MadMax: Surviving out-of-gas conditions in Ethereum smart contracts. PACMPL **2**, 116:1–116:27 (2018)
13. Grossman, S., Abraham, I., Golan-Gueta, G., Michalevsky, Y., Rinetzky, N., Sagiv, M., Zohar, Y.: Online detection of effectively callback free objects with applications to smart contracts. PACMPL **2**, 48:1–48:28 (2018)
14. Gurfinkel, A., Kahsai, T., Komuravelli, A., Navas, J.A.: The SeaHorn verification framework. In: CAV. LNCS, vol. 9206, pp. 343–361. Springer (2015)
15. Gurfinkel, A., Shoham, S., Meshman, Y.: SMT-based verification of parameterized systems. In: FSE. pp. 338–348. ACM (2016)
16. Hajdu, Á., Jovanovic, D.: solc-verify: A modular verifier for Solidity smart contracts. In: VSTTE. LNCS, vol. 12031, pp. 161–179. Springer (2019)
17. Ip, C.N., Dill, D.L.: Better verification through symmetry. In: IFIP WG10.2 International Conference on Computer Hardware Description Languages and their Applications. IFIP Transactions, vol. A-32, pp. 97–111. North-Holland (1993)
18. Jiang, B., Liu, Y., Chan, W.K.: ContractFuzzer: Fuzzing smart contracts for vulnerability detection. In: ASE. pp. 259–269. ACM (2018)
19. Kalra, S., Goel, S., Dhawan, M., Sharma, S.: ZEUS: Analyzing safety of smart contracts. In: NDSS. The Internet Society (2018)
20. Khalimov, A., Jacobs, S., Bloem, R.: Towards efficient parameterized synthesis. In: VMCAI. LNCS, vol. 7737, pp. 108–127. Springer (2013)
21. 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)
22. Krupp, J., Rossow, C.: teEther: Gnawing at Ethereum to automatically exploit smart contracts. In: USENIX Security. pp. 1317–1333. USENIX (2018)

23. Luu, L., Chu, D., Olickel, H., Saxena, P., Hobor, A.: Making smart contracts smarter. In: CCS. pp. 254–269. ACM (2016)
24. Mavridou, A., Laszka, A., Stachtiari, E., Dubey, A.: VeriSolid: Correct-by-design smart contracts for Ethereum. In: FC. LNCS, vol. 11598, pp. 446–465. Springer (2019)
25. Mossberg, M., Manzano, F., Hennenfent, E., Groce, A., Grieco, G., Feist, J., Brunson, T., Dinaburg, A.: Manticore: A user-friendly symbolic execution framework for binaries and smart contracts. In: ASE. pp. 1186–1189. IEEE (2019)
26. Namjoshi, K.S., Trefler, R.J.: Parameterized compositional model checking. In: TACAS. LNCS, vol. 9636, pp. 589–606. Springer (2016)
27. Nikolic, I., Kolluri, A., Sergey, I., Saxena, P., Hobor, A.: Finding the greedy, prodigal, and suicidal contracts at scale. In: ACSAC. pp. 653–663. ACM (2018)
28. Owicki, S.S., Gries, D.: An axiomatic proof technique for parallel programs I. Acta Informatica **6**, 319–340 (1976)
29. Permenev, A., Dimitrov, D., Tsankov, P., Drachsler-Cohen, D., Vechev, M.: VerX: Safety verification of smart contracts. In: S&P. pp. 1661–1677. IEEE (2020)
30. Pnueli, A., Ruah, S., Zuck, L.D.: Automatic deductive verification with invisible invariants. In: TACAS. LNCS, vol. 2031, pp. 82–97. Springer (2001)
31. Sergey, I., Hobor, A.: A concurrent perspective on smart contracts. In: FC. LNCS, vol. 10323, pp. 478–493. Springer (2017)
32. Siegel, S.F., Avrunin, G.S.: Verification of MPI-based software for scientific computation. In: SPIN. LNCS, vol. 2989, pp. 286–303. Springer (2004)
33. Siegel, S.F., Gopalakrishnan, G.: Formal analysis of message passing. In: VMCAI. LNCS, vol. 6538, pp. 2–18. Springer (2011)
34. So, S., Lee, M., Park, J., Lee, H., Oh, H.: VERISMART: A highly precise safety verifier for Ethereum smart contracts. In: S&P. pp. 1678–1694. IEEE (2020)
35. Tsankov, P., Dan, A.M., Drachsler-Cohen, D., Gervais, A., Bünzli, F., Vechev, M.T.: Securify: Practical security analysis of smart contracts. In: CCS. pp. 67–82. ACM (2018)
36. Wang, S., Zhang, C., Su, Z.: Detecting nondeterministic payment bugs in Ethereum smart contracts. PACMPL **3**, 189:1–189:29 (2019)
37. 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, vol. 12031, pp. 87–106. Springer (2019)
38. Wüstholz, V., Christakis, M.: Harvey: A greybox fuzzer for smart contracts. In: FSE. ACM (2020), to appear
39. Zhong, J.E., Cheang, K., Qadeer, S., Grieskamp, W., Blackshear, S., Park, J., Zohar, Y., Barrett, C.W., Dill, D.L.: The Move prover. In: CAV. LNCS, vol. 12224, pp. 137–150. Springer (2020)

A The Syntax of MiniSol

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

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. 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.

In Sec. 3, we also introduced two primitive domains: $\mathbb{A}$ and $\mathbb{D}$. In reality, $\mathbb{D}$ consists of several sets, depending on the type of the non-address value. 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 $\mathcal{C} :=$ `Auction`, from Fig. 1. We highlight some practical issues that were overlooked in Sec. 3. Note that in this example, we no longer assume that Fig. 1 consists of a single transaction. This means that the transaction input must also specify the transaction name. For this we define the set $\text{Tx} := \{$`constructor`, `bid`, `withdraw`, `close`$\}$.

The control states 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 $(a_1, a_2, d_1, d_2, \text{aux}_1, \text{aux}_2) \in \mathsf{control}(\mathcal{C})$, a_1, a_2, d_1, and d_2 correspond to `owner`, `topBidder`, `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 must be executed iff $\text{aux}_1 =$ **`false`**.

The user states are $\mathsf{user}(\mathcal{C}) \subseteq \mathbb{A} \times \mathbb{N}_{256} \times \mathbb{N}_{256}$. 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 $\mathsf{inputs}(\mathcal{C}) \subseteq \text{Tx} \times \mathbb{A} \times \mathbb{N}_{256} \times \mathbb{N}_{256}$. 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 **`now`**. If $t =$ **`constructor`**, then y_2 represents `duration`. If $t =$ `bid`, then y_2 represents **`msg.value`**. In all other cases, d_2 is unused.

For readability, we give transactional semantics for $f = [\![tr]\!]_N$. In Sec. 3, it was mentioned that $[\![tr]\!]_N$ can be reduced to internal and synchronized transitions. However, the existence of such a reduction is only important for proofs.

$$f(s, \mathbf{u}, (t, x, y_1, y_2)) = \begin{cases} (s, \mathbf{u}) & \text{if } y_1 < \mathsf{data}(s, 2) \\ g_1(s, \mathbf{u}, x, y_1, y_2) & \text{if } y_1 \geq \mathsf{data}(s, 2) \wedge t = \texttt{constructor} \\ g_2(s, \mathbf{u}, x, y_1, y_2) & \text{if } y_1 \geq \mathsf{data}(s, 2) \wedge t = \texttt{bid} \\ g_2(s, \mathbf{u}, x, y_1) & \text{if } y_1 \geq \mathsf{data}(s, 2) \wedge t = \texttt{withdraw} \\ g_4(s, \mathbf{u}, x, y_1) & \text{if } y_1 \geq \mathsf{data}(s, 2) \wedge t = \texttt{close} \end{cases}$$

$$g_1(s, \mathbf{u}, (t, x, y_1, y_2)) = \begin{cases} (s, \mathbf{u}) & \text{if } \mathsf{data}(s, 3) \neq \mathsf{false} \\ (s', \mathbf{u}) & \text{if } \mathsf{data}(s, 3) = \mathsf{false} \wedge \mathsf{role}(s', 0) = x \\ & \wedge \mathsf{role}(s', 1) = \mathsf{role}(s, 1) \wedge \mathsf{data}(s', 0) = y_1 + y_2 \\ & \wedge \mathsf{data}(s', 1) = \mathsf{data}(s, 1) \wedge \mathsf{data}(s', 2) = y_1 \\ & \wedge \mathsf{data}(s', 3) = 1 \end{cases}$$

In this model a reverted transaction is treated as a no-op. The first line in f guards against non-monotonic time, as $\mathsf{data}(s, 2)$ is the last time. The first line in g_1 guards against double construction, as $\mathsf{data}(s, 3)$ is the constructor flag. The second line of g_1 is verbose, but essentially says that all state variables remain unchanged, except for: (1) `owner` which is now **`msg.sender`**, (2) `endTime` which is now **`now`** `+ duration`, (3) the time auxiliary variable which is set to **`now`**, and (4) the constructor flag which is now raised. The functions g_2 to g_4 are defined similarly.

C Order-Oblivious Communication

Recall from Sec. 4 that 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[12]. 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) $\mathsf{map}(\mathbf{u}'_i) = \mathsf{map}(\mathbf{u}_i)$. 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 9 (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 $\mathit{Topology}(\mathcal{C}, N, p) = \langle \mathit{Explicit}, \mathit{Transient}, \mathit{Implicit} \rangle$. Then, the transaction f_p is* order-oblivious *if for every pair of states $s, t \in S$ and every $x, y \in \mathbb{A} \setminus \mathit{Implicit}$ 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 rder-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).

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

D Uniformity and Participation

This justifies local reasoning. Sketch for the time being.

1. We have an order oblivious contract by assumption.
2. From the PT we can bound the influence of each transaction to a neighbourhood.
3. The number of user equivalent classes under the address similar equivalence class is finite (even if we assume unbounded addresses).
4. By order-obliviousness, this is a sound abstraction.