Compositional Verification of Smart Contracts Through Communication Abstraction*

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. Solidity smart contracts are programs that manage up to 2^{160} users on a blockchain. Verifying a smart contract relative to all users is intractable due to state explosion. Existing solutions either restrict the number of users to under-approximate behaviour, or rely on manual proofs. In this paper, we present *local bundles* that reduce contracts with arbitrarily many users to sequential programs with a few *representative* users. Each representative user abstracts concrete users that are locally symmetric to each other relative to the contract and the property. Our abstraction is semi-automated. The representatives depend on communication patterns, and are computed via static analysis. A summary for the behaviour of each representative is provided manually, but a default summary is often sufficient. Once obtained, a local bundle is amenable to sequential static analysis. We show that local bundles are relatively complete for parameterized safety verification, under moderate assumptions. We implement local bundle abstraction in SmartACE, and show order-of-magnitude speedups compared to a state-of-the-art verifier.

1 Introduction

Solidity smart contracts are distributed programs that facilitate information flow between users. Users alternate and execute predefined transactions, that each terminate within a predetermined number of steps. Each user (and contract) is assigned a unique, 160-bit address, that is used by the smart contract to map the user to that user's data. In theory, smart contracts are finite-state systems with 2^{160} users. However, in practice, the state space of a smart contract is huge—with at least $2^{2^{160}}$ states to accommodate all users and their data (conservatively counting one bit per user). In this paper, we consider the challenge of automatically verifying Solidity smart contracts that rely on user data.

A naive solution for smart contract verification is to verify the finite-state system directly. However, verifying systems with at least $2^{2^{160}}$ states is intractable.

* This work was supported, in part, by Individual Discovery Grants from the Natural Sciences and Engineering Research Council of Canada, and Ripple Fellowship. Jorge A. Navas was supported by NSF grant 1816936.

```
contract Auction {
  mapping(address => uint) bids;
  address manager; uint leadingBid; bool stopped;
  uint _sum;

  constructor(address mgr) public { manager = mgr; }

  function bid(uint amount) public {
    require(msg.sender != manager);
    require(amount > leadingBid);
    require(!stopped);
    _sum = _sum + amount - bids[msg.sender];
    bids[msg.sender] = amount;
    leadingBid = amount;
  }

  function withdraw() public {
    require(msg.sender != manager);
    require(bids[msg.sender] != leadingBid);
    require(!stopped);
    _sum = _sum + 0 - bids[msg.sender];
    bids[msg.sender] = 0;
  }

  function stop() public {
    require(msg.sender == manager);
    stopped = true;
  }
}
```

Fig. 1: A smart contract that implements a simple auction.

```
Auction _a = new Auction(address(2));
_a.address = address(1);

while (true) {
  // Applies an interference invariant.
  uint bid = *;
  uint maxBid = _a.leadingBid;
  require(bid <= maxBid);
  require(bid == maxBid || bid + maxBid <= _a.sum);
  _a.bids[address(3)] = bid;
  // Selects a sender.
  msg.sender = *;
  require(msg.sender > address(1));
  require(msg.sender < address(5));
  require(msg.sender < address(4));
  // Selects a call.
  if      (*) _a.bid(*);
  else if (*) _a.withdraw();
  else if (*) _a.stop();
}
```

Fig. 2: A harness to verify **Prop. 1** (ignore the highlighted lines) and **Prop. 2**.

The naive solution fails because the state space is exponential in the number of users. Instead, we infer correctness from a small number of representative users to ameliorate state explosion. To restrict a contract to fewer users, we first generalize to a *family* of finite-state systems parameterized by the number of users. In this way, smart contract verification is reduced to parameterized verification.

For example, consider `Auction` in Fig. 1 (for now, ignore the highlighted lines). In `Auction`, each user starts with a bid of 0. Users alternate, and submit increasingly larger bids, until a designated manager stops the auction. While the auction is not stopped, a non-leading user may withdraw their bid[5]. `Auction` satisfies **Prop. 1**: "*Once* `stop()` *is called, all bids are immutable.*" **Prop. 1** is satisfied since `stop()` sets `stopped` to true, no function sets `stopped` to false, and while `stopped` is true neither `bid()` nor `withdraw()` is enabled. Formally, **Prop. 1** is initially true, and remains true due to **Prop. 1b**: "*Once* `stop()` *is called,* `stopped` *remains true.*" **Prop. 1** is said to be inductive relative to its *inductive strengthening* **Prop. 1b**. A *Software Model Checker (SMC)* can establish **Prop. 1** by an exhaustive search for its inductive strengthening. However, this requires a bound on the number of addresses, since a search with all 2^{160} addresses is intractable.

A bound of at least four addresses is necessary to represent the zero-account (i.e., a null user that cannot send transactions), the smart contract account, the manager, and an arbitrary sender. However, once the arbitrary sender submits a bid, the sender is now the leading bidder, and cannot withdraw its bid. To enable `withdraw()`, a fifth user is required. It follows by applying the results of [20], that a bound of five addresses is also sufficient, since users do not read each other's bids, and adding a sixth user does not enable additional changes to

[5] For simplicity of presentation, we do not use Ether, Ethereum's native currency.

leadingBid [20]. The bounded system, known as a harness, in Fig. 2 assigns the zero-account to address 0, the smart contract account to address 1, the manager to address 2, the arbitrary senders to addresses 3 and 4, and then executes an unbounded sequence of arbitrary function calls. Establishing **Prop. 1** on the harness requires finding its inductive strengthening. A strengthening such as **Prop. 1b** (or, in general, a counterexample violating **Prop. 1**) can be found by an SMC, directly on the harness code.

The above bound for **Prop. 1** also works for checking all control-reachability properties of Auction. This, for example, follows by applying the results of [20]. That is, Auction has a *Small Model Property (SMP)* (e.g., [20,1]) for such properties. However, not all contracts enjoy an SMP. Consider **Prop. 2**: "*The sum of all active bids is at least* leadingBid." Auction satisfies **Prop. 2** since the leading bid is never withdrawn. To prove Auction satisfies **Prop. 2**, we instrument the code to track the current sum, through the highlighted lines in Fig. 1. With the addition of _sum, Auction no longer enjoys an SMP. Intuitively, each user enables new combinations of _sum and leadingBid. As a proof, assume that there are N users (other than the zero-account, the smart contract account, and the manager) and let $S_N = 1 + 2 + \cdots + N$. In every execution with N users, if leadingBid is $N+1$, then _sum is less than S_{N+1}, since active bids are unique and S_{N+1} is the sum of $N+1$ bids from 1 to $N+1$. However, in an execution with $N+1$ users, if the i-th user has a bid of i, then leadingBid is $N+1$ and _sum is S_{N+1}. Therefore, increasing N extends the reachable combinations of _sum and leadingBid. For example, if $N = 2$, then $S_3 = 1+2+3 = 6$. If the leading bid is 3, then the second highest bid is at most 2, and, therefore, _sum $\leq 5 < S_3$. However, when $N = 3$, if the three active bids are $\{1, 2, 3\}$, then _sum is S_3. Therefore, instrumenting Auction with _sum violates the SMP of the original Auction.

Despite the absence of such an SMP, each function of Auction interacts with at most one user per transaction. Each user is classified as either the zero-account, the smart contract, the manager, or an arbitrary sender. In fact, all arbitrary senders are indistinguishable with respect to **Prop. 2**. For example, if there are exactly three active bids, $\{2, 4, 8\}$, it does not matter which user placed which bid. The leading bid is 8 and the sum of all bids is 14. On the other hand, if the leading bid is 8, then each participant of Auction must have a bid in the range of 0 to 8. To take advantage of these classes, rather than analyze Auction relative to all 2^{160} users, it is sufficient to analyze Auction relative to a representative user from each class. In our running example, there must be representatives for the zero-account, the smart contract account, the manager, and an (arbitrary) sender. The key idea is that each representative user can correspond to one or *many* concrete users.

Intuitively, each representative user summarizes the concrete users in its class. If a representative's class contains a single concrete user, then there is no difference between the concrete user and the representative user. For example, the zero-account, the smart contract account, and the manager each correspond to single concrete users. The addresses of these users, and in turn, their bids, are known with absolute certainty. On the other hand, there are many arbitrary senders. Since senders are indistinguishable from each other, the precise address of the representative sender is unimportant. What matters is that the representative sender does not share an address with the zero-account, the smart contract account, nor the manager. However, this means that at the start of each transaction the location of the representative sender is not absolute, and, therefore, the sender has a range of possible bids. To account for this, we introduce a predicate that is true of all initial bids, and holds inductively across all transactions. We provide this predicate manually, and use it to over-approximate all possible bids. An obvious predicate for `Auction` is that all bids are at most `leadingBid`, but this predicate is not strong enough to prove **Prop. 2**. For example, the representative sender could first place a bid of 10, and then (spuriously) withdraw a bid of 5, resulting in a sum of 5 but a leading bid of 10. A stronger predicate, that is adequate to prove **Prop. 2**, is given by θ_U: "*Each bid is at most* `leadingBid`*. If a bid is not* `leadingBid`*, then its sum with* `leadingBid` *is at most* `_sum`."

Given θ_U, **Prop. 2** can be verified by an SMC. This requires a new harness, with representative, rather than concrete, users. The new harness, Fig. 2 (now including the highlighted lines), is similar to the SMP harness in that the zero-account, the smart contract account, and the manager account are assigned to addresses 0, 1, and 2, respectively, followed by an unbounded sequence of arbitrary calls. However, there is now a single sender that is assigned to address 3 (line 15). That is, the harness uses a fixed configuration of representatives in which the fourth representative is the sender. Before each function call, the sender's bid is set to a non-deterministic value that satisfies θ_U (lines 6–10). If the new harness and **Prop. 2** are provided to an SMC, the SMC will find an inductive strengthening such as, "*The leading bid is at most the sum of all bids.*"

The harness in Fig. 2 differs from existing smart contract verification techniques in two ways. First, each address in Fig. 2 is an abstraction of one or more concrete users. Second, `msg.sender` is restricted to a finite address space by lines 13 to 15. If these lines are removed, then an inductive invariant must constrain all cells of `bids`, to accommodate `bids[msg.sender]`. This requires quantified invariants over arrays that is challenging to automate. By introducing lines 13 to 15, a quantifier-free predicate, such as our θ_U, can directly constrain cell `bids[msg.sender]` instead. Adding lines 13–15 makes the contract finite state. Thus, its verification problem is decidable and can be handled by existing SMCs. However, as illustrated by **Prop. 2**, the restriction on each user must not exclude feasible counterexamples. Finding such a restriction is the focus of this paper.

In this paper, we present a new approach to smart contract verification. We construct finite-state abstractions of parameterized smart contracts, known as *local bundles*. A local bundle generalizes the harness in Fig. 2, and is constructed from a set of representatives and their predicates. When a local bundle and a property are provided to an SMC, there are three possible outcomes. First, if a predicate does not over-approximate its representative, a counterexample to the predicate is returned. Second, if the predicates do not entail the property, then a counterexample to verification is returned (this counterexample refutes the proof, rather than the property itself). Finally, if the predicates do entail the property, then an inductive invariant is returned. As opposed to deductive smart contract solutions, our approach finds inductive strengthenings automatically [17,44]. As opposed to other model checking solutions for smart contracts, our approach is not limited to pre- and post-conditions [21], and can scale to 2^{160} users [24].

Key theoretical contributions of this paper are to show that verification with local bundle abstraction is an instance of Parameterized Compositional Model Checking (PCMC) [31] and the automation of the side-conditions for its applicability. Specifically, Theorem 3 shows that the local bundle abstraction is a sound proof rule, and a static analysis algorithm (`PTGBuilder` in Sec. 4) computes representatives so that the rule is applicable. Key practical contributions are the implementation and the evaluation of the method in a new smart contract verification tool SmartACE, using SeaHorn [15] for SMC. SmartACE takes as input a contract and a predicate. Representatives are inferred automatically from the contract, by analyzing the communication in each transaction. The predicate is then validated by SeaHorn, relative to the representatives. If the predicate is correct, then a local bundle, as in Fig. 2, is returned.

The rest of the paper is structured as follows. Sec. 2 reviews parameterized verification. Sec. 3 presents MicroSol, a subset of Solidity with network semantics. Sec. 4 relates user interactions to representatives. We formalize user interactions as *Participation Topologies (PTs)*, and define *PT Graphs (PTGs)* to over-approximate PTs for arbitrarily many users. Intuitively, each PTG over-approximates the set of representatives. We show that a PTG is computable for every MicroSol program. Sec. 5 defines local bundles and proves that our approach is sound. Sec. 6 evaluates SmartACE and shows that it can outperform VerX, a state-of-the-art verification tool, on all but one VerX benchmark.

2 Background

In this section, we briefly recall *Parameterized Compositional Model Checking (PCMC)* [31]. We write $\mathbf{u} = (u_0, \ldots, u_{n-1})$ 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 $\{0, \ldots, n-1\}$.

Labeled Transition Systems. A *labeled transition system (LTS)*, M, is a tuple (S, P, T, s_0), where S is a set of states, P is a set of actions, $T : S \times P \to 2^S$ is a transition relation, and $s_0 \in S$ is an initial state. M is *deterministic* if T is a function, $T : S \times P \to S$. A (finite) *trace* of M is an alternating sequence of states and actions, $(s_0, p_1, s_1, \ldots, p_k, s_k)$, such that $\forall i \in [k] \cdot s_{i+1} \in T(s_i, p_{i+1})$. A state s is *reachable* in M if s is in some trace $(s_0, p_1, \ldots, s_k)$ of M; that is, $\exists i \in [k+1] \cdot s_i = s$. A *safety property* for M is a subset of states (or a predicate[6]) $\varphi \subseteq S$. M satisfies φ, written $M \models \varphi$, if every reachable state of M is in φ.

Many transition systems are parameterized. For instance, a client-server application is parameterized by the number of clients, and an array-manipulating program is parameterized by the number of cells. In both cases, there is a single

[6] 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.

control process that interacts with many *user processes*. Such systems are called *synchronized control-user networks (SCUNs)* [31]. We let N be the number of processes, and $[N]$ be the process identifiers. We consider SCUNs in which users only synchronize with the control process and do not execute code on their own.

An SCUN $\mathcal{N}$ is a tuple $(S_C, S_U, P_I, P_S, T_I, T_S, c_0, u_0)$, where S_C is a set of control states, S_U a set of user states, P_I a set of internal actions, P_S a set of synchronized actions, $T_I : S_C \times P_I \to S_C$ an internal transition function, $T_S : S_C \times S_U \times P_S \to S_C \times S_U$ a synchronized transition function, $c_0 \in S_C$ is the initial control state, and $u_0 \in S_U$ is the initial user state. The semantics of $\mathcal{N}$ are given by a parameterized LTS, $M(N) := (S, P, T, s_0)$, where $S := S_C \times (S_U)^N$, $P := P_I \cup (P_S \times [N])$, $s_0 := (c_0, u_0, \dots, u_0)$, and $T : S \times P \to S$ such that: (1) if $p \in P_I$, then $T((c, \mathbf{u}), p) = (T_I(c, p), \mathbf{u})$, and (2) if $(p, i) \in P_S \times [N]$, then $T((c, \mathbf{u}), (p, i)) = (c', \mathbf{u}')$ where $(c', \mathbf{u}'_i) = T_S(c, \mathbf{u}_i, p)$, and $\forall j \in [N] \backslash \{i\} \cdot \mathbf{u}'_j = \mathbf{u}_j$.

Parameterized Compositional Model Checking (PCMC). Parameterized systems have parameterized properties [16,31]. A *k-universal safety property* [16] 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 $M(N)$ satisfies predicate φ if $\forall N \in \mathbb{N} \cdot M(N) \models \varphi$. For example, **Prop. 1** (Sec. 1) of `SimpleAuction` (Fig. 1) is 1-universal: "*For every user u, if* `stop()` *has been called, then u is immutable.*"

Proofs of k-universal safety employ compositional reasoning, e.g., [2,16,31,33]. Here, we use PCMC [31]. The keys to PCMC are *uniformity*—the property that finitely many neighbourhoods are distinguishable—and a *compositional invariant*—a summary of the reachable states for each equivalence class, that is closed under the actions of every other equivalence class. 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 $c \in \theta_C$, $(c, u) \in \theta_U$, $p \in P_S$, and $(c', u') \in T_S(c, u, p)$, then $c' \in \theta_C$ and $(c', u') \in \theta_U$;

Consecution 2 If $c \in \theta_C$, $(c, u) \in \theta_U$, $p \in P_C$, and $c' = T_I(c, p)$, then $c' \in \theta_C$ and $(c', u) \in \theta_U$;

Non-Interference If $c \in \theta_C$, $(c, u) \in \theta_U$, $(c, v) \in \theta_U$, $u \neq v$, $p \in P_S$, and $(c', u') = T_S(c, u, p)$, then $(c', v) \in \theta_C$.

By PCMC [31], if $\forall c \in \theta_C \cdot \forall \{(c, u_1), \dots, (c, u_k)\} \subseteq \theta_U \cdot \varphi(c, u_1, \dots, u_k)$, then $M \models \varphi$. This is as an extension of Owicki-Gries [33], 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 MicroSol: Syntax and Semantics

This section provides network semantics for MicroSol, a subset of Solidity[7]. Like Solidity, MicroSol is an imperative object-oriented language with built-in communication operations. The syntax of MicroSol is in Fig. 3. MicroSol restricts Solidity to a core subset of communication features. For example, MicroSol does not

[7] `https://docs.soliditylang.org/`

$$
\begin{array}{rl}
\langle \text{FName} \rangle \models & \textit{a valid function name} \\
\langle \text{VName} \rangle \models & \textit{a valid variable name} \\
\langle \text{CName} \rangle \models & \textit{a valid contract name} \\
\langle \text{Literal} \rangle \models & \textit{an integer, Boolean, or address literal} \\
\langle \text{Types} \rangle \models & \texttt{uint} \mid \texttt{bool} \mid \texttt{address} \mid \texttt{mapping(address => uint)} \mid \langle \text{CName} \rangle \\
\langle \text{Operator} \rangle \models & \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 \models & \langle \text{Literal} \rangle \mid \langle \text{VName} \rangle \mid \texttt{this} \mid \texttt{msg.sender} \mid \langle \text{Expr} \rangle\, \langle \text{Operator} \rangle\, \langle \text{Expr} \rangle \\
& \mid \texttt{address(} \langle \text{VName} \rangle \texttt{)} \mid \langle \text{Expr} \rangle\texttt{.}\langle \text{FName} \rangle \texttt{(} \langle \text{Expr} \rangle\texttt{, ...)} \\
& \mid \langle \text{FName} \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{Assign} \rangle \models & \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{Decl} \rangle \models & \langle \text{Types} \rangle\, \langle \text{VName} \rangle \\
\langle \text{Stmt} \rangle \models & \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{if(} \langle \text{Expr} \rangle \texttt{) \{} \langle \text{Stmt} \rangle \texttt{\}} \mid \texttt{while(} \langle \text{Expr} \rangle \texttt{) \{} \langle \text{Stmt} \rangle \texttt{\}} \mid \langle \text{Stmt} \rangle\texttt{;}\, \langle \text{Stmt} \rangle \\
\langle \text{Ctor} \rangle \models & \texttt{constructor (} \langle \text{Decl} \rangle\texttt{, ...) public \{} \langle \text{Stmt} \rangle \texttt{\}} \\
\langle \text{Func} \rangle \models & \texttt{function } \langle \text{FName} \rangle \texttt{(} \langle \text{Decl} \rangle\texttt{, ...) public \{} \langle \text{Stmt} \rangle \texttt{\}} \\
\langle \text{Contract} \rangle \models & \texttt{contract } \langle \text{CName} \rangle \texttt{\{} \langle \text{Decl} \rangle\texttt{; ...;} \langle \text{Ctor} \rangle\, \langle \text{Func} \rangle \texttt{ ... \}} \\
\langle \text{Bundle} \rangle \models & \langle \text{Contract} \rangle\, \langle \text{Contract} \rangle \ldots
\end{array}
$$

Fig. 3: The formal grammar of the MicroSol language.

include inheritance, cryptographic operations, or mappings between addresses. In our evaluation (Sec. 6), we use a superset of MicroSol, called MiniSol (see the extended version [42]), that extends our semantics to a wider set of smart contracts. Throughout this section, we illustrate MicroSol using `Auction` in Fig. 1.

A MicroSol *smart contract* is similar to a class in object-oriented programming, and consists of variables, and transactions (i.e., functions) for users to call. A transaction is a deterministic sequence of operations. Each smart contract user has a globally unique identifier, known as an *address*. We view a smart contract as operating in an SCUN: the control process executes each transaction sequentially, and the user processes are contract users that communicate with the control process. Users in the SCUN enter into a transaction through a synchronized action, then the control process executes the transaction as an internal action, and finally, the users are updated through synchronized actions. For simplicity of presentation, each transaction is given as a global transition.

A constructor is a special transaction that is executed once after contract creation. Calls to **`new`** (i.e., creating new smart contracts) are restricted to constructors. `Auction` in Fig. 1 is a smart contract that defines a constructor (line 6), three other functions (lines 8, 17, and 25), and four state variables (lines 2–3).

MicroSol has four types: *address*, *numeric* (including **`bool`**), *mapping*, and *contract reference*. Address variables prevent arithmetic operations, and numeric variables cannot cast to address variables. Mapping and contract-reference variables correspond to dictionaries and object pointers in other object-oriented languages. Each typed variable is further classified as either *state*, *input*, or *local*. We use *role* and *data* to refer to state variables of address and numeric types, respectively. Similarly, we use *client* and *argument* to refer to inputs of address and numeric types, respectively. In `Auction` of Fig. 1, there is 1 role (`manager`), 2 contract data (`leadingBid` and `stopped`), 1 mapping (`bids`), 1 client common to all transactions (`msg.sender`), and at most 1 argument in any transaction (`amount`).

Note that in MicroSol, *user* denotes any user process within a SCUN. A *client* is defined relative to a transaction, and denotes a user passed as an input.

Semantics of MicroSol. Let $\mathcal{C}$ be a MicroSol program with a single transaction tr (see the extended version [42] for multiple transactions). An N-user *bundle* is an N-user network of several (possibly identical) MicroSol programs. The semantics of a bundle is an LTS, $\mathsf{lts}(\mathcal{C}, N) := (S, P, f, s_0)$, where $S_C := \mathsf{control}(\mathcal{C}, [N])$ is the set of control states, $S_U := \mathsf{user}(\mathcal{C}, [N])$, is the set of user states, $s_\perp$ is the error state, $S \subseteq (S_C \cup \{s_\perp\}) \times (S_U)^N$ is the set of LTS states, $P := \mathsf{action}(\mathcal{C}, [N])$ is the set of actions, $f : S \times P \to S$ is the *transition function*, and s_0 is the initial state. We assume, without loss of generality, that there is a single control process[8].

Let $\mathbb{D}$ be the set of 256-bit unsigned integers. The state space of a smart contract is determined by the address space, $\mathcal{A}$, and the state variables of $\mathcal{C}$. In the case of $\mathsf{lts}(\mathcal{C}, N)$, the address space is fixed to $\mathcal{A} = [N]$. Assume that n, m, and k are the number of roles, data, and mappings in $\mathcal{C}$, respectively. State variables are stored by their numeric indices (i.e., variable $0, 1$, etc.). Then, $\mathsf{control}(\mathcal{C}, \mathcal{A}) \subseteq \mathcal{A}^n \times \mathbb{D}^m$ and $\mathsf{user}(\mathcal{C}, \mathcal{A}) \subseteq \mathcal{A} \times \mathbb{D}^k$. For $c = (\mathbf{x}, \mathbf{y}) \in \mathsf{control}(\mathcal{C}, \mathcal{A})$, $\mathsf{role}(c, i) = \mathbf{x}_i$ is the i-th role and $\mathsf{data}(c, i) = \mathbf{y}_i$ is the i-th datum. For $u = (z, \mathbf{y}) \in \mathsf{user}(\mathcal{C}, \mathcal{A})$, z is the address of u, and $\mathsf{map}(u) = \mathbf{y}$ are the mapping values of u.

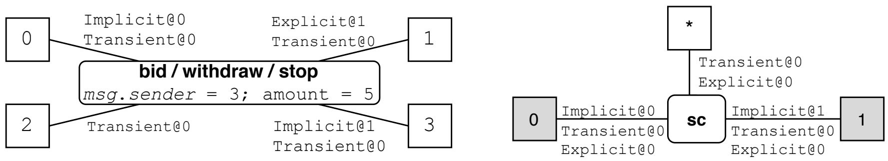
Similarly, actions are determined by the address space, $\mathcal{A}$, and the input variables of tr. Assume that q and r are the number of clients and arguments of tr, respectively. Then $\mathsf{action}(\mathcal{C}, \mathcal{A}) \subseteq \mathcal{A}^q \times \mathbb{D}^r$. For $p = (\mathbf{x}, \mathbf{y}) \in \mathsf{action}(\mathcal{C}, \mathcal{A})$, $\mathsf{client}(p, i) = \mathbf{x}_i$ 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 p, we write $f_p(s, \mathbf{u})$ to denote $f((s, \mathbf{u}), p)$.

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

An N-user transition function is determined by the (usual) semantics of tr, and a *bijection* from addresses to user indices, $\mathcal{M} : \mathcal{A} \to [N]$. If $\mathcal{M}(a) = i$, then address a belongs to user $\mathbf{u}_i$. In the case of $\mathsf{lts}(\mathcal{C}, N)$, the i-th user has address i, so $\mathcal{M}(i) = i$. We write $f := [\![\mathcal{C}]\!]_{\mathcal{M}}$, and given an action p, f_p updates the state variables according to the source code of tr with respect to $\mathcal{M}$. If an `assert` fails or an address is outside of $\mathcal{A}$, then the error state $s_\perp$ is returned. If a `require` fails, then the state is unchanged. Note that f preserves the address of each user.

For example, $\mathsf{lts}(\text{Auction}, 4) = (S, P, f, s_0)$ is the 4-user bundle of `Auction`. Assume that $(c, \mathbf{u})$ is the state reached after evaluating the constructor. Then $\mathsf{role}(c, 0) = 2$, $\mathsf{data}(c, 0) = 0$, $\mathsf{data}(c, 1) = 0$, and $\forall i \in [4] \cdot \mathsf{map}(\mathbf{u}_i)_0 = 0$. That is, the manager is at address 2, the leading bid is 0, the auction is not stopped, and there are no active bids. This is because variables are zero-indexed, and `stopped`

[8] Restrictions place on `new` ensure that the number of MicroSol smart contracts in a bundle is a static fact. Therefore, all control states are synchronized, and can be combined into a product machine.



(a) A PT for 4 users and a fixed action. (b) The PTG from `PTGBuilder`.

Fig. 4: A PT of `Auction` contrasted with a PTG for `Auction`.

is the second numeric variable (i.e., at index 1). If the user at address 3 placed a bid of 10, this corresponds to $p \in P$ such that $\mathsf{client}(p, 0) = 3$ and $\mathsf{arg}(p, 0) = 10$. A complete LTS for this example is in the extended version [42].

Limitations of MicroSol. MicroSol places two restrictions on Solidity. First, addresses are not numeric. We argue that this restriction is reasonable, as address manipulation is a form of pointer manipulation. Second, **`new`** must only appear in constructors. In our evaluation (Sec. 6), all calls to **`new`** could be moved into a constructor with minimal effort. We emphasize that the second restriction does not preclude the use of abstract interfaces for arbitrary contracts.

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 in interaction analysis to re-imagine smart contracts as having 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 the state of u affects execution of f or f affects a state of u. We call this *influence*. For example, in Fig. 1, the sender influences `withdraw` on line 19. Similarly, `withdraw` influences the sender on line 22. In all cases, the influence is *witnessed* by the state of the contract and the configuration of users that exhibit the influence.

Let $\mathcal{C}$ be a contract, $N \in \mathbb{N}$ be the network size, $(S, P, f, s_0) = \mathsf{lts}(\mathcal{C}, N)$, and $p \in P$. A user with address $a \in \mathbb{N}$ *influences* transaction f_p if there exists an $s, r, r' \in \mathsf{control}(\mathcal{C}, [N])$, $\mathbf{u}, \mathbf{u}', \mathbf{v}, \mathbf{v}' \in \mathsf{user}(\mathcal{C}, [N])^N$, and $i \in [N]$ such that:

1. $\mathsf{id}(\mathbf{u}_i) = a$;
2. $\forall j \in [N] \cdot (\mathbf{u}_j = \mathbf{v}_j) \iff (i \neq j)$;
3. $(r, \mathbf{u}') = f_p(s, \mathbf{u})$ and $(r', \mathbf{v}') = f_p(s, \mathbf{v})$;
4. $(r = r') \Rightarrow \big(\exists j \in [N] \setminus \{i\} \cdot \mathbf{u}'_j \neq \mathbf{v}'_j\big)$.

That is, there exists two network configurations that differ only in the state of the user $\mathbf{u}_i$, and result in different network configurations after applying f_p. In practice, f_p must compare the address of $\mathbf{u}_i$ to some other address, or must use the state of $\mathbf{u}_i$ to determine the outcome of the transaction. The tuple $(s, \mathbf{u}, \mathbf{v})$ is a *witness* to the influence of a over transaction f_p. A user with address $a \in \mathbb{N}$ is *influenced* by transaction f_p if there exists an $s, s' \in \mathsf{control}(\mathcal{C}, [N])$, $\mathbf{u}, \mathbf{u}' \in \mathsf{user}(\mathcal{C}, [N])^N$, and $i \in [N]$ such that:

1. $\mathsf{id}(\mathbf{u}_i) = a$;
2. $(s', \mathbf{u}') = f_p(s, \mathbf{u})$;
3. $\mathbf{u}'_i \neq \mathbf{u}_i$.

That is, f_p must write into the state of $\mathbf{u}_i$, and the changes must persist after the transaction terminates. The tuple $(s, \mathbf{u})$ is a *witness* to the influence of transaction f_p over user a.

Definition 1 (Participation). *A user with address $a \in \mathbb{N}$* participates *in a transaction f_p if either a influences f_p, witnessed by some $(s, \mathbf{u}, \mathbf{v})$, or f_p influences a, witnessed by some $(s, \mathbf{u})$. In either case, s is a* witness state*.*

Smart contracts facilitate communication between many users across many transactions. We need to know every possible participant, and the cause of their 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* if the participant is a client of the transaction; *transient* if the participant has a role during the transaction; *implicit* if there is a state such that the participant is neither a client nor holds any roles. In the case of MiniSol, all implicit participation is due to literal address values, as users designated by literal addresses must participate regardless of clients and roles. An example of implicit participation is when a client is compared to the address of the zero-account (i.e., `address(0)`) in Fig. 1.

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. *$Explicit \subseteq \mathbb{N} \times [N]$ where $(i, a) \in Explicit$ iff a participates during f_p, with $\mathsf{client}(p, i) = a$;*
2. *$Transient \subseteq \mathbb{N} \times [N]$ where $(i, a) \in Transient$ iff a participates during f_p, as witnessed by a state $s \in \mathsf{control}(\mathcal{C}, [N])$, where $\mathsf{role}(s, i) = a$;*
3. *$Implicit \subseteq [N]$ where $a \in Implicit$ iff a participates during f_p, as witnessed by a state $s \in \mathsf{control}(\mathcal{C}, [N])$, where $\forall i \in \mathbb{N}$, $\mathsf{role}(s, i) \neq a$ and $\mathsf{client}(p, i) \neq a$.*

For example, Fig. 4a shows a PT for any function of Fig. 1 with 4 users. From Sec. 1, it is clear that each function can have an affect. The zero-account and smart contract account are both implicit participants, since changing either account's address to 3 would block the affect of the transaction. The manager is a transient participant and the sender is an explicit participant, since the (dis)equality of their addresses is asserted at lines 9, 18, and 26.

Def. 2 is semantic and dependent on actions. A syntactic summary of all PTs for all actions is required to reason about communication. This summary is analogous to over-approximating control-flow with a "control-flow graph" [3]. This motivates the *Participation Topology Graph (PTG)* that is a syntactic over-approximation of all possible PTs, independent of network size. A PTG has a vertex for each user and each action, such that edges between vertices represent participation classes. In general, a single vertex can map to many users or actions.

PTG edges are labeled by participation classes. For any contract $\mathcal{C}$, there are at most m explicit classes and n transient classes, where n is the number of roles, and m is the maximum number of clients taken by any function of $\mathcal{C}$. On the other hand, the number of implicit classes is determined by the PTG itself. In general, there is no bound on the number of implicit participants, and it is up to a PTG to provide an appropriate abstraction (i.e., L in Def. 3). The label set common to all PTGs is $AP(\mathcal{C}) := \{\mathit{explicit@}i \mid i \in [n]\} \cup \{\mathit{transient@}i \mid i \in [m]\}$.

Definition 3 (Participation Topology Graph). *Let L be a finite set of implicit classes, $V \subsetneq \mathbb{N}$ be finite, $E \subseteq V \times V$, and $\delta \subseteq E \times (AP(\mathcal{C}) \cup L)$. A* PT Graph *for a contract $\mathcal{C}$ is a tuple $((V, E, \delta), \rho, \tau)$, where (V, E, δ) is a graph labeled by δ, $\rho \subseteq \mathsf{action}(\mathcal{C}, \mathbb{N}) \times V$, and $\tau \subseteq \mathsf{action}(\mathcal{C}, \mathbb{N}) \times \mathbb{N} \times V$, such that for all $N \in \mathbb{N}$ and for all $p \in \mathsf{action}(\mathcal{C}, [N])$, with $\mathsf{pt}(\mathcal{C}, N, p) = (\mathit{Explicit}, \mathit{Transient}, \mathit{Implicit})$:*

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

In Def. 3, τ and ρ map actions and users to vertices, respectively. An edge between an action and a user indicates the potential for participation. The labels describe the potential participation classes. As an example, Fig. 4b is a PTG for Fig. 1, where all actions map to *sc*, the zero-account maps to vertex 0, the smart contract account maps to vertex 1, and all other users map to $\star$. The two implicit classes have the label *implicit*@0 and *implicit*@1, respectively.

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

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

For any PT, there are many over-approximating PTGs. The weakest PTG joins every user to every action using all possible labels and a single implicit class. Fig. 4b, shows a simple, yet stronger, PTG for Fig. 1. First, note that there are two implicit participants, identified by addresses 0 and 1, with labels *implicit*@0 and *implicit*@1, respectively. Next, observe that any arbitrary user can become the manager. Finally, the distinctions between actions are ignored. Thus, there are three user vertices, two which are mapped to the zero-account and smart contract account, and another mapped to all other users. Such a PTG is constructed automatically using an algorithm named `PTGBuilder`.

`PTGBuilder` takes a contract $\mathcal{C}$ and returns a PTG. The implicit classes are $L := \{\mathit{implicit}@a \mid a \in \mathbb{N}\}$, where $\mathit{implicit}@a$ signifies implicit communication with address a. PTG construction is reduced to taint analysis [23]. Input address variables, state address variables, and literal addresses are tainted sources. Sinks are memory writes, comparison expressions, and mapping accesses. `PTGBuilder` computes $(\mathit{Args}, \mathit{Roles}, \mathit{Lits})$, 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. Finally, a PTG is constructed as (G, ρ, τ), where $G = (V, E, \delta)$, $\rho \subseteq \mathsf{action}(\mathcal{C}, \mathbb{N}) \times V$, $\tau \subseteq \mathsf{action}(\mathcal{C}, \mathbb{N}) \times \mathbb{N} \times V$, sc, and $\star$ are unique vertices:

1. $V := \mathit{Lits} \cup \{sc, \star\}$ and $E := \{(sc, v) \mid v \in V \backslash \{sc\}\}$;
2. $\delta := \{(e, \mathit{explicit}@i) \mid e \in E, i \in \mathit{Args}\} \cup \{(e, \mathit{transient}@i) \mid e \in E, i \in \mathit{Roles}\} \cup \{((sc, a), \mathit{transient}@a) \mid a \in \mathit{Lits}\}$;
3. $\rho := \{(p, sc) \mid p \in \mathsf{action}(\mathcal{C}, \mathbb{N})\}$;
4. $\tau := \{(p, a, \star) \mid p \in \mathsf{action}(\mathcal{C}, \mathbb{N}), a \in \mathbb{N} \backslash \mathit{Lits}\} \cup \{(p, a, a) \mid p \in \mathsf{action}(\mathcal{C}, \mathbb{N}), a \in \mathit{Lits}\}$.

`PTGBuilder` formalizes the intuition of Fig. 4b. Rule 1 ensures that every literal address has a vertex, and that all user vertices connect to sc. Rule 2 over-approximates explicit, transient, and implicit labels. The first set states that if an input address is never used, then the client is not an explicit participant. This statement is self-evident, and over-approximates explicit participation. The second and third set make similar claims for roles and literal addresses Rules 3 and 4 define ρ and τ as expected. Note that in MicroSol, implicit participation stems from literal addresses, since addresses do not support arithmetic operations, and since numeric expressions cannot be cast to addresses.

By re-framing smart contracts with rendezvous synchronization, each transaction is re-imagined as a communication between 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 PCMC as it provides an upper bound on the number of equivalence classes, and the users in each equivalence class (see the extended version [42]).

5 Local Reasoning in Smart Contracts

In this section, we present a proof rule for the parameterized safety of MicroSol programs. Our proof rule extends the existing theory of PCMC. The section is structured as follows. Sec. 5.1 introduces syntactic restrictions, for properties and interference invariants, that expose address dependencies. Sec. 5.2, defines local bundle reductions, that reduce parameterized smart contract models to finite-state models. We show that for the correct choice of local bundle reduction, the safety of the finite-state model implies the safety of the parameterized model.

5.1 Guarded Properties and Split Invariants

Universal properties and interference invariants might depend on user addresses. However, PCMC requires explicit address dependencies. This is because address dependencies allow predicates to distinguish subsets of users. To resolve this, we introduce two syntactic forms that make address dependencies explicit: guarded universal safety properties and split interference invariants. We build both forms from so called *address-oblivious* predicates that do not depend on user addresses.

For any smart contract $\mathcal{C}$ and any address space $\mathcal{A}$, a pair of user configurations, $\mathbf{u}, \mathbf{v} \in \mathsf{user}(\mathcal{C}, \mathcal{A})^k$, are *k-address similar* if $\forall i \in [k] \cdot \mathsf{map}(\mathbf{u}_i) = \mathsf{map}(\mathbf{v}_i)$. A predicate $\xi \subseteq \mathsf{control}(\mathcal{C}, \mathcal{A}) \times \mathsf{user}(\mathcal{C}, \mathcal{A})^k$ is address-oblivious if, for every choice of $s \in \mathsf{control}(\mathcal{C}, \mathcal{A})$, and every pair of k-address similar configurations, $\mathbf{u}$ and $\mathbf{v}$, $\xi(s, \mathbf{u}) \iff \xi(s, \mathbf{v})$. **Prop. 1** and **Prop. 2** in Sec. 1 are 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 its k user addresses. Each constraint compares a single user's address to either a literal address or a role. This notion is formalized by Def. 4, and illustrated in Ex. 1.

Definition 4 (Guarded Universal Safety). *For $k \in \mathbb{N}$, a* guarded k-universal safety property *is a k-universal safety property φ, given by a tuple (L, R, ξ), where $L \subsetneq \mathbb{N} \times [k]$ is finite, $R \subsetneq \mathbb{N} \times [k]$ is finite, and ξ is an address-oblivious k-user predicate, such that:*

$$\varphi(s, \mathbf{u}) := \left(\left(\bigwedge_{(a,i) \in L} a = \mathit{id}(\mathbf{u}_i) \right) \wedge \left(\bigwedge_{(i,j) \in R} \mathit{role}(s, i) = \mathit{id}(\mathbf{u}_j) \right) \right) \Rightarrow \xi(s, \mathbf{u})$$

Note that $\mathcal{A}_L := \{a \mid (a, i) \in L\}$ and $\mathcal{A}_R := \{i \mid (i, j) \in R\}$ and define the literal *and* role guards *for φ.*

Example 1. Consider the claim that in `Auction` of Fig. 1, the zero-account cannot have an active bid. This claim is stated as **Prop. 3**: *For each user process $\mathbf{u}$, if $\mathit{id}(\mathbf{u}_0) = 0$, then $\mathit{map}(\mathbf{u}_0)_0 = 0$.* That is, **Prop. 3** is a guarded 1-universal safety property $\varphi_1(s, \mathbf{u}) := (0 = \mathsf{id}(\mathbf{u}_0)) \Rightarrow (\mathsf{map}(\mathbf{u}_0)_0 = 0)$. Following Def. 4, φ_1 is determined by $(L_1, \varnothing, \xi_1)$, where $L_1 = \{(0, 0)\}$ and $\xi_1(s, \mathbf{u}) := \mathsf{map}(\mathbf{u}_0)_0 = 0$. The second set is $\varnothing$ as there are no role constraints in **Prop. 3**. If a state $(s, \mathbf{u})$ satisfies φ_1, then $\forall \{i\} \subseteq [N] \cdot \varphi_1(s, (\mathbf{u}_i))$. 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 a 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 other constraints. Intuitively, each address-oblivious predicate summarizes the class of users that satisfy its guard. The split interference invariant is the conjunction of all (guarded predicate) clauses. We proceed with the formal definition in Def. 5 and a practical illustration in Ex. 2.

Definition 5 (Split Interference Invariant). *A* split interference invariant *is an interference invariant θ, given by a tuple $(\mathcal{A}_L, \mathcal{A}_R, \zeta, \mu, \xi)$, where $\mathcal{A}_L = \{l_0, \ldots, l_{m-1}\} \subsetneq \mathbb{N}$ is finite, $\mathcal{A}_R = \{r_0, \ldots, r_{n-1}\} \subsetneq \mathbb{N}$ is finite, 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 an address-oblivious 1-user predicate, such that:

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

Note that $\mathcal{A}_L$ and $\mathcal{A}_R$ define literal *and* role guards *of θ, and that $|\mathbf{u}| = 1$.*

Example 2. To establish φ_1 from Ex. 1, we require an adequate interference invariant such as **Prop. 4**: *The zero-account never has an active bid, while all other users can have active bids.* That is, **Prop. 4** is a split interference invariant:

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

Following Def. 5, θ_1 is determined by $\mathit{Inv} = (\mathcal{A}_L, \varnothing, (\xi_1), \varnothing, \xi_2)$, where $\mathcal{A}_L = \{0\}$, ξ_1 is defined in Ex. 1, and $\xi_2(s, \mathbf{u}) := \mathsf{map}(\mathbf{u}_0)_0 \geq 0$. The two instances of $\varnothing$ in *Inv* correspond to the lack of role constraints in θ_1. If *Inv* is related back to Def. 5, then $\psi_{\text{Roles}}(s, \mathbf{u}) := \top$, $\psi_{\text{Lits}}(s, \mathbf{u}) := (\mathsf{id}(\mathbf{u}_0) = 0) \Rightarrow (\mathsf{map}(\mathbf{u}_0)_0 = 0)$, and $\psi_{\text{Else}}(s, \mathbf{u}) := (\mathsf{id}(\mathbf{u}_0) \neq 0) \Rightarrow (\mathsf{map}(\mathbf{u}_0)_0 \geq 0)$. □

5.2 Localizing a Smart Contract Bundle

A local bundle is a finite-state abstraction of a smart contract bundle. This abstraction reduces smart contract PCMC to software model checking. 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. The term *candidate interference invariant* describes any predicate with the syntax of an interference invariant, regardless of its semantic interpretation. Sets of addresses are used to identify representatives in a neighbourhood.

Let $\mathcal{A}$ be an N-user neighbourhood and θ_U be a candidate interference invariant. The local bundle corresponding to $\mathcal{A}$ and θ_U is defined using a special relation called an *N-user interference relation*. The N-user interference relation (for θ_U) sends an N-user smart contract state to the set of all N-user smart contract states that are reachable under the interference of θ_U. A state is reachable under the interference of θ_U if the control state is unchanged, each address is unchanged, and all user data satisfies θ_U. For example, lines 6–10 in Fig. 2 apply a 4-user interference relation to the states of `Auction`. Note that if the interference relation for θ_U fails to relate $(s, \mathbf{u})$ to itself, then $(s, \mathbf{u})$ violates θ_U.

Definition 6 (Interference Relation). *Let $N \in \mathbb{N}$, $\mathcal{C}$ be a contract, $S = \mathit{control}(\mathcal{C}, \mathbb{N}) \times \mathit{user}(\mathcal{C}, \mathbb{N})^N$, and θ_U be a split candidate interference invariant. The N-user interference relation for θ_U is the relation $g : S \to 2^S$ such that $g(c, \mathbf{u}) := \{(c, \mathbf{v}) \in S \mid \forall i \in [N] \cdot \mathit{id}(\mathbf{u}_i) = \mathit{id}(\mathbf{v}_i) \wedge \theta_U(s, \mathbf{v}_i)\}$.*

Each state of the *local bundle* for $\mathcal{A}$ and θ_U is a tuple $(s, \mathbf{u})$, where s is a control state and $\mathbf{u}$ is an N-user configuration. The N users in the local bundle correspond to the N representatives in $\mathcal{A}$, and therefore, the address space of the local bundle can be non-consecutive. The transition relation of the local bundle is defined in terms of the (global) transaction function f. First, the transition relation applies f. If the application of f is closed under θ_U, then the interference relation is applied. Intuitively, θ_U defines a safe envelop under which the interference relation is compositional.

Definition 7 (Local Bundle). *Let $\mathcal{C}$ be a contract, $\mathcal{A} = \{a_0, \ldots, a_{N-1}\} \subseteq \mathbb{N}$ be an N-user neighbourhood, θ_U be a candidate split interference invariant, and g be the N-user interference relation for θ_U. A* local bundle *is an LTS $\mathit{local}(\mathcal{C}, \mathcal{A}, \theta_U) := (S, P, \hat{f}, s_0)$, such that $S := \mathit{control}(\mathcal{C}, \mathcal{A}) \times \mathit{user}(\mathcal{C}, \mathcal{A})^N$, $P := \mathit{action}(\mathcal{C}, \mathcal{A})$, $s_0 := (c_0, \mathbf{u})$, $c_0 := (\mathbf{0}, \mathbf{0})$, $\forall i \in [N] \cdot \mathit{id}(\mathbf{u}_i) = a_i \wedge \mathit{map}(\mathbf{u}_i) = \mathbf{0}$, and $\hat{f}$ is defined with respect to $\mathcal{M} : \mathcal{A} \to [N]$, $\mathcal{M}(a_i) = i$, such that:*

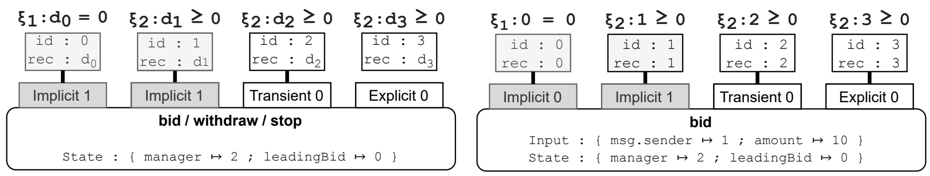
$$\hat{f}((s, \mathbf{u}), p) := \begin{cases} g(s', \mathbf{u}') & \text{if } (s', \mathbf{u}') = [\![\mathcal{C}]\!]_{\mathcal{M}}((s, \mathbf{u}), p) \wedge (s', \mathbf{u}') \in g(s', \mathbf{u}') \\ [\![\mathcal{C}]\!]_{\mathcal{M}}((s, \mathbf{u}), p) & \text{otherwise} \end{cases}$$

Example 3. We briefly illustrate the transition relation of Def. 7 using `Auction` of Fig. 1. Let $\mathcal{A}_1 = \{0, 1, 2, 3\}$ be a neighbourhood, θ_1 be as in Ex. 2, g be the 4-user interference relation for θ_1, and $(S, P, \hat{f}, s_0) = \mathsf{local}(\mathcal{C}, \mathcal{A}_1, \theta_1)$. Consider applying $\hat{f}$ to $(s, \mathbf{u}) \in S$ with action $p \in P$, such that $s = \{\texttt{manager} \mapsto 2; \texttt{leadingBid} \mapsto 0\}$, $\forall i \in [4] \cdot \mathsf{map}(\mathbf{u}_i) = 0$, and p is a bid of 10 from a sender at address 3.

By definition, if $(s', \mathbf{v}) = f(s, \mathbf{u}, p)$, then the leading bid is now 10, and the bid of the sender is also 10, since the sender of p was not the manager and the leading bid was less than 10. Clearly $(s', \mathbf{v}) \in g(s', \mathbf{v})$, and therefore, $g(s', \mathbf{v}) = \hat{f}((s, \mathbf{u}), p)$. A successor state is then selected, as depicted in Fig. 5a. This is done by first assigning an arbitrary bid to each representative, and then requiring that each bid satisfies θ_1 relative to s'. In Fig. 5a, a network is selected in which $\forall i \in [4] \cdot \mathsf{id}(\mathbf{v}_i) = i$. As depicted in Fig. 5a, θ_1 stipulates that the zero-account must satisfy ξ_1 and that all other users must satisfy ξ_2.

In Fig. 5b, a satisfying bid is assigned to each user. The choice for d_0 was fixed since $\xi_1(s, \mathbf{v}_0)$ entails $d_0 = 0$. For d_1 to d_3, any non-negative value could have been selected. After the transaction is executed, $\mathsf{map}(\mathbf{u}'_0)_0 = 0$, $\mathsf{map}(\mathbf{u}'_1)_0 = 1$, $\mathsf{map}(\mathbf{u}'_2)_0 = 2$, $\mathsf{map}(\mathbf{u}'_3)_0 = 3$, and $s' = \{\texttt{manager} \mapsto 2; \texttt{leadingBid} \mapsto 10\}$. Then $(s', \mathbf{u}') \in \hat{f}(s, \mathbf{u})$, as desired. Note that $(s', \mathbf{u}')$ is not reachable in $\mathsf{lts}(\mathcal{C}, 4)$. □

Ex. 3 motivates an important result for local bundles. Observe that $(s', \mathbf{u}') \models \theta_1$. This is not by chance. First, by the compositionality of θ_1, all user configurations reached by $\mathsf{local}(\mathcal{C}, \mathcal{A}_1, \theta_1)$ must satisfy θ_U. Second, and far less obviously,



(a) A local 4-user configuration. (b) The saturating property of $\mathcal{A}_1$.

Fig. 5: The local bundle for `Auction` in Fig. 1, as defined by $\mathcal{A}_1$ and θ_1 in Ex. 3.

by choice of $\mathcal{A}_1$, if all reachable user configurations satisfy θ_1, then θ_1 must be compositional. The proof of this result relies on a saturating property of $\mathcal{A}_1$.

A neighbourhood $\mathcal{A}$ is *saturating* if it contains representatives from each participation class of a PTG, and for all role guards ($\mathcal{A}_R \subsetneq \mathbb{N}$) and literal guards ($\mathcal{A}_L \subseteq \mathbb{N}$) of interest. Intuitively, each participation class over-approximates an equivalence class of $\mathcal{C}$. The number of representatives is determined by the equivalence class. In the case of `PTGBuilder`, a saturating neighbourhood contains one address for each participation class. For an implicit class, such as *implicit*@x, x is literal and must appear in the neighbourhood. All other addresses are selected arbitrarily. The saturating property of $\mathcal{A}_1$ is depicted in Fig. 5b by the correspondence between users and participation classes ($\mathcal{A}_R = \varnothing$, $\mathcal{A}_L = \{0\}$).

Definition 8 (Saturating Neighbourhood). *Let* $\mathcal{A}_R, \mathcal{A}_L \subseteq \mathbb{N}$*,* $\mathcal{C}$ *be a contract,* (G, ρ, τ) *be the* `PTGBuilder` *PTG of* $\mathcal{C}$*, and* $G = (V, E, \delta)$ *such that* $\mathcal{A}_R$ *and* $\mathcal{A}_L$ *are finite. A* saturating neighbourhood *for* $(\mathcal{A}_R, \mathcal{A}_L, (G, \rho, \tau))$ *is a set* $\mathcal{A}_{\text{Exp}} \cup \mathcal{A}_{\text{Trans}} \cup \mathcal{A}_{\text{Impl}}$ *s.t.* $\mathcal{A}_{\text{Exp}}, \mathcal{A}_{\text{Trans}}, \mathcal{A}_{\text{Impl}} \subseteq \mathbb{N}$ *are pairwise disjoint and:*

1. $|\mathcal{A}_{\text{Exp}}| = |\{i \in \mathbb{N} \mid \exists e \in E \cdot \delta\,(e, \textit{explicit}@i)\}|$,
2. $|\mathcal{A}_{\text{Trans}}| = |\{i \in \mathbb{N} \mid \exists e \in E \cdot \delta\,(e, \textit{transient}@i)\} \cup \mathcal{A}_R|$,
3. $\mathcal{A}_{\text{Impl}} = \{x \in \mathbb{N} \mid \exists e \in E \cdot \delta\,(e, \textit{implicit}@x)\} \cup \mathcal{A}_L$.

A saturating neighbourhood can be used to reduce compositionality and k-safety proofs to the safety of local bundles. We start with compositionality. Consider a local bundle with a neighbourhood $\mathcal{A}^+$, where $\mathcal{A}^+$ contains a saturating neighbourhood, the guards of θ_U, and some other address a. The neighbourhood $\mathcal{A}^+$ contains a representative for: each participation class; each role and literal user distinguished by θ_U; an arbitrary user under interference (i.e., a). We first claim that if θ_U is compositional, then a local bundle constructed from θ_U must be safe with respect to θ_U (as in Ex. 3). The first claim follows by induction. By **Initialization** (Sec. 2), the initial users satisfy θ_U. For the inductive step, assume that all users satisfy θ_U and apply $\hat{f}_p$. The users that participate in $\hat{f}_p$ maintain θ_U by **Consecution** (Sec. 2). The users that do not participate also maintain θ_U by **Non-Interference** (Sec. 2). By induction, the first claim is true. We also claim that for a sufficiently large neighbourhood—say $\mathcal{A}^+$—the converse is also true. Intuitively, $\mathcal{A}^+$ is large enough to represent each equivalence class imposed by both the smart contract and θ_U, along with an arbitrary user under interference. Our key insight is that the reachable control states of the local bundle form an inductive invariant θ_C. If the local bundle is safe, then the interference relation is applied after each transition, and, therefore, the local bundle considers every pair of control and user states (c, u) such that $c \in \theta_C$ and $(c, u) \in \theta_U$. Therefore, the safety of the local bundle implies **Initialization**, **Consecution**, and **Non-Interference**. This discussion justifies Theorem 2.

Theorem 2. *Let $\mathcal{C}$ be a contract, G be a PTG for $\mathcal{C}$, θ_U be a candidate split interference invariant with role guards $\mathcal{A}_R$ and literal guards $\mathcal{A}_L$, $\mathcal{A}$ be a saturating neighbourhood for $(\mathcal{A}_R, \mathcal{A}_L, G)$, $a \in \mathbb{N} \backslash \mathcal{A}$, and $\mathcal{A}^+ = \{a\} \cup \mathcal{A}$. Then, $\mathit{local}(\mathcal{C}, \mathcal{A}^+, \theta_U) \models \theta_U$ if and only if θ_U is an interference invariant for $\mathcal{C}$.*

Next, we present our main result: a sound proof rule for k-universal safety. As in Theorem 2, Theorem 3 uses a saturating neighbourhood $\mathcal{A}^+$. This proof rule proves inductiveness, rather than compositionality, so $\mathcal{A}^+$ does not require an arbitrary user under interference. However, a k-universal property can distinguish between k users at once. Thus, $\mathcal{A}^+$ must have at least k arbitrary representatives.

Theorem 3. *Let φ be a k-universal safety property with role guards $\mathcal{A}_R$ and literal guards $\mathcal{A}_L$, $\mathcal{C}$ be a contract, θ_U be an interference invariant for $\mathcal{C}$, G be a PTG for $\mathcal{C}$, $\mathcal{A} = \mathcal{A}_{\mathrm{Exp}} \cup \mathcal{A}_{\mathrm{Trans}} \cup \mathcal{A}_{\mathrm{Impl}}$ be a saturating neighbourhood for $(\mathcal{A}_R, \mathcal{A}_L, G)$. Define $\mathcal{A}^+ \subseteq \mathbb{N}$ such that $\mathcal{A} \subseteq \mathcal{A}^+$ and $|\mathcal{A}^+| = |\mathcal{A}| + \max(0, k - |\mathcal{A}_{\mathrm{Exp}}|)$. If $\mathit{local}(\mathcal{C}, \mathcal{A}^+, \theta_U) \models \varphi$, then $\forall N \in \mathbb{N} \cdot \mathit{lts}(\mathcal{C}, N) \models \varphi$.*

Theorem 3 completes Ex. 2. Recall $(\varphi_1, \theta_1, \mathcal{A}_1)$ from Ex. 3. Since φ_1 is 1-universal and $\mathcal{A}_1$ has one explicit representative, it follows that $\mathcal{A}^+ = \mathcal{A}_1 \cup \varnothing$. Using an SMC, $\mathsf{local}(\mathcal{C}, \mathcal{A}_1^+, \theta_1) \models \varphi_1$ is certified by an inductive strengthening θ_1^*. Then by Theorem 3, $\mathcal{C}$ is also safe for 2^{160} users. Both the local and global bundle have states exponential in the number of users. However, the local bundle has 4 users (a constant fixed by $\mathcal{C}$), whereas the global bundle is defined for any number of users. This achieves an exponential state reduction with respect to the network size. 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. Therefore, we have achieved an exponential speedup in verification and have automated the discovery of an inductive invariant.

6 Implementation and Evaluation

We implement smart contract PCMC as an open-source tool called SmartACE, that is built upon the Solidity compiler. It works in the following automated steps: (1) consume a Solidity smart contract and its interference invariants; (2) validate the contract's conformance to MiniSol; (3) perform source-code analysis and transformation (i.e., inheritance inlining, devirtualization, `PTGBuilder`); (4) generate a local bundle in LLVM IR; (5) verify the bundle using SeaHorn [15]. In this section, we report on the effectiveness of SmartACE in verifying real-world smart contracts. A full description of the SmartACE architecture and of

Contracts			SmartACE			VerX
Name	Prop.	LOC	Time	Inv. Size	Users	Time
Alchemist	3	401	7	0	7	29
ERC20	9	599	12	1	5	158
Melon	16	462	30	0	7	408
MRV	5	868	2	0	7	887
Overview	4	66	4	0	8	211
PolicyPal	4	815	26	0	8	20,773
Zebi	5	1,209	8	0	7	77
Zilliqa	5	377	8	0	7	94
Brickblock	6	549	13	0	10	191
Crowdsale	9	1,198	223	0	8	261
ICO	8	650	371	0	16	6,817
VUToken	5	1,120	19	0	10	715
Mana	4	885	—	—	—	41,409
Fund	2	38	1	0	6	—
Auction	1	42	1	1	5	—
QSPStaking	4	1,550	3	7	8	—

Table 1: Experimental results for SmartACE. All reported times are in seconds.

each case study is beyond the scope of this paper. Both SmartACE and the case studies are available[9]. Our evaluation answers the following research questions:

RQ1: Compliance. Can 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 89 properties across 15 smart contracts (see Tab. 1). Contracts `Alchemist` to `Mana` are from VerX [34]. Contracts `Fund` and `Auction` were added to offset the lack of parameterized properties in existing benchmarks. The `QSPStaking` contract comprises the Quantstamp Assurance Protocol[10] for which we checked real-world properties provided by Quantstamp. Some properties require additional instrumentation techniques (i.e., temporal [34] and aggregate [17] properties). Aggregate properties allow SmartACE to reason about the sum of all records within a mapping. In Tab. 1, *Inv. Size* is the clause size of an interference invariant manually provided to SmartACE and *Users* is the maximum number of users requested by `PTGBuilder`. All experiments were run on an Intel® Core i7® CPU @ 2.8GHz 4-core machine with 16GB of RAM on Ubuntu 18.04.

RQ1: Compliance. To assess if the restrictions of MiniSol are reasonable, we find the number of *compliant* VerX benchmarks. We found that 8 out of 13 bench-

[9] https://github.com/contract-ace
[10] https://github.com/quantstamp/qsp-staking-protocol

marks are compliant after removing dead code. With manual abstraction, 4 more benchmarks complied. `Brickblock` uses inline assembly to revert transactions with smart contract senders. We remove the assembly as an over-approximation. To support `Crowdsale`, we manually resolve dynamic calls not supported by SmartACE. In `ICO`, calls are made to arbitrary contracts (by address). However, these calls adhere to *effectively external callback freedom* [12,34] and can be omitted. Also, `ICO` uses dynamic allocation, but the allocation is performed once. We inline the first allocation, and assert that all other allocations are unreachable. To support `VUToken`, we replace a dynamic array of bounded size with variables corresponding to each element of the array. The function `_calcTokenAmount` iterates over the array, so we specialize each call (i.e., `_calcTokenAmount_{1,2,3,4}`) to eliminate recursion. Two other functions displayed unbounded behaviour (i.e., `massTransfer` and `addManyToWhitelist`), but are used to sequence calls to other functions, and do not impact reachability. We conclude that the restrictions of MiniSol are reasonable.

RQ2: Effectiveness. To assess the effectiveness of SmartACE, we determined the number of properties verified from compliant VerX contracts. We found that all properties could be verified, but also discovered that most properties were not parameterized. To validate SmartACE with parameterized properties, we conducted a second study using `Auction`, as described on our development blog[11]. To validate SmartACE in the context of large-scale contract development, we performed a third study using `QSPStaking`. In this study, 4 properties were selected at random, from a specification provided by Quantstamp, and validated. It required 2 person days to model the environment, and 1 person day to discover an interference invariant. The major overhead in modeling the environment came from manual abstraction of unbounded arrays. The discovery of an interference invariant and array abstractions were semi-automatic, and aided by counterexamples from SeaHorn. For example, one invariant used in our abstraction says that all elements in the array `powersOf100` must be non-zero. This invariant was derived from a counterexample in which 0 was read spuriously from `powersOf100`, resulting in a division-by-zero error. We conclude that SmartACE is suitable for high-assurance contracts, and with proper automation, can be integrated into contract development.

RQ3: Performance. To evaluate the performance of SmartACE, we compared its verification time to the reported time of VerX, a state-of-the-art, semi-automated verification tool. Note that in VerX, predicate abstractions must be provided manually, whereas SmartACE automates this step. VerX was evaluated on a faster processor (3.4GHz) with more RAM (64GB)[12]. In each case, SmartACE significantly outperformed VerX, achieving a speedup of at least 10x for all but 2 contracts[13]. One advantage of SmartACE is that it benefits

[11] `http://seahorn.github.io/blog/`
[12] We have requested access to VerX and are awaiting a response.
[13] We compare the average time for VerX to the total evaluation time for SmartACE.

from state-of-the art software model checkers, whereas the design of VerX requires implementing a new verification tool. In addition, we suspect that local bundle abstractions obtained through smart contract PCMC are easier to reason about than the global arrays that VerX must quantify over. However, a complete explanation for the performance improvements of SmartACE is challenging without access to the source code of VerX. We observe that one bottleneck for SmartACE is the number of users (which extends the state space). A more precise `PTGBuilder` would reduce the number of users. Upon manual inspection of `Melon` and `Alchemist` (in a single bundle), we found that user state could be reduced by 28%. We conclude that SmartACE can scale.

7 Related Work

In recent years, the program analysis community has developed many tools for smart contract analysis. These tool range from dynamic analysis [19,43] to static analysis [27,30,26,39,13,32,25,40,5] and verification [21,17,41,29,34,38]. The latter are most related to SmartACE since their focus is on functional correctness, as opposed to generic rules (e.g., the absence of reentrancy [14] and integer overflows). Existing techniques for functional correctness are either deductive, and require that most invariants be provided manually (i.e., [17,41]), or are automated but neglect the parameterized nature of smart contracts (i.e., [28,29,34,38]). The tools that do acknowledge parameterization employ static analysis [25,5]. In contrast, SmartACE uses a novel local reasoning technique that verifies parameterized safety properties with less human guidance 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., [9,36,37]). 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 [35,10] generalize from small instances through the use of ranking functions. The combination of abstract domains and SMPs has also proven useful in finding parameterized invariants [2]. The addresses used in our analysis are similar to the scalarsets of [18]. Most compositional techniques require cutoff analysis—considering network instances up to a given size [7,20,22]. Local bundles avoid explicit cutoff analysis by simulating all smaller instances, and is similar to existing work on bounded parameterized model checking [8]. SmartACE is the first application of PCMC in the context of smart contracts.

8 Conclusions

In this paper, we present a new verification approach for Solidity smart contracts. Unlike many of the existing approaches, we automatically reason about smart contracts relative to all of their clients and accross multiple transaction. Our approach is based on treating smart contracts as a parameterized system and using Parameterized Compositional Model Checking (PCMC).

Our main theoretical contribution is to show that PCMC offers an exponential reduction for k-universal safety verification of smart contracts. That is, verification of safety properties with k arbitrary clients.

The theoretical results of this paper are implemented in an automated Solidity verification tool SmartACE. SmartACE is built upon a novel model for smart contracts, in which users are processes and communication is explicit. In this model, communication is over-approximated by static analysis, and the results are sufficient to find all local neighbourhoods, as required by PCMC. The underlying parameterized verification task is reduced to sequential Software Model Checking. In SmartACE, we use the SeaHorn verification framework for the underlying analysis. However, other Software Model Checkers can potentially be used as well.

Our approach is almost completely automated – SmartACE automatically infers the necessary predicates, inductive invariants, and transaction summaries. The only requirement from the user is to provide an occasional interference invariant (that is validated by SmartACE). However, we believe that this step can be automated as well through reduction to satisfiability of Constrained Horn Clauses. We leave exploring this to future work.

References

1. Abdulla, P.A., Haziza, F., Holík, L.: All for the price of few. In: Giacobazzi, R., Berdine, J., Mastroeni, I. (eds.) Verification, Model Checking, and Abstract Interpretation, 14th International Conference, VMCAI 2013, Rome, Italy, January 20-22, 2013. Proceedings. Lecture Notes in Computer Science, vol. 7737, pp. 476–495. Springer (2013). https://doi.org/10.1007/978-3-642-35873-9_28
2. Abdulla, P.A., Haziza, F., Holík, L.: Parameterized verification through view abstraction. Int. J. Softw. Tools Technol. Transf. **18**(5), 495–516 (2016). https://doi.org/10.1007/s10009-015-0406-x
3. Allen, F.E.: Control flow analysis. In: Proceedings of a Symposium on Compiler Optimization. pp. 1–19. Association for Computing Machinery, New York, NY, USA (1970). https://doi.org/10.1145/800028.808479, https://doi.org/10.1145/800028.808479
4. Bloem, R., Jacobs, S., Khalimov, A., Konnov, I., Rubin, S., Veith, H., Widder, J.: Decidability in parameterized verification. SIGACT News **47**(2), 53–64 (2016). https://doi.org/10.1145/2951860.2951873
5. Brent, L., Grech, N., Lagouvardos, S., Scholz, B., Smaragdakis, Y.: Ethainter: a smart contract security analyzer for composite vulnerabilities. In: Donaldson, A.F., Torlak, E. (eds.) Proceedings of the 41st ACM SIGPLAN International Conference on Programming Language Design and Implementation, PLDI 2020, London, UK, June 15-20, 2020. pp. 454–469. ACM (2020). https://doi.org/10.1145/3385412.3385990
6. Clarke, E.M., Talupur, M., Veith, H.: Environment abstraction for parameterized verification. In: Emerson, E.A., Namjoshi, K.S. (eds.) Verification, Model Checking, and Abstract Interpretation, 7th International Conference,

VMCAI 2006, Charleston, SC, USA, January 8-10, 2006, Proceedings. Lecture Notes in Computer Science, vol. 3855, pp. 126–141. Springer (2006). https://doi.org/10.1007/11609773_9
7. Emerson, E.A., Namjoshi, K.S.: On reasoning about rings. Int. J. Found. Comput. Sci. **14**(4), 527–550 (2003). https://doi.org/10.1142/S0129054103001881
8. Emerson, E.A., Trefler, R.J., Wahl, T.: Reducing model checking of the few to the one. In: Liu, Z., He, J. (eds.) Formal Methods and Software Engineering, 8th International Conference on Formal Engineering Methods, ICFEM 2006, Macao, China, November 1-3, 2006, Proceedings. Lecture Notes in Computer Science, vol. 4260, pp. 94–113. Springer (2006). https://doi.org/10.1007/11901433_6
9. Esparza, J., Ganty, P., Majumdar, R.: Parameterized verification of asynchronous shared-memory systems. In: Sharygina, N., Veith, H. (eds.) Computer Aided Verification - 25th International Conference, CAV 2013, Saint Petersburg, Russia, July 13-19, 2013. Proceedings. Lecture Notes in Computer Science, vol. 8044, pp. 124–140. Springer (2013). https://doi.org/10.1007/978-3-642-39799-8_8
10. Fang, Y., Piterman, N., Pnueli, A., Zuck, L.D.: Liveness with invisible ranking. In: Steffen, B., Levi, G. (eds.) Verification, Model Checking, and Abstract Interpretation, 5th International Conference, VMCAI 2004, Venice, Italy, January 11-13, 2004, Proceedings. Lecture Notes in Computer Science, vol. 2937, pp. 223–238. Springer (2004). https://doi.org/10.1007/978-3-540-24622-0_19
11. German, S.M., Sistla, A.P.: Reasoning about systems with many processes. J. ACM **39**(3), 675–735 (1992). https://doi.org/10.1145/146637.146681
12. Gershuni, E., Amit, N., Gurfinkel, A., Narodytska, N., Navas, J.A., Rinetzky, N., Ryzhyk, L., Sagiv, M.: Simple and precise static analysis of untrusted linux kernel extensions. In: McKinley, K.S., Fisher, K. (eds.) Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2019, Phoenix, AZ, USA, June 22-26, 2019. pp. 1069–1084. ACM (2019). https://doi.org/10.1145/3314221.3314590
13. Grech, N., Kong, M., Jurisevic, A., Brent, L., Scholz, B., Smaragdakis, Y.: Madmax: surviving out-of-gas conditions in ethereum smart contracts. Proc. ACM Program. Lang. **2**(OOPSLA), 116:1–116:27 (2018). https://doi.org/10.1145/3276486
14. 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. Proc. ACM Program. Lang. **2**(POPL), 48:1–48:28 (2018). https://doi.org/10.1145/3158136
15. Gurfinkel, A., Kahsai, T., Komuravelli, A., Navas, J.A.: The seahorn verification framework. In: Kroening, D., Pasareanu, C.S. (eds.) Computer Aided Verification - 27th International Conference, CAV 2015, San Francisco, CA, USA, July 18-24, 2015, Proceedings, Part I. Lecture Notes in Computer Science, vol. 9206, pp. 343–361. Springer (2015). https://doi.org/10.1007/978-3-319-21690-4_20
16. Gurfinkel, A., Shoham, S., Meshman, Y.: Smt-based verification of parameterized systems. In: Zimmermann, T., Cleland-Huang, J., Su, Z. (eds.) Proceedings of the 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering, FSE 2016, Seattle, WA, USA, November 13-18, 2016. pp. 338–348. ACM (2016). https://doi.org/10.1145/2950290.2950330
17. Hajdu, Á., Jovanovic, D.: solc-verify: A modular verifier for solidity smart contracts. In: Chakraborty, S., Navas, J.A. (eds.) 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

18. Ip, C.N., Dill, D.L.: Better verification through symmetry. In: Agnew, D., Claesen, L.J.M., Camposano, R. (eds.) Computer Hardware Description Languages and their Applications, Proceedings of the 11th IFIP WG10.2 International Conference on Computer Hardware Description Languages and their Applications - CHDL '93, sponsored by IFIP WG10.2 and in cooperation with IEEE COMPSOC, Ottawa, Ontario, Canada, 26-28 April, 1993. IFIP Transactions, vol. A-32, pp. 97–111. North-Holland (1993)
19. Jiang, B., Liu, Y., Chan, W.K.: Contractfuzzer: fuzzing smart contracts for vulnerability detection. In: Huchard, M., Kästner, C., Fraser, G. (eds.) Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering, ASE 2018, Montpellier, France, September 3-7, 2018. pp. 259–269. ACM (2018). https://doi.org/10.1145/3238147.3238177
20. Kaiser, A., Kroening, D., Wahl, T.: Dynamic cutoff detection in parameterized concurrent programs. In: Touili, T., Cook, B., Jackson, P.B. (eds.) Computer Aided Verification, 22nd International Conference, CAV 2010, Edinburgh, UK, July 15-19, 2010. Proceedings. Lecture Notes in Computer Science, vol. 6174, pp. 645–659. Springer (2010). https://doi.org/10.1007/978-3-642-14295-6_55
21. Kalra, S., Goel, S., Dhawan, M., Sharma, S.: ZEUS: analyzing safety of smart contracts. In: 25th Annual Network and Distributed System Security Symposium, NDSS 2018, San Diego, California, USA, February 18-21, 2018. The Internet Society (2018)
22. Khalimov, A., Jacobs, S., Bloem, R.: Towards efficient parameterized synthesis. In: Giacobazzi, R., Berdine, J., Mastroeni, I. (eds.) Verification, Model Checking, and Abstract Interpretation, 14th International Conference, VMCAI 2013, Rome, Italy, January 20-22, 2013. Proceedings. Lecture Notes in Computer Science, vol. 7737, pp. 108–127. Springer (2013). https://doi.org/10.1007/978-3-642-35873-9_9
23. Kildall, G.A.: A unified approach to global program optimization. In: Fischer, P.C., Ullman, J.D. (eds.) Conference Record of the ACM Symposium on Principles of Programming Languages, Boston, Massachusetts, USA, October 1973. pp. 194–206. ACM Press (1973). https://doi.org/10.1145/512927.512945
24. Kolb, J.: A Languge-Based Approach to Smart Contract Engineering. Ph.D. thesis, University of California at Berkeley, USA (2020)
25. Kolluri, A., Nikolic, I., Sergey, I., Hobor, A., Saxena, P.: Exploiting the laws of order in smart contracts. In: Zhang, D., Møller, A. (eds.) Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2019, Beijing, China, July 15-19, 2019. pp. 363–373. ACM (2019). https://doi.org/10.1145/3293882.3330560
26. Krupp, J., Rossow, C.: teether: Gnawing at ethereum to automatically exploit smart contracts. In: Enck, W., Felt, A.P. (eds.) 27th USENIX Security Symposium, USENIX Security 2018, Baltimore, MD, USA, August 15-17, 2018. pp. 1317–1333. USENIX Association (2018)
27. Luu, L., Chu, D., Olickel, H., Saxena, P., Hobor, A.: Making smart contracts smarter. In: Weippl, E.R., Katzenbeisser, S., Kruegel, C., Myers, A.C., Halevi, S. (eds.) Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, Vienna, Austria, October 24-28, 2016. pp. 254–269. ACM (2016). https://doi.org/10.1145/2976749.2978309
28. Marescotti, M., Otoni, R., Alt, L., Eugster, P., Hyvärinen, A.E.J., Sharygina, N.: Accurate smart contract verification through direct modelling. In: Margaria, T., Steffen, B. (eds.) Leveraging Applications of Formal Methods, Verification and Validation: Applications - 9th International Symposium on Leveraging Applications of

Formal Methods, ISoLA 2020, Rhodes, Greece, October 20-30, 2020, Proceedings, Part III. Lecture Notes in Computer Science, vol. 12478, pp. 178–194. Springer (2020). https://doi.org/10.1007/978-3-030-61467-6_12

29. Mavridou, A., Laszka, A., Stachtiari, E., Dubey, A.: Verisolid: Correct-by-design smart contracts for ethereum. In: Goldberg, I., Moore, T. (eds.) Financial Cryptography and Data Security - 23rd International Conference, FC 2019, Frigate Bay, St. Kitts and Nevis, February 18-22, 2019, Revised Selected Papers. Lecture Notes in Computer Science, vol. 11598, pp. 446–465. Springer (2019). https://doi.org/10.1007/978-3-030-32101-7_27
30. 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: 34th IEEE/ACM International Conference on Automated Software Engineering, ASE 2019, San Diego, CA, USA, November 11-15, 2019. pp. 1186–1189. IEEE (2019). https://doi.org/10.1109/ASE.2019.00133
31. Namjoshi, K.S., Trefler, R.J.: Parameterized compositional model checking. In: Chechik, M., Raskin, J. (eds.) Tools and Algorithms for the Construction and Analysis of Systems - 22nd International Conference, TACAS 2016, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2016, Eindhoven, The Netherlands, April 2-8, 2016, Proceedings. Lecture Notes in Computer Science, vol. 9636, pp. 589–606. Springer (2016). https://doi.org/10.1007/978-3-662-49674-9_39
32. Nikolic, I., Kolluri, A., Sergey, I., Saxena, P., Hobor, A.: Finding the greedy, prodigal, and suicidal contracts at scale. In: Proceedings of the 34th Annual Computer Security Applications Conference, ACSAC 2018, San Juan, PR, USA, December 03-07, 2018. pp. 653–663. ACM (2018). https://doi.org/10.1145/3274694.3274743
33. Owicki, S.S., Gries, D.: An axiomatic proof technique for parallel programs I. Acta Informatica **6**, 319–340 (1976). https://doi.org/10.1007/BF00268134
34. Permenev, A., Dimitrov, D., Tsankov, P., Drachsler-Cohen, D., Vechev, M.T.: 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
35. Pnueli, A., Ruah, S., Zuck, L.D.: Automatic deductive verification with invisible invariants. In: Margaria, T., Yi, W. (eds.) Tools and Algorithms for the Construction and Analysis of Systems, 7th International Conference, TACAS 2001 Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2001 Genova, Italy, April 2-6, 2001, Proceedings. Lecture Notes in Computer Science, vol. 2031, pp. 82–97. Springer (2001). https://doi.org/10.1007/3-540-45319-9_7
36. Siegel, S.F., Avrunin, G.S.: Verification of mpi-based software for scientific computation. In: Graf, S., Mounier, L. (eds.) Model Checking Software, 11th International SPIN Workshop, Barcelona, Spain, April 1-3, 2004, Proceedings. Lecture Notes in Computer Science, vol. 2989, pp. 286–303. Springer (2004). https://doi.org/10.1007/978-3-540-24732-6_20
37. Siegel, S.F., Gopalakrishnan, G.: Formal analysis of message passing - (invited talk). In: Jhala, R., Schmidt, D.A. (eds.) Verification, Model Checking, and Abstract Interpretation - 12th International Conference, VMCAI 2011, Austin, TX, USA, January 23-25, 2011. Proceedings. Lecture Notes in Computer Science, vol. 6538, pp. 2–18. Springer (2011). https://doi.org/10.1007/978-3-642-18275-4_2
38. So, S., Lee, M., Park, J., Lee, H., Oh, H.: VERISMART: A highly precise safety verifier for ethereum smart contracts. In: 2020 IEEE Symposium on Security and

Privacy, SP 2020, San Francisco, CA, USA, May 18-21, 2020. pp. 1678–1694. IEEE (2020). https://doi.org/10.1109/SP40000.2020.00032
39. Tsankov, P., Dan, A.M., Drachsler-Cohen, D., Gervais, A., Bünzli, F., Vechev, M.T.: Securify: Practical security analysis of smart contracts. In: Lie, D., Mannan, M., Backes, M., Wang, X. (eds.) Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security, CCS 2018, Toronto, ON, Canada, October 15-19, 2018. pp. 67–82. ACM (2018). https://doi.org/10.1145/3243734.3243780
40. Wang, S., Zhang, C., Su, Z.: Detecting nondeterministic payment bugs in ethereum smart contracts. Proc. ACM Program. Lang. **3**(OOPSLA), 189:1–189:29 (2019). https://doi.org/10.1145/3360615
41. 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)
42. Wesley, S., Christakis, M., Navas, J.A., Trefler, R.J., Wüstholz, V., Gurfinkel, A.: Compositional verification of smart contracts through communication abstraction (extended). CoRR **abs/2107.08583** (2021), https://arxiv.org/abs/2107.08583
43. Wüstholz, V., Christakis, M.: Harvey: a greybox fuzzer for smart contracts. In: Devanbu, P., Cohen, M.B., Zimmermann, T. (eds.) ESEC/FSE '20: 28th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, Virtual Event, USA, November 8-13, 2020. pp. 1398–1409. ACM (2020). https://doi.org/10.1145/3368089.3417064
44. 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: Lahiri, S.K., Wang, C. (eds.) Computer Aided Verification - 32nd International Conference, CAV 2020, Los Angeles, CA, USA, July 21-24, 2020, Proceedings, Part I. Lecture Notes in Computer Science, vol. 12224, pp. 137–150. Springer (2020). https://doi.org/10.1007/978-3-030-53288-8_7