Smart Contract Analysis Through Communication Abstractions

by

Arthur Scott Wesley

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Master of Mathematics
in
Computer Science

Waterloo, Ontario, Canada, 2021

Author's Declaration

This thesis consists of material all of which I authored or co-authored: see Statement of Contributions included in the thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.

Statement of Contributions

Some of the text, figures, and tables in this thesis are restated from our publication in SAS 2021 [72]. The fuzzing source code in SmartACE was co-authored by Xinwen Hu.

Abstract

Smart contracts are programs that manage interactions between many users. Recently, SOLIDITY smart contract have become a popular way to enforce financial agreements between untrusting users. However, such agreements do not eliminate trust, but rather redirects trust into the correctness of the smart contract. This means that each user must verify that a smart contract behaves correctly, regardless of how other users interact with it. Verifying a smart contract relative to all possible users is intractable due to state explosion. This thesis studies how local symmetry can be used to analyze smart contracts from a few representative users.

This thesis builds on the novel notion of *participation*, that gives explicit semantics to user interactions. From participation, a topology is obtained for how users interact during each transaction of a smart contract. Local symmetry analysis shows that most users are interchangeable within a topology, and therefore, most users are locally symmetric. This motivates *local bundle abstractions* that reduce contracts with arbitrarily many users to sequential programs with a few representative users. It is shown that local bundle abstractions can be used to ameliorate state explosion in smart contract verification, and to accelerate counterexample search in bounded analysis (e.g., fuzzing and bounded model checking). We implement local bundle abstraction in SMARTACE, and show order-of-magnitude improvements in time when compared to a state-of-the-art smart contract verification tool.

Acknowledgements

First, I would like to offer my greatest thanks to my co-supervisors, Dr. Arie Gurfinkel and Dr. Richard Trefler, for their guidance, insight, and feedback. Without this involvement, I would not have been able to complete this thesis. Their collaboration has been indispensable.

Second, I would like to thank my professional collaborators, Dr. Maria Christakis, Dr. Jorge A. Navas, and Dr. Valentin Wüstholz, for their involvement during my master program. Specifically, I extend my thanks for their feedback and contributions to our publication in SAS 2021 [72].

Third, I would like to thank the readers of this thesis, Dr. Florian Kerschbaum and Dr. Chengnian Sun, for the time they have dedicate to its review.

Fourth, I would like to thank Quantstamp for providing access to real-world smart contract problems.

Dedication

To my parents, Brain and Lorraine, for your unending support. To my friends, Sam and Tanisha, for your patience and understanding. In loving memory of Nilla, 2020–2021.

Table of Contents

List of Figures

List of Tables

List of Abbreviations

EECF Effectively External Callback Free.

LTS Labeled Transition System.

MSL MiniSol Specification Language.

PCMC Parameterized Compositional Model Checking.

PT Participation Topology.

PTG Participation Topology Graph.

SCUN Synchronized Control User Network.

Chapter 1

Introduction

Smart contracts were proposed in 1996 [67] as reactive state machines that managed legal assets. The first large-scale smart contract system was realized in 2008 through the Bitcoin peer-to-peer blockchain protocol [51]. Bitcoin has inspired other financial smart contract systems, such as Zilliqa [63], Algorand [24], Facebook Libra [11], and Ethereum. Ethereum has seen widespread acceptance, resulting in a capitalization of 1 billion dollars (USD) within the first 2 years following its creation [6]. Ethereum is an open, decentralized compute framework which allows developers to deploy Turing-complete smart contracts [73], often written in the SOLIDITY language.

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).

Once a SOLIDITY smart contract is deployed to Ethereum, it is immutable [73]. For this reason, smart contracts have become a popular way to enforce financial agreements between untrusting users. However, such agreements do not eliminate trust, but rather redirects trust into the correctness of the smart contract. Furthermore, since smart contracts are immutable, a mistake in the design of a smart contract cannot be corrected. This means that each user must verify that a smart contract behaves correctly, regardless of how other users interact with it.

A direct solution to smart contract verification is to verify the finite-state system di-

rectly. The direct approach is used by tools such as VerX [58], and results in analysis times of up to 11 hours. This is because verifying systems with at least $2^{2^{160}}$ states is intractable in general. Other tools, such as [32, 36, 75] use fuzzing to randomly test a smart contract, often with a restricted number of users or a restricted number of transactions. This is also insufficient, as testing can never ensure correctness.

In this thesis, we propose a solution to efficient smart contract analysis through the theory of local symmetry (adopted from parameterized compositional model checking [54]). Intuitively, correctness is inferred from a small number of representative users to ameliorate state explosion. These representatives are locally symmetric, as explained below. 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.

Our key insight is that for many smart contracts, each transaction interacts with only a fixed number of users. These users are called *participants* of a transaction. Our analysis is performed *locally* to a transaction and its participants. We show that in most transactions, most participants can be switched with a non-participating user without impacting the outcome of the transaction. These users are said to be locally symmetric. From local symmetry, proof rules are obtained for parameterized smart contract verification, and a procedure is obtained to accelerate bounded analysis. We implement these techniques in an open-source tool named SmartACE.

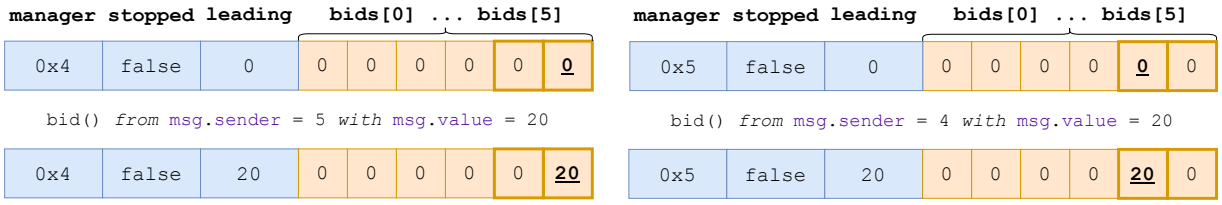
1.1 Motivating Example

Local symmetry analysis is applicable to many smart contracts. For example, consider `SimpleAuction` in Fig. 1.1. This Solidity smart contract enforces the rules of an open-bid auction (i.e., all bids are public). In `SimpleAuction`, each user starts with a bid of zero. Users alternate, and submit increasing larger bids, until a designated manager stops the auction. While the auction is not stopped, a non-leading user may withdraw their bid. To ensure that the auction is fair, a manager is not allowed to place their own bid. Furthermore, the role of manager is never assigned to the *zero-account* (i.e., the null user at address 0).

An important property of any open-bid auction is that subsequent bids are increasing in value. In `SimpleAuction`, this property is ensured by tracking the *leading bid*. This example shows that `SimpleAuction` satisfies **Prop. 0**: "*All bids are less than or equal to the recorded leading bid*". In Section 6.2, this example is revisited to show that there is always a user whose bid is equal to the leading bid.

```
/// @title A simple open-bid auction.
contract Auction {
  mapping(address => uint) bids;
  address manager;
  uint leadingBid;
  bool stopped;

  /// Initiates a new auction controlled by _manager.
  constructor(address _manager) public {
    require(manager != address(0));
    manager = _manager;
  }

  /// Returns a non-leading bid.
  function withdraw() public {
    require(!stopped);
    require(msg.sender != manager);

    uint bid = bids[msg.sender];
    require(bid < leadingBid);
    bids[msg.sender] = 0;

    msg.sender.transfer(bid);
  }

  /// Processes a valid bid.
  /// A bid is valid if it:
  /// 1. The auction is not stopped.
  /// 2. The sender is not the manager.
  /// 3. The sender is not the leading bidder.
  /// 4. The bid exceeds the current leading bid.
  function bid() public payable {
    require(!stopped); // (1)
    require(msg.sender != manager); // (2)

    uint oldBid = bids[msg.sender];
    uint newBid = oldBid + msg.value;
    require(oldBid == 0 || oldBid < leadingBid); // (3)
    require(newBid > leadingBid); // (4)

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

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

Figure 1.1: A simple open-bid auction smart contract intended to illustrate local smart contract analysis.



(a) A bid of 20 from user 5 with manager 4.

(b) A bid of 20 from user 4 with manager 5.

Figure 1.2: Illustration of a local transaction symmetry for `SimpleAuction` in .

To establish **Prop. 0**, it must be shown that **Prop. 0** is initially true, and that **Prop. 0** continues to be true after any sequence of transactions. A *software model checker*, such as SeaHorn [28], can establish this property automatically by enumerating the reachable states of `SimpleAuction`. However, the state space of `SimpleAuction` is intractable large. For example, the mapping `bids` must store 2^{160} bids, each of size 256-bits. Our hypothesis is that tools such as SeaHorn struggle to scale to properties such as **Prop. 0** due to the impact of users on the size of the state space. To overcome this challenge, we first instead identify groups of communicating users, and then establish **Prop. 0** relative to a *representative* (i.e., abstract) group of communicating users.

In the case of `SimpleAuction`, all communication takes place between the zero-account, the auction itself, the manager, and an arbitrary sender. The zero-account and auction are unique, whereas any arbitrary user may be a manager or a sender. However, these

managers and senders are all *interchangeable* with respect to **Prop. 0**. That is to say, if two arbitrary users have the same bid, then swapping their addresses and manager status preserves **Prop. 0**. For example, assume that all users have a bid of zero, the manager is the user with address 4, and the user with address 5 places a bid of 20. Then after this transaction, the manager continues to have a bid of 0, the sender now has a bid of 20, and the leading bid is set to 20. Since all other bids were initially zero, and were not *affected* by this transaction, then **Prop. 0** is satisfied. The same outcome is obtained if all users have a bid of 0, the manager is the user with address 5, and the user with address 4 places a bid of 20. In other words, the satisfiability of **Prop. 0** is preserved after swapping user 4 and 5. This symmetry is illustrated in Fig. 1.2, and formalized in Section 5.2.

All users of `SimpleAuction` can then be classified as either the zero-account, the auction itself, the current manager, or an arbitrary sender. Users within the same class are symmetric with respect to **Prop. 0**. Therefore, it is sufficient to verify **Prop. 0** relative to a representative user from each class. The key idea is that each representative summarizes the concrete users in its class [54]. 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 and the auction each correspond to single concrete users (this generalizes to *implicit participation* in Section 4.1). Similarly, the manager refers to a single concrete user, so long as the `manager` variable does not change (this generalizes to *transient participation* in Section 4.1). Therefore, 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 `SimpleAuction` only compares addresses by equality, 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 auction, 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, called an *interference invariant*, that is true of all initial bids, and holds inductively for bids across all transactions. We provide this invariant manually, and use it to over-approximate all possible bids. An obvious interference invariant for `SimpleAuction` is **Prop. 0**.

Given an interference invariant, **Prop. 0** can then be verified effectively by a software model checker. To do this, the concrete users in `SimpleAuction` must be abstracted by representative users. The abstract system (see Fig. 1.3), known as a *local bundle abstraction*, assigns the zero-account to address 0, the auction to address 1, the manager to address 2, the representative sender to address 3, and then executes an unbounded sequence of transactions. Before each transaction, the sender's bid is set to a nondeterministic value that satisfies its interference invariant. If the abstract system and **Prop. 0** are provided

```
1 // Configures address space.
2 address zacc_addr = address(0);
3 address auct_addr = address(1);
4 address mngr_addr = address(2);
5 address sndr_addr = address(3);
6
7 // Constructs contract.
8 SimpleAuction _a = new SimpleAuction(mngr_addr);
9 require(address(_a) == auct_addr);
10
11 // Transaction loop.
12 while (true) {
13   // Applies an interference invariant.
14   _a.bids[sndr_addr] = *;
15   require(_a.bids[sndr_addr] <= _a.leadingBid);
16
17   // Selects a sender.
18   msg.sender = *;
19   require(msg.sender >= mngr_addr);
20   require(msg.sender <= sndr_addr);
21
22   // Selects a transaction.
23   if      (*) { msg.value = *; _a.bid(); };
24   else if (*) { msg.value = 0; _a.withdraw(); }
25   else if (*) { msg.value = 0; _a.stop(); }
26 }
```

Figure 1.3: A harness to establish **Prop. 0** on `SimpleAuction`. Each `*` is nondeterministic choice of value.

to a software model checker, then the software model checker will verify that all states reachable in the abstract system satisfy **Prop. 0**. As a result, **Prop. 0** is established for `SimpleAuction`.

1.2 Scope of Thesis

Two unique challenges in SOLIDITY development are gas constraints and reentrancy. In Ethereum, gas is a payment made to execute a smart contract. Gas is paid upfront, but computed at runtime. This means that a smart contract may abort unexpectedly if insufficient gas is paid. Reentrancy is what happens when one smart contract recursively interacts with another smart contract. Overlooking gas constraints and reentrancy can lead to subtle bugs that are hard to find manually.

Both gas and reentrancy are outside of the scope of this thesis. However, the techniques described in this thesis do not preclude solutions to these problems. We have chosen to place gas and reentrancy out of scope as there are many existing solutions to these problems. For example, [25] is a static technique that detects gas related vulnerabilities. For reentrancy, [27] describes static and dynamic techniques to determine if a smart contract is vulnerable to reentrancy. In [58], Effectively External Callback Free (EECF) contracts are suggested as a design pattern to prevent reentrancy vulnerabilities.

1.3 Summary of Contributions

This thesis makes the following contributions:

- We propose a new approach to smart contract analysis. In this paradigm, smart contracts are parameterized and all communication between users is explicit. Our work is based on the local symmetry analysis of parameterized compositional model checking. We use this analysis to obtain a local abstraction of a smart contract, with applications to parameterized smart contract verification and bounded analysis.
- We extend parameterized compositional model checking in two ways. First, we introduce aggregate properties, and show that they can verified using compositional techniques. Second, we show that local symmetry analysis is applicable to problems outside of parameterized verification.
- We implement local abstractions in an open-source tool named SmartACE. The tool is designed to target both parameterized verification and fuzzing. We evaluate SmartACE on real-world smart contracts, and show order-of-magnitude improvements over a state-of-the-art verification tool.

1.4 Outline of Thesis

The rest of this thesis is structured as follows:

- **Chapter 2** provides preliminary knowledge for this thesis;
- **Chapter 3** defines MiniSol, a subset of Solidity with network semantics and a parameterized property language;
- **Chapter 4** introduces explicit communication semantics for Solidity smart contracts, and defines syntactic over-approximations that can be computed automatically for MiniSol smart contracts;
- **Chapter 5** studies the local symmetries that exist in MiniSol communication, and presents a *local bundle abstraction* that is inspired by these symmetries;
- **Chapter 6** proves that local bundle abstractions can be used to automate parameterized verification of MiniSol smart contracts;
- **Chapter 7** shows that local bundle abstractions are not limited to parameterized verification, by demonstrating an application to bounded analysis;
- **Chapter 8** describes the implementation SmartACE;

- **Chapter 9** evaluates SmartACE on several real-world smart contracts;
- **Chapter 10** provides concluding remarks, and suggests future work.

Chapter 2

Background

This chapter reviews key concepts used freely throughout the subsequent chapters. First, notation is given for vector operations, and then relevant concepts are recalled from group theory, order theory, graph theory, and model theory. Building on these concepts, relevant models of computation are then defined. Key static analysis techniques are reviewed for the aforementioned models.

The following notation is used in all chapters. For $n \in \mathbb{N}$, the set $\{0, 1, \ldots, n-1\}$ is written $[n]$. For a Boolean expression e, and numeric expressions x and y, the piece-wise function "If e is true, then return x, else return y," is written $\mathsf{ite}(e, x, y)$.

2.1 Vectors and Vector Manipulations

We write $\mathbf{u} := (u_0, u_1, \ldots, u_{n-1})$ for a vecter of n elements. For the length of $\mathbf{u}$, we write $|\mathbf{u}|$. For the i-th element of $\mathbf{u}$, we write $\mathbf{u}_i$.

Let $f : S \to T$. The *point-wise application of f onto* $\mathbf{u}$ is the vector $\mathbf{u}'$ obtained by applying f to each element of $\mathbf{u}$. Formally, $\mathbf{u}' := (f(\mathbf{u}_0), f(\mathbf{u}_1), \ldots, f(\mathbf{u}_{n-1}))$ and we write that $f(\mathbf{u}) = \mathbf{u}'$.

Let $x \in S$. The *substitution of x for the i-th element of* $\mathbf{u}$ is the vector $\mathbf{u}'$ obtained by replacing the i-th element of $\mathbf{u}$ with x. Formally, $\mathbf{u}' := (\mathbf{u}_0, \mathbf{u}_1, \ldots, \mathbf{u}_{i-1}, x, \mathbf{u}_{i+1}, \ldots, \mathbf{u}_{n-1})$ and we write that $\mathbf{u}[i \leftarrow x] = \mathbf{u}'$.

Let $f : [n] \to [n]$ be a one-to-one function. The *permutation of* $\mathbf{u}$ *induced by* f is the vector $\mathbf{u}'$ obtained by reordering the elements of $\mathbf{u}$ according to $i \mapsto f(i)$. Formally, $\mathbf{u}' := \left(\mathbf{u}_{f(0)}, \mathbf{u}_{f(1)}, \ldots, \mathbf{u}_{f(n)}\right)$.

Let $f : S \times S \to S$ and $z \in S$. The *fold* operation is a higher-order function that recursively accumulates the elements of $\mathbf{u}$ using the operator f. In the base case, $|\mathbf{u}| = 1$ and fold returns $f(z, \mathbf{u}_0)$. Formally:

$$\mathsf{fold}(f, z, \mathbf{u}) := \begin{cases} f\left(z, \mathbf{u}_0\right) & \text{if } |\mathbf{u}| = 1 \\ f\left(\mathsf{fold}(f, z, \mathbf{u}'), \mathbf{u}_{|\mathbf{u}|-1}\right) & \text{if } |\mathbf{u}| > 1 \text{ and } \mathbf{u}' = \left(\mathbf{u}_0, \ldots, \mathbf{u}_{|\mathbf{u}|-2}\right) \end{cases}$$

For example, if $\mathbf{u} = (1, 3, 5)$ and $f(x, y) = x + y$, then $\mathsf{fold}(f, 0, \mathbf{u}) = ((0 + 1) + 3) + 5 = 9$.

2.2 Group Theory

This section briefly reviews permutation groups and Abelian groups. The presentation follows [22], except for the discussion on permutation groups, which follows [43]. A *group*, $G = (S, \circ)$, is a possibly infinite set, S, equipped with a total function, $\circ : S \times S \to S$, satisfying the following axioms:

1. (Associativity) $\forall a, b, c \in S \cdot (a \circ b) \circ c = a \circ (b \circ c)$;
2. (Identity) $\exists e \in S \cdot \forall a \in S \cdot a \circ e = a = e \circ a$, where e is the *identity* of G;
3. (Inverse) $\forall a \in S \cdot \exists a^{-1} \in S \cdot a \circ a^{-1} = e = a^{-1} \circ a$, where a^{-1} is the *inverse* of a.

The operator $\circ$ is *commutative* if $\forall a, b \in S \cdot a \circ b = b \circ a$. If $\circ$ is commutative, then G is an *Abelian group*.

Many operations form groups, such as matrix multiplication and modular arithmetic. An important group in later chapters is the group of permutations for a set $[n]$. A *permutation* is a one-to-one mapping $f : [n] \to [n]$. If S is a set of permutations for $[n]$ that is closed under function composition, then S equipped with function composition forms a *permutation group*. A surprising result is that every permutation group can be constructed using adjacent transpositions. For $i \in [n - 1]$, the *transposition* is the permutation $\tau_i : [n] \to [n]$ such that:

$$\tau_i(x) = \begin{cases} i + 1 & \text{if } x = i \\ i & \text{if } x = i + 1 \\ x & \text{otherwise} \end{cases}$$

In a group of permutations, the identity function is the identity, and $\tau_i^{-1} = \tau_i$. Using transpositions, it is easy to prove that permutation groups are non-Abelian in general.

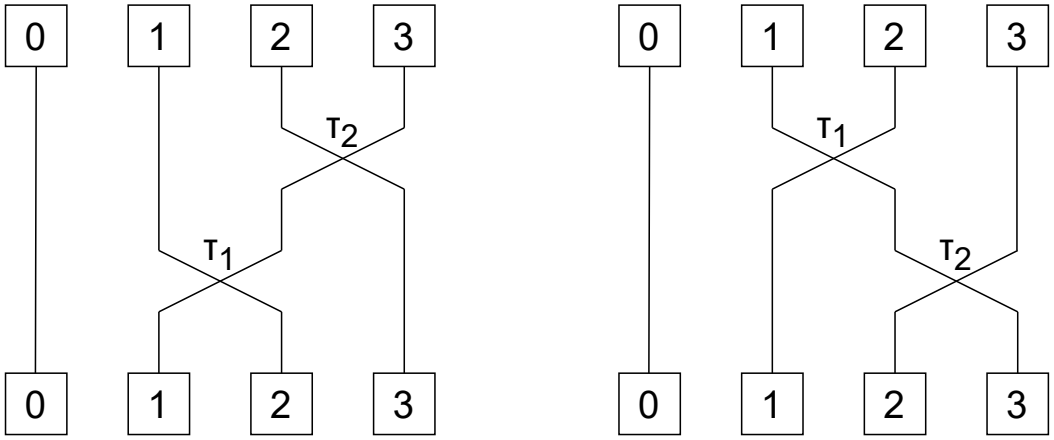


Figure 2.1: An illustration that not all permutation groups are Abelian. In this example, $n = 4$. It is shown that $\tau_2 \circ \tau_1 \neq \tau_1 \circ \tau_2$. This diagram follows the presentation of [43]. Note that the elements are propagated bottom to top.

For example, $\tau_2(\tau_1(1)) = 3$ whereas $\tau_1(\tau_2(1)) = 2$. As a result, not all τ_i commute over function composition. The key properties of transpositions are illustrated in Fig. 2.1 and summarized in Proposition 1.

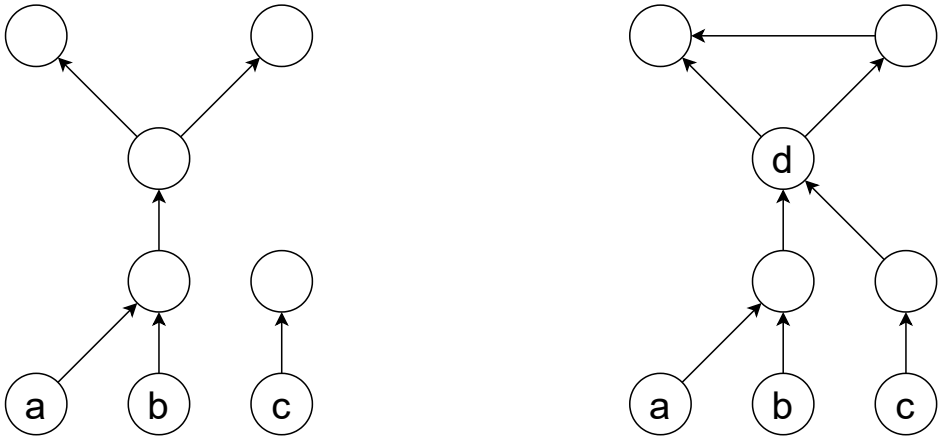
Proposition 1. Let $n \in \mathbb{N}$, and $f : [n] \to [n]$ be a permutation. For some $k \in \mathbb{N}$, there exists $\{i_1, \ldots, i_k\} \in [n-1]$ such that $f = \tau_{i_1} \circ \cdots \circ \tau_{i_k}$.

A well-known example of an Abelian group is the set of integers equipped with addition. Formally, $G = (\mathbb{Z}, +)$. It is not hard to see that $\forall a, b, c \in \mathbb{Z} \cdot a + (b + c) = (a + b) + c$ and $\forall a, b \in \mathbb{Z} \cdot a + b = b + a$. The identity is 0, since $\forall a \in \mathbb{Z} \cdot a + 0 = a = 0 + a$. The inverse of $a \in \mathbb{Z}$ is $(-a)$ since $(-a) \in \mathbb{Z}$ and $a + (-a) = 0$.

2.3 Order Theory and Lattices

This section briefly reviews the concept of lattices from order theory. The presentation summarizes the first three chapters of [62]. A *partially ordered set*, $(S, \leq)$, is a possibly infinite set, S, equipped with a relation, $(\leq) \subseteq S \times S$, satisfying the following axioms:

1. (Reflexivity) $\forall a \in S \cdot a \leq a$;
2. (Antisymmetry) $\forall a, b \in S \cdot (a \leq b \wedge b \leq a) \Rightarrow (a = b)$;



(a) A partially ordered set that is not a lattice. (b) A partially ordered set that is a lattice.

Figure 2.2: A comparison between two partially ordered sets. An arrow from element x to element y denotes that $x \leq y$. The order in Fig. 2.2b forms a join-semilattice as each finite subset has a supremum.

3. (Transitivity) $\forall a, b, c \in S \cdot (a \leq b \wedge b \leq c) \Rightarrow (a \leq b)$.

For $X \subseteq S$ and $a \in S$, we write $X \leq a$ to denote $\forall x \in X \cdot x \leq a$. The *supremum* of X, denoted by $\sup\{X\}$, is an element $a \in S$, such that $X \leq a$ and for every $b \in S$, if $X \leq b$, then $a \leq b$. For example, the supremum of $\{a, b, c\}$ in Fig. 2.2b is d. If $\sup\{X \cup Y\}$ is defined, then $\sup\{X \cup Y\} = \sup\{\sup\{X\}, \sup\{Y\}\}$.

In general, the supremum of X might not exist. For example, there is no supremum for $\{a, b, c\}$ in Fig. 2.2a. If every finite subset of S has a supremum, then $(S, \leq)$ is a *join-semilattice*. An example of a join-semilattice is given in Fig. 2.2b. Throughout this paper, all lattices are assumed to be join-semilattices. All results generalize naturally to meet-semilattices (see [62]).

2.4 Graph Theory

This section briefly reviews key concepts from graph theory. The presentation follows [16], unless stated otherwise. A *graph* is a tuple $G = (V, E)$, where V is a finite set of objects, called *vertices*, and E is a set of two element subsets from V, called *edges*. A *digraph* is a tuple $D = (V, E)$, where V is a finite set of vertices, and $E \subseteq V \times V$. In either case, $v_1 v_2$ is written to denote an edge from v_1 to v_2.

A *path* in a (di)graph (V, E) is a set of vertices $\{v_0, \ldots, v_k\} \subseteq V$, and a set of edges $\{v_i v_{i+1} \mid i \in [k]\} \subseteq E$. A vertex $u \in V$ is *reachable* from a vertex $u' \in V$ if there exists a path with $v_0 = u$ and $v_k = u'$.

A *bipartite (di)graph* is a (di)graph $(V_1 \cup V_2, E)$ such that for all $v_1 v_2 \in E$, both $v_1 \in V_1$ and $v_2 \in V_2$.

A *labelled (di)graph* associates one or more labels with each vertex. Let L be the set of labels. In this thesis, a labelled graph is a tuple (V, E, δ) such that (V, E) is a (di)graph and $\delta \subseteq E \times L$. If $(e, l) \in \delta$, then $e \in E$ is labelled by $l \in L$.

2.5 Labeled Transition Systems and Properties

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[1]) $\varphi \subseteq S$. 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 array cells. In both cases, there is a single *control process* that interacts with a number of *user processes*. Such systems are called *Synchronized Control User Network (SCUN)* [54]. Let N be the number of processes, and $[N]$ be the process identifiers. We consider SCUNs in which processes 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 $S := S_C \times (S_U)^N$, $P := P_I \cup (P_S \times [N])$, $s_0 := (c_0, u_0, \ldots, u_0)$, and $T : S \times P \to S$ such that:

[1] 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.

1. If $p \in P_I$, then $T((c, \mathbf{u}), p) = (T_I(c, p), \mathbf{u})$;

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 systems have parameterized properties [29, 54]. A *k-universal safety property* [29] is a predicate $\varphi \subseteq S_C \times (S_U)^k$. A state $(c, \mathbf{u})$ satisfies predicate φ if $\forall \{i_1, \ldots, i_k\} \subseteq [N] \cdot \varphi(c, \mathbf{u}_{i_1}, \ldots, \mathbf{u}_{i_k})$. A parameterized system $M(\cdot)$ satisfies predicate φ if $\forall N \in \mathbb{N} \cdot M(N) \models \varphi$.

So far, this section has been limited to safety properties. However, many interesting smart contract properties deal with either the past or the future. These are called temporal properties, and subsume safety properties [59]. Formally, a safety property is a temporal formula of the form $\mathsf{always}(\varphi)$, where φ is a property that must *always* be true of the current state. In this thesis, we consider pure-past properties, such as "*If a participant of an auction has not placed a bid, then the participation has not withdrawn a bid.*" Pure-past properties are constructed from the temporal operators $\mathsf{once}(\cdot)$, $\mathsf{historically}(\cdot)$, $\mathsf{since}(\cdot, \cdot)$, and $\mathsf{prev}(\cdot)$ where:

1. The expression $\mathsf{once}(\varphi)$ is true if φ was true in at least one prior state;

2. The expression $\mathsf{historically}(\varphi)$ is true if φ was true in all prior states;

3. The expression $\mathsf{since}(\varphi, \psi)$ is true if either φ was false in all prior states, or ψ has been true since φ was first true;

4. The expression $\mathsf{prev}(\varphi)$ is true if φ was true in the previous state.

Prior work has shown pure-past properties to be a convenient specification language for controllers that interact with an environment [14, 58]. Furthermore, it is known that the verification of a pure-past property for one LTS can be reduced to the safety of a larger LTS [31].

2.6 Simulations and Bisimulations

This section briefly reviews simulation and bisimulation, following the presentation of [64]. Let $M = (S, P, T, s_0)$ and $M^* = (S^*, P, T^*, s_0^*)$ be LTSs. A *simulation* of M by M^* is a relation $\sigma \subseteq S \times S^*$ satisfying the following properties:

1. $(s_0, s_0^*) \in \sigma$;

2. If $(s, p, t) \in T$, then there exists an $(s^*, p, t^*) \in T^*$ such that $(t, t^*) \in \sigma$.

If σ^{-1} is a simulation of M^* by M, then σ is a *bisimulation*.

Simulations and bisimulations are used to prove properties about M and M^*. A simulation σ is said to *preserve* a safety property φ if $\forall (s, s^*) \in \sigma \cdot s \models \varphi \iff s^* \models \varphi$. If σ preserves φ and $M^* \models \varphi$, then $M \models \varphi$. As a direct consequence, if σ preserves φ and σ is a bisimulation, then $(M \models \varphi) \iff (M^* \models \varphi)$.

2.7 Parameterized Compositional Model Checking

Model checking is an approach that automatically proves a transition system satisfies a property (see [59]). In this section, the discussion is limited to LTSs. The inputs to a model checker are a LTS and a property for the LTS. A model checker then enumerates traces until a violation of the property is discovered, or all traces are shown to be free from violations. The output of a model checker is either a trace of the LTS that violates the property, or an inductive invariant of the LTS that entails the property.

Extending model checking to parameterized transition systems is non-trivial, as the parameter space is infinite. Two solutions are *bounded model checking* and *parameterized model checking*. In bounded model checking, the parameter space is restricted to a finite subset, and then the transition system is verified for only these parameters (e.g., [60]). In parameterized model checking, specialized techniques are use to generalize the results of model checking to all possible parameters (see [2]). This thesis considers *Parameterized Compositional Model Checking (PCMC)*.

The results of PCMC rely on equivalence classes. An *equivalence relation* [62] on a set S is a relation $\sim \subseteq S \times S$ that satisfies the following axioms:

1. (Reflexivity) $\forall a \in S \cdot a \sim a$;

2. (Symmetry) $\forall a, b \in S \cdot (a \sim b) \iff (b \sim a)$;

3. (Transitivity) $\forall a, b, c \in S \cdot (a \sim b \wedge b \sim c) \Rightarrow (a \sim c)$.

An example of an equivalence relation is equality. For each $a \in S$, the *equivalence class* [62] of a is the set of elements in S that are equivalence to a, and is defined by $\{b \in S \cdot a \sim b\}$. Given an equivalence class E, if $a \in E$, the a is a *representative* of E.

The rest of this section briefly reviews the presentation of PCMC in [54]. In PCMC, a distinction is made between the *processes* executing within a parameterized transition system, and the *network topology*[2] on which they are placed. At a high-level, PCMC decomposes the network topology into smaller structures called *neighbourhoods*, and then identifies equivalencies between processes in different neighbourhoods called *local symmetries*. These equivalencies are formulated as a *balance relation*, that generalizes bisimulation relations to processes on a network topology.

Definition 2.7.1 (Balance Relation)**.** A *balance relation* B is a set of local symmetries satisfying the following properties. For any $(m, \beta, n) \in B$, where m and n are locally symmetric processes and β is the witnessing bijection from m to n:

1. Its inverse, (n, β^{-1}, m) is also in B;
2. For any j that points to n, there is a k which points to m, there exists a bijection δ from j to k such that δ, $(j, \delta, k) \in B$ and both β and δ agree on edges in the topology.

PCMC proceeds by discovering a *compositional invariant* for each equivalence class, and then proving that these invariants entail a parameterized property φ. As with a bisimulation relation, a balance relation must preserve φ in order for PCMC to be applicable. Often, compositional invariants are obtained by computing the strongest inductive invariant for each process in a sufficiently large network. The size of this network is a *compositional cutoff*.

For example, in a SCUN, there are user and control processes that are assigned to a network topology which connects all user processes to a single control process. Each neighbourhood consists of a single user interacting with a control process. The compositional invariant is defined 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 $(c', u) \in \theta_U$;

[2] A network topology consists of vertices, and edges between vertices. Processes execute on vertices, and share state across edges.

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 [54], if $\forall c \in \theta_C \cdot \forall\{(c, u_1), \ldots, (c, u_k)\} \subseteq \theta_U \cdot \varphi(c, u_1, \ldots, u_k)$, then $\forall N \in \mathbb{N} \cdot M(N) \models \varphi$. This is as an extension of Owicki-Gries [57], 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*.

Chapter 3

Smart Contracts as Parameterized Networks

This chapter generalizes SOLIDITY smart contracts to networks of arbitrarily many users. By generalizing a smart contract to any number of users, it is then possible to consider smart contracts with a reduced number of users (Chapter 5). The generalization is presented for a subset of SOLIDITY called MINISOL. Like SOLIDITY, MINISOL is an imperative, object-oriented language with built-in communication operations. However, MINISOL does not include features such as inheritance, cryptographic operations, or mappings between addresses. The semantics of MINISOL programs are given by SCUNs.

A class of parameterized properties are then defined for MINISOL programs. The property language is inspired by current work in smart contract analysis. Special attention is given to aggregate properties such as sum(·), count(·), min(·), and max(·), due to their importance in many smart contract specifications.

3.1 The MiniSol Language

MINISOL is an object-oriented language with built-in communication and monetary operations. Each class in MINISOL is called a *smart contract*, and consists of state variables, and methods for users to call. In addition to state variables, each smart contract also has an *address* and a *balance*. The address of a smart contract is a globally unique identifier that distinguishes the smart contract from all other smart contracts and users. The balance of

a smart contract is the amount of currency that the smart contract holds. In MiniSol, all currency is measured in Ether.

A *user* in MiniSol is an external entity that interacts with a smart contract. Users alternate, and invoke smart contract methods. Similar to a smart contract, each user also has an address. The user's address is used by a smart contract to map the user to said user's data. A null user is given by the address 0, and is called the *zero-account*.

A smart contract operates in a SCUN. The smart contract is the control process and the smart contract users are the user processes. The network is updated through a sequences of synchronized and internal actions, known as a *transaction*. A transaction begins when a user process invokes a method through a synchronized action. The control process then executes the method using one or more of internal actions. Throughout a transaction, the control process may invoke additional methods from the context of another smart contract. Furthermore, the control process may read from (or write to) a user's data via a synchronized action. For simplicity of presentation, each transaction is given as a global transition.

The main features of MiniSol are illustrated by the smart contracts `Auction` and `TimedAuctionManager` in Fig. 3.1. `Auction` implements an open-bid 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. At any point during the auction, a non-leading user may query the minimal bid required to become the leading bidder. `TimedAuctionManager` implements a simple protocol to manage `Auction`. Initially, `TimedAuctionManager` creates a new auction and sets a timeout. Once the timeout has elapsed, any user may stop the auction through a request to the manager. Throughout this chapter, Fig. 3.1 is used as a running example.

3.1.1 Syntax

This section presents the syntax for MiniSol. Program expressions are given by Fig. 3.2 and program statements are given by Fig. 3.3.

MiniSol has four basic types: *address*, *numeric*, *mapping*, and *contract*. The address-type prevents arithmetic operations, and numeric values cannot be converted (i.e., cast) to address values. The address-type corresponds to ⟨AExpr⟩ in Fig. 3.2. The numeric-type includes both 256-bit integer and Boolean values, and corresponds to both ⟨NExpr⟩ and ⟨BExpr⟩ in Fig. 3.2. The mapping-type contains multi-dimensional dictionaries, and corresponds to ⟨MapType⟩ in Fig. 3.3. Finally, the contract-type contains object pointers, and corresponds to ⟨CExpr⟩ in Fig. 3.2.

```
1  /// @title An open-bid auction.
2  contract Auction {
3    mapping(address => uint) bids;
4    address manager;
5    uint leadingBid;
6    bool stopped;
7
8    /// Initiates a new auction controlled by _manager.
9    constructor(address _manager) public {
10     require(manager != address(0));
11     manager = _manager;
12   }
13
14   /// Interprets each payment as a bid.
15   function () public payable {
16     bid();
17   }
18
19   /// Processes a valid bid.
20   /// A bid is valid if it:
21   /// 1. The auction is not stopped.
22   /// 2. The sender is not the manager.
23   /// 3. The sender is not the leading bidder.
24   /// 4. The bid exceeds the current leading bid.
25   function bid() public payable {
26     require(!stopped); // (1)
27     require(msg.sender != manager); // (2)
28
29     uint oldBid = bids[msg.sender];
30     uint newBid = oldBid + msg.value;
31     require(oldBid == 0 || oldBid < leadingBid); // (3)
32     require(newBid > leadingBid); // (4)
33
34     bids[msg.sender] = newBid;
35     leadingBid = newBid;
36   }
37
38   /// Computes the minimum bid a user must place.
39   function computeMinBid() public returns (uint) {
40     require(!stopped);
41     return leadingBid - bids[msg.sender];
42   }
43
44   /// Returns a non-leading bid.
45   function withdraw() public {
46     require(!stopped);
47     require(msg.sender != manager);
48
49     uint bid = bids[msg.sender];
50     require(bid < leadingBid);
51     bids[msg.sender] = 0;
52
53     msg.sender.transfer(bid);
54   }
55
56   /// Stops the auction.
57   function stop() public {
58     require(msg.sender == manager);
59     stopped = true;
60   }
61 }
62
63 /// @title An interface to create a timed auction.
64 contract TimedAuctionManager {
65   Auction auction;
66   uint end;
67
68   constructor(uint duration) public {
69     auction = new Auction(address(this));
70     end = block.number + duration;
71   }
72
73   /// Stops the auction if its duration has elapsed.
74   function tryStop() public {
75     if (end < block.number) {
76       auction.stop.value(0)();
77     }
78   }
79 }
```

Figure 3.1: An open-bid auction smart contract that demonstrates the main features of MiniSol. This smart contract extends on Fig. 1.1.

Each 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. 3.1, there is 1 role (`manager`), 2 contract data (`leadingBid` and `stopped`), 1 mapping (`bids`), 1 client common to all transactions (`msg.sender`), and 1 (used) argument common to every transaction (`msg.value`). In `TimedAuctionManager`, there is 1 contract-typed value (`auction`).

A smart contract definition (⟨Contract⟩ in Fig. 3.3) consists of one or more variable declarations (⟨Decl⟩ in Fig. 3.3) and one or more function declarations (⟨Func⟩ in Fig. 3.3). As in most object-oriented languages, each function has a *visibility modifier* (⟨Vis⟩ in Fig. 3.3). Functions with `public` visibility can be called by users (e.g., Line 25), whereas functions with `internal` visibility can only be executed as part of an ongoing transaction. In either case, the inputs `tx.origin` and `msg.sender` expose the addresses of the first and most recent caller during the current transaction.

⟨CExpr⟩ ::= x_c | `this`

⟨AExpr⟩ ::= x_a | *a literal address* |
`msg.sender` | `tx.origin` | `address(` ⟨CExpr⟩ `)` |

⟨Call⟩ ::= ⟨CExpr⟩`.`f`.value(` ⟨NExpr⟩ `)` `(` ⟨ExprList⟩ `)` | f `(` ⟨ExprList⟩ `)`

⟨Index⟩ ::= m `[` ⟨AExpr⟩ `]` ... `[` ⟨AExpr⟩ `]`

⟨NExpr⟩ ::= x_n | *a 256-bit (un)signed integer literal* | ⟨Index⟩ | ⟨Call⟩ |
`block.number` | `block.timestamp` | `msg.value` | ⟨CExpr⟩`.balance` |
⟨NExpr⟩ `+` ⟨NExpr⟩ | ⟨NExpr⟩ `-` ⟨NExpr⟩ | ⟨NExpr⟩ `*` ⟨NExpr⟩ | ⟨NExpr⟩ `/` ⟨NExpr⟩

⟨BExpr⟩ ::= x_b | `true` | `false` | ⟨Index⟩ | ⟨Call⟩ |
⟨BExpr⟩ `&&` ⟨BExpr⟩ | ⟨BExpr⟩ `||` ⟨BExpr⟩ | `!` ⟨BExpr⟩ |
⟨NExpr⟩ `<` ⟨NExpr⟩ | ⟨NExpr⟩ `>` ⟨NExpr⟩ | ⟨NExpr⟩ `==` ⟨NExpr⟩ | ⟨NExpr⟩ `!=` ⟨NExpr⟩ |
⟨AExpr⟩ `==` ⟨AExpr⟩ | ⟨AExpr⟩ `!=` ⟨AExpr⟩

⟨Expr⟩ ::= ⟨AExpr⟩ | ⟨NExpr⟩ | ⟨BExpr⟩

⟨ExprList⟩ ::= ⟨Expr⟩ ... ⟨Expr⟩

Figure 3.2: The formal grammar for MiniSol expressions. In this grammar, x_c, x_a, x_n, and x_b are contract-, address-, integer-, and Boolean-typed variables, respectively. Also, m and f are mappings and functions, respectively. For simplicity, mapping and function types are elided.

A function can also be marked as `payable` (e.g. Line 25). A payable function can receive Ether. The amount of Ether sent to a payable function is accessible through the `msg.value` argument (e.g., Line 30). There are two ways that one smart contract can send Ether to another smart contract. If a function is called by name, then the `.value` modifier is used (e.g., Line 76). Otherwise, a built-in `transfer` function (e.g., Line 53) can be used to call a special unnamed function, known as a *fallback function* (e.g., Line 15).

A constructor is a special function that is executed once after contract creation. Calls to `new` (i.e., creating new smart contracts) are restricted to loop-free code segments within constructors. `TimedAuctionManager` implements a constructor at Line 9 with a single call to `new`.

MiniSol also exposes a built-in time primitive to each function. In MiniSol, time is measured with respect to the number of blocks that have been executed. The `block.number` argument exposes the number of blocks executed so far (e.g, Line 70). The `block.timestamp` argument exposes the real-world time at which the last block was executed. MiniSol enforces that `block.timestamp` is non-decreasing.

⟨BasicType⟩ ::= `int` | `uint` | `bool`

⟨MapType⟩ ::= `mapping(` `address` `=>` ⟨MapType⟩ `)` | `mapping(` `address` `=>` ⟨BasicType⟩ `)`

⟨ComplexType⟩ ::= `address` | C | ⟨MapType⟩

⟨Decl⟩ ::= C x_c | `address` x_a | `int` x_n | `uint` x_n | `bool` x_b | ⟨MapType⟩ m

⟨Assign⟩ ::= x_a `=` ⟨AExpr⟩ | x_n `=` ⟨NExpr⟩ | x_b `=` ⟨BExpr⟩ | x_c `=` `new` C `(` ⟨ExprList⟩ `)` |
⟨Index⟩ `=` ⟨NExpr⟩ | ⟨Index⟩ `=` ⟨BExpr⟩

⟨Stmt⟩ ::= ⟨Decl⟩ | ⟨Assign⟩ | ⟨Stmt⟩`;` ⟨Stmt⟩ |
`require(` ⟨BExpr⟩ `)` | `assert(` ⟨BExpr⟩ `)` | `return` | `return` ⟨Expr⟩ |
`if` `(` ⟨BExpr⟩ `)` `{` ⟨Stmt⟩ `}` `else` `{` ⟨Stmt⟩ `}` | `while(` ⟨Expr⟩ `)` `{` ⟨Stmt⟩ `}` |
⟨AExpr⟩`.transfer(` ⟨NExpr⟩ `)`

⟨Payable⟩ ::= `payable` | ϵ

⟨Vis⟩ ::= `public` | `internal`

⟨RetVal⟩ ::= `returns` `(` ⟨BasicType⟩ `)` | ϵ

⟨DeclList⟩ ::= ⟨Decl⟩`,` ... `,` ⟨Decl⟩ | ϵ

⟨Ctor⟩ ::= `constructor` `(` ⟨DeclList⟩ `)` `public` ⟨Payable⟩ `{` ⟨Stmt⟩ `}`

⟨Func⟩ ::= `function` f `(` ⟨DeclList⟩ `)` ⟨Vis⟩ ⟨Payable⟩ ⟨RetVal⟩ `{` ⟨Stmt⟩ `}`

⟨Contract⟩ ::= `contract` C `{` ⟨Decl⟩`;` ... ⟨Decl⟩`;` ⟨Ctor⟩ ⟨Func⟩ ... ⟨Func⟩ `}`

⟨Program⟩ ::= ⟨Contract⟩ ... ⟨Contract⟩

Figure 3.3: The formal grammar for MiniSol statements, methods, and contracts. Expressions are defined as in Fig. 3.2. In this grammar, C is the name of a contract.

3.1.2 Semantics

Let $\mathcal{C}$ be a MiniSol program with methods, $\mathbb{F} := \{fn_1, \ldots, fn_k\}$. Assume that each mapping in $\mathcal{C}$ is 1-dimensional (multi-dimensional mappings are discussed in Appendix A). An N-user *bundle* is an N-user network of several (possibly identical) MiniSol smart contracts. The semantics of a bundle is an LTS, $\mathsf{global}(\mathcal{C}, N) := (S, P, f, s_0)$, such that:

1. $S_C := \mathsf{control}(\mathcal{C}, [N])$ is the set of control states;
2. $S_U := \mathsf{user}(\mathcal{C}, [N])$, is the set of user states;
3. $s_\perp$ is the error state;
4. $S := (S_C \cup \{s_\perp\}) \times (S_U)^N$ is the set of global states;
5. $P := \mathsf{inputs}(\mathcal{C}, [N])$ is the set of actions;
6. $f : S \times P \to S$ is the *transition function*;

7. $s_0 \in S$ is the initial state.

We assume, without loss of generality, that there is a single control process[1].

Let $\mathbb{D}$ be the set of 256-bit integer values. The states are determined by the address space, $\mathcal{A}$, and the state variables of $\mathcal{C}$. Assume that m, n, 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, $\mathsf{control}(\mathcal{C}, \mathcal{A}) \subseteq \left(\mathcal{A}^m \times \mathbb{D}^{(n+3)}\right)$ and $\mathsf{user}(\mathcal{C}, \mathcal{A}) \subseteq \left(\mathcal{A} \times \mathbb{D}^k\right)$. For $c = (\mathbf{x}, \mathbf{y}) \in \mathsf{control}(\mathcal{C}, \mathcal{A})$:

1. $\mathsf{role}(c, i) = \mathbf{x}_i$ is the i-th role;
2. $\mathsf{data}(c, 0) = \mathbf{y}_0$ is `block.number`;
3. $\mathsf{data}(c, 1) = \mathbf{y}_1$ is `block.timestamp`;
4. $\mathsf{data}(c, 2) = \mathbf{y}_2$ is the smart contract balance;
5. $\mathsf{data}(c, i + 3) = \mathbf{y}_{i+3}$ is the i-th datum.

For $u = (a, \mathbf{x}) \in \mathsf{user}(\mathcal{C}, \mathcal{A})$:

1. $\mathsf{id}(u) = a$ is the address of user u;
2. $\mathsf{map}(u)_i = \mathbf{x}_i$ is the i-th mapping value for user u.

The actions are determined by the address space, $\mathcal{A}$, and the input variables for each method in $\mathbb{F}$. Assume that q and r are the maximum number of clients and arguments of any method in $\mathbb{F}$. Then $\mathsf{inputs}(\mathcal{C}, \mathcal{A}) \subseteq \left(\mathbb{F} \times \mathcal{A}^{(q+1)} \times \mathbb{D}^{(r+3)}\right)$. For $p = (fn, \mathbf{x}, \mathbf{y}) \in \mathsf{inputs}(\mathcal{C}, \mathcal{A})$:

1. fn is the first method invoked in the transaction;
2. $\mathsf{client}(p, 0) = \mathbf{x}_0$ is `tx.origin` and the initial `msg.sender`;
3. $\mathsf{client}(p, i + 1) = \mathbf{x}_{i+2}$ is the i-th client in fn;
4. $\mathsf{arg}(p, 0) = \mathbf{y}_0$ is `msg.value`;
5. $\mathsf{arg}(p, 1) = \mathbf{y}_1$ is the next value of `block.number`;

[1] The number of users in a MiniSol bundle is a static fact. Therefore, all control states are synchronized, and can be combined into a product machine.

6. $\mathsf{arg}(p, 2) = \mathbf{y}_2$ is the next value of `block.timestamp`;

7. $\mathsf{arg}(p, i + 3) = \mathbf{y}_{i+3}$ is the i-th argument in fn.

For a fixed p, we write $f_p(s, \mathbf{u})$ to denote $f((s, \mathbf{u}), p)$.

The initial state of $\mathsf{global}(\mathcal{C}, N)$ is $s_0 := (c, \mathbf{u}) \in S_C \times (S_U)^N$, such that $c = (\mathbf{0}, \mathbf{0})$, $\forall i \in [N] \cdot \mathsf{map}(\mathbf{u}_i) = \mathbf{0}$, 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 for $fn \in \mathbb{F}$ is determined by the (usual) semantics of fn (e.g. [4, 26, 38]), and a bijection from addresses to user indices, $\mathcal{M} : \mathcal{A} \to [\,|\mathcal{A}|\,]$. If $\mathcal{M}(a) = i$, then address a belongs to user $\mathbf{u}_i$. In the case of $\mathsf{global}(\mathcal{C}, N)$, the i-th user has address i, so $\mathcal{M}(i) = i$. We write $f := [\![\mathcal{C}]\!]_{\mathcal{M}}$, and given an action $p = (fn, \mathbf{x}, \mathbf{y})$, f_p updates the state variables according to the source code of fn with respect to $\mathcal{M}$. If a call originates from the zero-account or a smart contract, then the state is unchanged. 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. At the end of each transaction, $[\![\mathcal{C}]\!]_{\mathcal{M}}$ may increment `block.number` and `block.timestamp` in lockstep. The change in time is determinstic, and determined by $\mathsf{arg}(p, 1)$ and $\mathsf{arg}(p, 2)$. Note that $[\![\mathcal{C}]\!]_{\mathcal{M}}$ preserves the address of each user.

A complete LTS for Fig. 3.1 is given in Appendix B.

Remark 3.1.1. In Sections 5.3 and 7.1, it is helpful to give semantics to a smart contract with an arbitrary address space $\{0\} \subseteq \mathcal{A} \subseteq \mathbb{N}$. Let $N = |\mathcal{A}|$ and a_i be the i-th largest element in $\mathcal{A}$. In these cases, $S_C = \mathsf{control}(\mathcal{C}, \mathcal{A})$, $S_U = \mathsf{user}(\mathcal{C}, \mathcal{A})$, $P = \mathsf{inputs}(\mathcal{C}, \mathcal{A})$, and $S = (S_C \cup \{s_\perp\}) \times (S_U)^N$. Furthermore, in the initial state $s_0 = (c, \mathbf{u})$, $\forall i \in [N] \cdot \mathsf{id}(\mathbf{u}_i) = a_i$, and consequently, in the transition function, $\mathcal{M}(a_i) = i$. The LTS is denoted $\mathsf{global}^\circ(\mathcal{C}, \mathcal{A})$ and $\mathsf{global}(\mathcal{C}, N) = \mathsf{global}^\circ(\mathcal{C}, [N])$.

3.1.3 Limitations

MiniSol places three key restrictions on Solidity. First, addresses are nominal and do not permit arithmetic operations. This means that addresses may only be compared by equality and disequality. We argue that this restriction is reasonable, as address manipulation is a form of pointer manipulation. Second, MiniSol does not model gas. However, gas is used in Solidity to restrict the set of feasible transactions. Therefore, MiniSol strictly over-approximates the executions of Solidity. Third, `new` must only appear in loop-free constructor code. In effect, this restriction states that smart contracts are never created nor destroyed. This third assumption is common is smart contract analysis (e.g. [30, 37, 58]).

MiniSol is also insufficient for modeling reentrancy attacks. This is because MiniSol does not model calls from smart contracts outside of the bundle. However, reentrancy attacks are a well studied problem and can be avoided by writing EECF smart contracts. We follow with prior work (e.g. [58]), and assume that all smart contracts are EECF.

3.2 The MiniSol Specification Language

This section presents the MSL. First, a syntax is presented for aggregate functions, that generalizes sum(·), count(·), min(·), and max(·). A parameterized, temporal specification language, called MSL, is then defined using this class of aggregates. Finally, it is shown that MSL can be used to express many interesting smart contract specifications.

3.2.1 Aggregate Functions

An aggregate function takes as input a list of values, and gives as output a single value. For example, count(·) is applied to a list of Boolean values, and returns a single integer value. If S_1 is the input domain, and S_2 is the output domain, then each aggregate is characterized by some family of functions, $\left\{ f_n : (S_1)^N \to S_2 \mid N \in \mathbb{N} \right\}$.

However, most practical aggregates have far more structure. In the case of count($\mathbf{x}$), the elements of $\mathbf{x}$ are mapped to either 0 or 1, and then the new elements are combined using the + operator. Similarly, max($\mathbf{x}$) maps each element of $\mathbf{x}$ into $\mathbb{N}$, and then combines the new elements by taking their supremum. In either case, the aggregate first applies a function $g : S_1 \to S_2$ to each element of the list, and the folds the elements together using a function $f : S_2 \times S_2 \to S_2$. This intuition is formalized in Definition 3.2.1.

Definition 3.2.1 (Point-wise Aggregate)**.** Let $g : S_1 \to S_2$, $h : S_2 \times S_2 \to S_2$, and $z \in S_2$. The *point-wise aggregate of* (g, h, z) is the function $f(\mathbf{x}) := \mathsf{fold}(h, z, g(\mathbf{x}))$. As a shorthand, $\mathsf{agg}(g, h, z, \mathbf{x}) := f(\mathbf{x})$.

An important property of many aggregate functions is that the function is *order-insensitive*. That is, $f(\mathbf{x})$ does not change if the elements of $\mathbf{x}$ are reordered. For example, sum($\mathbf{x}$) is order-insensitive since addition is commutative. Similarly, max($\mathbf{x}$) is order-insensitive since max(·) finds the supremum of the set of elements in $\mathbf{x}$. These properties generalize to any aggregate constructed from an Abelian group (Theorem 1) or a lattice (Theorem 2).

Theorem 1. Let $g : S_1 \to S_2$, $h : S_2 \times S_2 \to S_2$, $z \in S_2$, and f be the point-wise aggregate of (g, h, z). If (S_2, h) is an Abelian group, then f is order-insensitive.

Proof. Assume that (S_2, h) is an Abelian group. For simplicity of presentation, let $x \circ y = f(x, y)$. By Proposition 1, if f preserved under transpositions, then f is also preserved under list permutations. Fix some $N \in \mathbb{N}$, $\mathbf{u} \in (S_2)^N$, $i \in [N-1]$, and let $\mathbf{x} = g(\mathbf{u})$.

$$f(\tau_i(\mathbf{u})) = (((((z \circ \mathbf{x}_0) \circ \mathbf{x}_1) \circ \cdots \circ \mathbf{x}_i) \circ \mathbf{x}_{i-1}) \circ \cdots \circ \mathbf{x}_N)$$
$$f(\tau_i(\mathbf{u})) = (((((z \circ \mathbf{x}_0) \circ \mathbf{x}_1) \circ \cdots \circ \mathbf{x}_{i-1}) \circ \mathbf{x}_i) \circ \cdots \circ \mathbf{x}_N)$$
$$f(\tau_i(\mathbf{u})) = f(\mathbf{x})$$

Since (S_2, h) is Abelian, then $\circ$ is commutative, and therefore, $f(\tau_i(\mathbf{u})) = f(\mathbf{u})$. Consequently, f is order-insensitive. □

Theorem 2. Let $g : S_1 \to S_2$, $h : S_2 \times S_2 \to S_2$, $z \in S_2$, and f be the point-wise aggregate of (g, h, z). If $(S_2, \leq)$ is a lattice and h defines the supremum operator for S_2, then f is an order-insensitive.

Proof. Assume that $(S_2, \leq)$ is a lattice, and that h defines the supremum of S_2. For simplicity of presentation, let $\sup\{x, y\} = h(x, y)$. Fix some $N \in \mathbb{N}$, $\mathbf{u} \in (S_2)^N$, and let $\mathbf{x} = g(\mathbf{u})$. It follows by induction that $f(\mathbf{u}) = \sup\{z, \mathbf{x}_0, \ldots, \mathbf{x}_{N-1}\}$. In the base case, $N = 1$ and $f(\mathbf{u}) = \{z, \mathbf{x}_0\}$. In the inductive case, assume that $\mathbf{u}'$ is the length $N-1$ prefix of $\mathbf{u}$, and that $f(\mathbf{u}') = \sup\{z, \mathbf{x}_0, \ldots, \mathbf{x}_{N-2}\}$. Then $f(\mathbf{u}) = \sup\{\mathbf{x}_{N-1}, f(\mathbf{x}')\} = \sup\{z, \mathbf{x}_0, \ldots, \mathbf{x}_{N-1}\}$. Therefore, the inductive case holds. As sets are unordered, then f is order-insensitive. □

An important property of $\text{sum}(\mathbf{x})$ is that it can be updated in $O(1)$ time. Specifically, $\text{sum}(\mathbf{x}[i \leftarrow n]) = \text{sum}(\mathbf{x}) - \mathbf{x}_i + n$. This relation is well known, and utilized by many static analysis tools for Solidity (e.g. [30, 58]). A similar, but weaker, relationship holds for $\max(\mathbf{x})$ as well. For any choice of n, $\max(\mathbf{x}[i \leftarrow n]) \leq \max\{\max(\mathbf{x}), n\}$. When $\mathbf{x}_i \leq n$, the equality is strict. As with order-insensitivity, these properties also generalize to aggregates constructed from Abelian groups (Theorem 3), or lattices (Theorem 4).

Theorem 3. Let $g : S_1 \to S_2$, $h : S_2 \times S_2 \to S_2$, $z \in S_2$, and f be the point-wise aggregate of (g, h, z). If (S_2, h) is an Abelian group, then $f(\mathbf{x}[i \leftarrow n]) = h\left(h^{-1}(f(\mathbf{x}), g(\mathbf{x}_i)), g(n)\right)$.

Proof. Assume that (S_2, h) is an Abelian group. By Theorem 1, it can be assumed that $i = |\mathbf{x}| - 1$. Let $\mathbf{x}'$ be the length $|\mathbf{x}| - 1$ prefix of $\mathbf{x}$. By definition, $f(\mathbf{x}') = h^{-1}(f(\mathbf{x}), g(\mathbf{x}_i))$ and $f(\mathbf{x}[i \leftarrow n]) = h(f(\mathbf{x}'), g(n))$. Therefore, $f(\mathbf{x}[i \leftarrow n]) = h\left(h^{-1}(f(\mathbf{x}), g(\mathbf{x}_i)), g(n)\right)$. □

Aggregate	$g(u)$	$h(x, y)$	z	Structure of (S_2, f)
Sum of all bids	$\mathsf{map}(u)_0$	$x + y$	0	Abelian Group
Count of active bids	$\mathsf{ite}(\mathsf{map}(u)_0 = 0, 0, 1)$	$x + y$	0	Abelian Group
Maximum bid	$\mathsf{map}(u)_0$	$\sup\{x, y\}$	0	Complete Lattice
Minimum bid	$\mathsf{map}(u)_0$	$\inf\{x, y\}$	0	Complete Lattice

Table 3.1: Example point-wise aggregates for `bids` in Fig. 3.1.

Theorem 4. Let $g : S_1 \to S_2$, $h : S_2 \times S_2 \to S_2$, $z \in S_2$, and f be the point-wise aggregate of (g, h, z). If $(S_2, \leq)$ is a lattice and h defines the supremum operator for S_2, then $f(\mathbf{x}[i \leftarrow n]) \leq h(f(\mathbf{x}), g(n))$. Furthermore, if $g(\mathbf{x}_i) \leq g(n)$, the the equality is strict.

Proof. Assume that $(S_2, \leq)$ is lattice and that h defines the supremum of S_2. For simplicity of presentation, let $\sup\{x, y\} = h(x, y)$. By Theorem 2, it can be assumed that $i = |\mathbf{x}| - 1$. Let $v = \sup\{z, g(\mathbf{x}_0), \ldots, g(\mathbf{x}_i)\}$ and $v' = \sup\{z, g(\mathbf{x}_0), \cdots, g(\mathbf{x}_{i-1}), g(n)\}$. If $g(\mathbf{x}_i) \leq g(n)$, then $v' = \sup\{v, g(n)\}$. Otherwise, $\sup\{g(\mathbf{x}_i), g(n)\} > g(n)$, and consequently, $v' \leq \sup\{v, g(n)\}$. □

In MSL, aggregates are limited to user data. This means that for any user u, g maps $\mathsf{map}(u)$ to some domain of scalars. Often, properties are concerned with the aggregates of a single mapping, for example, `bids` in Fig. 3.1. The sum of all bids in `Auction` is computed by setting $g(u) = \mathsf{map}(u)_0$ and $f(x, y) = x + y$. More examples are given in Table 3.1.

3.2.2 Guarded k-Universal Safety Properties

Recall that users do not have addresses in a traditional SCUN. This means that k-universal properties are oblivious to user addresses. Formally, a k-universal property for an N-user MiniSol bundle would have the form $\forall\{i_1, \ldots, i_k\} \in [N] \cdot \varphi\left(c, \mathsf{map}\left(\mathbf{u}_{i_1}\right), \ldots, \mathsf{map}\left(\mathbf{u}_{i_k}\right)\right)$. In this section, a syntax is presented for k-universal safety properties in MSL. The syntax is then extended to addresses through the introduction of *guarded k-universal safety properties*. Finally, temporal operators are introduced to capture the temporal nature of MiniSol programs.

A k-universal property is a Boolean combination of atomic predicates (see Fig. 3.4). Each atomic predicate is either a Boolean variable in the smart contract, an (in)equality between two numerically-typed expressions, or an application of $\mathsf{called}(\cdot)$. The $\mathsf{called}(\cdot)$

$$
\begin{array}{rcl}
\langle \text{Term} \rangle & ::= & \mathsf{agg}(z, g, h, \mathbf{u}) \mid \mathsf{map}(\mathbf{u}_{i_a})_b \mid \mathsf{data}(c, b) \mid n \mid \langle \text{Term} \rangle \circ \langle \text{Term} \rangle \\
\langle \text{Atom} \rangle & ::= & \langle \text{Term} \rangle \mid \langle \text{Term} \rangle \bowtie \langle \text{Term} \rangle \mid \mathsf{called}(fn) \\
\langle \text{Prop} \rangle & ::= & \langle \text{Atom} \rangle \mid \langle \text{Prop} \rangle \star \langle \text{Prop} \rangle
\end{array}
$$

Figure 3.4: The syntax of k-universal properties in MSL. In this grammar $a \in [k]$, $b \in \mathbb{N}$, $n \in \mathbb{D}$, $\circ$ is an arithmetic operation, $\bowtie$ is an (in)equality, and $\star$ is a Boolean operation.

operator is applied to the name of a method, and is true if the given method is the last invoked method. For simplicity, variable names are used in place of semantic expressions when the meaning is unambiguous (i.e., writing `bids[i]` for $\mathsf{map}(\mathbf{u}_i)_0$).

Next, k-universal properties are extended with guards. In MSL, there are two sorts of guards: literal guards and role guards. A *literal guard*, written $(i, a) \in [k] \times \mathbb{N}$, says that the i-th user has address a (i.e., $\mathsf{id}(\mathbf{u})_i = a$). A *role guard*, written $(i, j) \in [k] \times \mathbb{N}$, says that the i-th user is assigned to the j-th role (i.e., $\mathsf{id}(\mathbf{u})_i = \mathsf{role}(c, j)$). The *guarded k-universal safety property* is formalized in Definition 3.2.2, and illustrated in Example 3.2.1.

Definition 3.2.2 (Guarded Universal Safety)**.** For $k \in \mathbb{N}$, a *guarded k-universal safety property* is the parameterized property φ, given by a tuple, (L, R, ξ), where $L \subseteq [k] \times \mathbb{N}$ is finite, $R \subseteq [k] \times \mathbb{N}$ is finite, ξ is a k-universal safety property, such that:

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

Example 3.2.1. Consider the claim that in `Auction` of Fig. 3.1, the zero-account and manager have bids less than or equal to all other users. This claim is stated by **Prop. 1**: *For any three unique user processes, u, v, and w, if $\mathsf{id}(u) = 0$ and $\mathsf{id}(v) = \mathsf{role}(c, 0)$, then $\mathsf{map}(u)_0 \leq \mathsf{map}(w)_0$ and $\mathsf{map}(v)_0 \leq \mathsf{map}(w)_0$.* Then **Prop. 1** is the guarded 3-universal safety property φ_1 defined to be:

$$
(\mathsf{id}(\mathbf{u}_{i_0}) = 0 \ \wedge \ \mathsf{id}(\mathbf{u}_{i_1}) = \mathsf{role}(c, 0)) \Rightarrow (\mathsf{map}(\mathbf{u}_{i_0})_0 \leq \mathsf{map}(\mathbf{u}_{i_2})_0 \ \wedge \ \mathsf{map}(\mathbf{u}_{i_1})_0 \leq \mathsf{map}(\mathbf{u}_{i_2})_0)
$$

Following Definition 3.2.2, φ_1 is determined by the tuple (L_1, R_1, ξ_1), where $L_1 = \{(0, 0)\}$, $R_1 = \{(1, 0)\}$, and $\xi_1 := (\mathsf{map}(\mathbf{u}_{i_0})_0 \leq \mathsf{map}(\mathbf{u}_{i_2})_0 \ \wedge \ \mathsf{map}(\mathbf{u}_{i_1})_0 \leq \mathsf{map}(\mathbf{u}_{i_2})_0)$. If a state $(c, \mathbf{u})$ satisfies φ_1, then $(c, \mathbf{u}) \models \forall \{i_1, i_2, i_3\} \in [\,|\mathbf{u}|\,] \cdot \varphi_1$. □

The full syntax for MSL allows for pure-past expressions of the form $\mathsf{always}(\varphi)$, where φ is a pure-past expression over guarded k-universal safety properties. Recall from Section 2.5

that all pure-past properties are reducible to safety Therefore, it is assumed in later proofs that all properties are guarded k-universal safety properties. In the style of [58], we abuse notation and write $\mathsf{prev}(\mathtt{x})$ to denote the previous value of `x`.

3.2.3 Example Properties

Additional MSL properties for `Auction` of Fig. 3.1 include:

Prop. 2 : *If two users have active bids, then their bids are not equal.*

Prop. 3 : *Once stopped is called, all bids are immutable.*

Prop. 4 : *The sum of all bids is greater than or equal to the leading bid.*

Prop. 5 : *If the auction has not stopped, then the leading bid is the maximum bid.*

Prop. 6 : *The manager cannot bid.*

The properties are formalized by $\mathsf{always}(\varphi_2)$ through to $\mathsf{always}(\varphi_6)$, respectively:

$$\varphi_2 := \mathsf{historically}\,(\forall\{i,j\} \in [N] \cdot (\mathtt{bids[i]} = 0) \ \vee \ (\mathtt{bids[i]} \neq \mathtt{bids[j]}))$$
$$\varphi_3 := \mathsf{historically}\,(\mathsf{once}(\mathsf{called}(\mathtt{stopped})) \Rightarrow (\forall\{i\} \in [N] \cdot \mathsf{prev}(\mathtt{bids[i]}) = \mathtt{bids[i]}))$$
$$\varphi_4 := \mathsf{historically}\,(\mathrm{sum}(\mathtt{bids}) \geq \mathtt{leadingBid})$$
$$\varphi_5 := \mathsf{historically}\,(\neg\mathsf{once}(\mathsf{called}(\mathtt{stopped})) \Rightarrow (\max(\mathtt{bids}) = \mathtt{leadingBid}))$$
$$\varphi_6 := \mathsf{historically}\,(\forall\{i\} \in [N] \cdot (\mathsf{role}(c,0) = i) \Rightarrow (\mathtt{bids[i]} = 0))$$

Chapter 4

Communication and its Abstractions

The core functionality of any smart contract is communication between users. Usually, user communicate indirectly by reading from, and writing, to designated mapping entries. That is, the communication paradigm is shared memory. However, the indirection of shared memory obfuscates the interactions between users. Instead, it is convenient to re-imagine smart contracts as having rendezvous synchronization. That is, a group of one or more users first meet at the start of a transaction (the barrier), and then participate in message passing until the transaction has terminated.

In this chapter, MiniSol smart contracts are formally re-framed with explicit communication. First, influence is introduced to give explicit semantics to the shared memory operations. A participation topology is defined from influence to uncover the users that synchronize at each transaction. Finally, the participation topology graph is define to offer finite abstractions for each participation topology.

4.1 Communication as a Participation Topology

A user u *influences* a transaction f_p whenever the state of u affects an execution of f_p. For example, at Line 47 of Fig. 3.1, the sender influences `withdraw` by having its address compared to `manager`. There are two ways that a user's state can affect the execution of f_p. First, f_p can compare the address of u to the address of some other user. Second, f_p can access the data of u, and then use this data to determine the outcome of the transaction. Intuitively, user influence provides explicit semantics to each shared memory read. User influence is formalized by Definition 4.1.1. It requires the existence of two

network configurations that differ only in the state of u, and result in different network configurations after applying f_p.

Definition 4.1.1 (User Influence)**.** Let $\mathcal{C}$ be a bundle, $N \in \mathbb{N}$, $(S, P, f, s_0) = \mathsf{global}(\mathcal{C}, N)$, and $p \in P$. The user with address $a \in \mathbb{N}$ *influences* transaction f_p if there exists an $c, r, r' \in \mathsf{control}(\mathcal{C}, [N])$, $\mathbf{u}, \mathbf{u}', \mathbf{v}, \mathbf{v}' \in \mathsf{user}(\mathcal{C}, [N])^N$, 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. $(r, \mathbf{u}') = f_p(c, \mathbf{u})$ and $(r', \mathbf{v}') = f_p(c, \mathbf{v})$;
4. $(r = r') \Rightarrow \big(\exists j \in [N] \setminus \{i\} \cdot \mathbf{u}'_j \neq \mathbf{v}'_j\big)$.

The tuple $(c, \mathbf{u}, \mathbf{v})$ is a *witness to the influence of user a over transaction f_p.*

A user u is *influenced* by a transaction f_p whenever f_p affects the state of u. For example, at Line 51 of Fig. 3.1, `withdraw` influences the sender by overwriting the sender's bid. It is not hard to see that transaction influence provides explicit semantics to each shared memory write.

Definition 4.1.2 (Transaction Influence)**.** Let $\mathcal{C}$ be a bundle, $N \in \mathbb{N}$, $(S, P, f, s_0) = \mathsf{global}(\mathcal{C}, N)$, and $p \in P$. The user with address $a \in \mathbb{N}$ is *influenced by* transaction f_p if there exists an $c, c' \in \mathsf{control}(\mathcal{C}, [N])$, $\mathbf{u}, \mathbf{u}' \in \mathsf{user}(\mathcal{C}, [N])^N$, and $i \in [N]$ such that $(c', \mathbf{u}') = f_p(c, \mathbf{u})$, $\mathsf{id}(\mathbf{u}_i) = a$, and $\mathbf{u}'_i \neq \mathbf{u}_i$. The tuple $(c, \mathbf{u})$ is a *witness to the influence of transaction f_p over user a.*

A user u *participates* in transaction f_p if either u influences f_p or u is influenced by f_p. In other words, f_p reads from, or writes to, the shared memory of u, and after the transaction terminates, these changes persist.

Definition 4.1.3 (Participation)**.** The user with address $a \in \mathbb{N}$ *participates* in the transaction f_p if either, a influences f_p with some witness $(c, \mathbf{u}, \mathbf{v})$, or a is influenced by f_p with some witness state $(c, \mathbf{u})$. In either case, c is a *witness state.*

In general, smart contracts facilitate communication between many users across many transactions. Given any transaction, communication analysis requires knowledge of all possible participants, and the root cause of their participation. Such a summary is called a 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. Intuitively, explicit participation occurs when a user enters into a transaction explicitly, by virtue of being a client. Conversely, implicit participation occurs when a user enters into a transaction without its address appearing in either the action, or the control state. An example of implicit participation occurs at line Line 10 of Fig. 3.1, when `Auction` compares the manager's address to the *constant* address of the zero-account. Finally, transient participation starts out as either implicit or explicit participation, but then persists across multiple transactions by assignment of a role to the user.

Definition 4.1.4 (Participation Topology)**.** Let $\mathcal{C}$ be a bundle with m roles, $N \in \mathbb{N}$, $(S, P, f, s_0) = \mathsf{global}(\mathcal{C}, N)$, and $p \in P$ be an action with q clients. The *Participation Topology* of a transaction f_p is the tuple $\mathsf{pt}(\mathcal{C}, N, p) := (\text{Explicit}, \text{Transient}, \text{Implicit})$, such that:

1. $\text{Explicit} \subseteq [q] \times [N]$ such that $(i, a) \in \text{Explicit}$ if and only if a participates during f_p and $\mathsf{client}(p, i) = a$;
2. $\text{Transient} \subseteq [m] \times [N]$ such that $(i, a) \in \text{Transient}$ if and only if a participates during f_p, as witnessed by some state $c \in \mathsf{control}(\mathcal{C}, [N])$ satisfying $\mathsf{role}(c, i) = a$;
3. $\text{Implicit} \subseteq [N]$ such that $a \in \text{Implicit}$ if and only if a participates during f_p, as witnessed by some state $c \in \mathsf{control}(\mathcal{C}, [N])$ satisfying $\forall i \in [m] \cdot \mathsf{role}(c, i) \neq a$ and $\forall i \in [q] \cdot \mathsf{client}(p, i) \neq a$.

To illustrate Definition 4.1.4, a PT is constructed for Fig. 3.1. Let $\mathcal{C} = (\texttt{Auction})$, 4 be the network size, and assume for simplicity that the address of `Auction` is 1. Consider the action $p = (\texttt{bid}, (3), (10)) \in \mathsf{inputs}(\mathcal{C}, [4])$, that states that the user at address 3 invokes `bid` with a message value of 10. The first client, at address 3, is an explicit participant since `bid` updates the balance of the sender. The users at addresses 0 and 1 are implicit participant, since neither the the zero-account nor the `Auction` can invoke a transaction. Consequently, remapping the sender to 0 or 1 would force the transaction to revert (this is expanded on in the proof of Theorem 6). The manager is a transient participant, since the sender's is compared to the manager's address. If the manager is not equal to the sender, then remapping the manager to the address of the sender would force the transaction to revert. Then the full PT is $(\text{Explicit}, \text{Transient}, \text{Implicit}) = \mathsf{pt}(\mathcal{C}, 3, p)$, such that: $\text{Explicit} = \{(0, 3)\}$, $\text{Transient} = \{(0, 0), (0, 1), (0, 2), (0, 3)\}$, and $\text{Implicit} = \{0, 1\}$. It

is important to note that the user at address $a \in [4]$ appears in $\mathsf{pt}(\mathcal{C}, 3, p)$ if and only if a could participate in p. This generalizes to every PT, as stated by Theorem 5.

Theorem 5. Let $\mathcal{C}$ be a bundle, $N \in \mathbb{N}$, $(S, P, f, s_0) = \mathsf{global}(\mathcal{C}, N)$, $p \in P$, and $\mathsf{pt}(\mathcal{C}, N, p) = (\text{Explicit}, \text{Transient}, \text{Implicit})$. The user $a \in \mathbb{N}$ participates in f_p if and only if $(\exists i \in \mathbb{N} \cdot (i, a) \in \text{Explicit}) \vee (\exists i \in \mathbb{N} \cdot (i, a) \in \text{Transient}) \vee (a \in \text{Implicit})$.

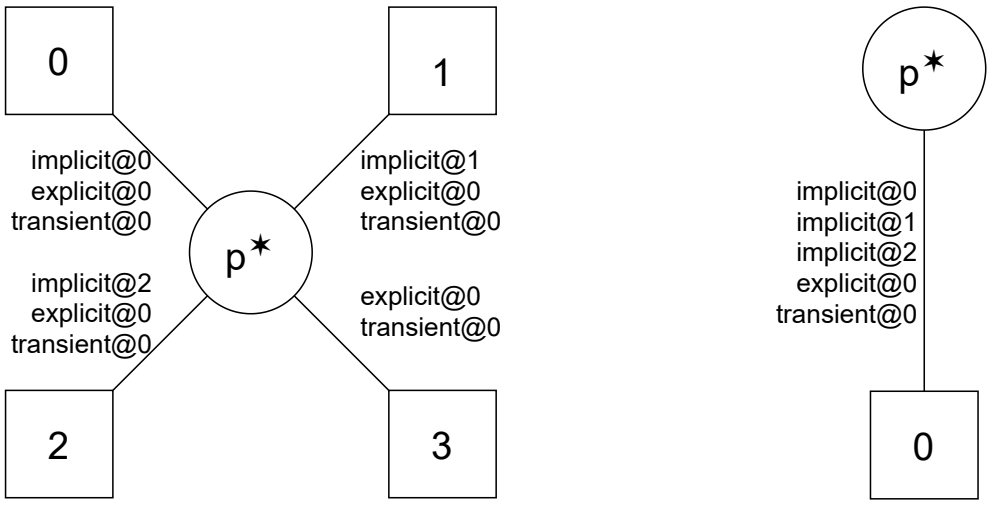
Proof. The ($\Leftarrow$) direction follows directly from Definition 4.1.4. Consider the ($\Rightarrow$) direction. Assume for the intent of contradiction that a participates in f_p and a does not appear in $\mathsf{pt}(\mathcal{C}, N, p)$. If a participates in f_p, then by Definition 4.1.3 there is at least one witness state, $c \in \mathsf{control}(\mathcal{C}, [N])$. By assumption, $a \notin$ Implicit. Then $\exists i \in \mathbb{N} \cdot \mathsf{role}(c, i) = a$ or $\exists i \in \mathbb{N} \cdot \mathsf{client}(p, i) = a$. In the first case, $(i, a) \in$ Explicit, and in the second case, $(i, a) \in$ Transient. By contradiction, a appears in $\mathsf{pt}(\mathcal{C}, N, p)$. □

4.2 Abstraction via Participation Topology Graphs

Definition 4.1.4 is semantic and depends on both the choice of action, and the size of the network. The set of all network sizes is countably infinite, and therefore, the set of all PTs is also infinite. However, communication analysis requires reasoning about all possible PTs. This motivates the PTG that is a syntactic over-approximation of all PTs, independent of network size. A PTG has a vertex for each user and each action. In general, a single vertex can map to many user or many actions. An edge exists between a user and an action if the user can participate in the corresponding transaction. Furthermore, each edge is labelled by all corresponding participation classes. This intuition between a PTG is inspired by "control-flow graphs" that map program locations to vertices, and then over-approximate program reachability via graph reachability [3].

PTG edges are labelled by participation class. For any bundle $\mathcal{C}$, there are at most q explicit classes and m transient classes, where m is the number of roles, and q 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 Definition 4.1.4). Intuitively, the implicit participation class exploit domain-specific knowledge to classify each implicit participant. The label set common to all PTGs is $\mathsf{PC}(\mathcal{C}) := \{\mathit{explicit}@i \mid i \in [q]\} \cup \{\mathit{transient}@i \mid i \in [m]\}$.

Definition 4.2.1 (Participation Topology Graph)**.** Let $\mathcal{C}$ be a bundle, L be a finite set of implicit classes, and $P_{\mathbb{N}} = \mathsf{inputs}(\mathcal{C}, \mathbb{N})$. A *Participation Topology Graph* for $\mathcal{C}$ is a



(a) The PTG produced by `PTGBuilder`.

(b) The weakest possible PTG.

Figure 4.1: Two PTGs for `TimedAuctionManager` in Fig. 3.1. In each diagram, $p^\star$ is an arbitrary action.

tuple $((N_A \cup N_P, E, \delta), \mu, \eta)$ where $(N_A, \cup N_P, E, \delta)$ is a labelled bipartite graph, $N_A \subseteq \mathbb{N}$, $N_P \subseteq P_{\mathbb{N}}$, $\delta \subseteq E \times (\mathsf{PC}(\mathcal{C}) \cup L)$, $\mu : P_{\mathbb{N}} \to N_P$, $\eta : N_P \times \mathbb{N} \to N_A$, such that for all $N \in \mathbb{N}$, and for all $p \in \mathsf{inputs}(\mathcal{C}, [N])$, with $\mathsf{pt}(\mathcal{C}, N, p) = (\text{Explicit}, \text{Transient}, \text{Implicit})$:

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

In Definition 4.2.1, μ and η map actions and users to vertices, respectively. An edge between $\mu(p)$ and $\eta(\mu(p), a)$ indicates the potential for a to participate in f_p. Each label on the edge states a potential participation class for a in f_p. As an example, Fig. 4.1a is a PTG for Fig. 3.1, such that all actions map to a single representative action, the zero-account maps to 0, the address of `TimedAuctionManager` maps to 1, the address of `Auction` maps to 2, and all other users map to the representative address 3. The three implicit classes have the labels *implicit*@0, *implicit*@1, and *implicit*@2, respectively.

It is not by coincidence that each smart contract appears as a participant of $\mathcal{C}$. In fact, nearly all MiniSol programs require that every smart contract appear as a participant[1]. This is because the transition function rejects actions that originate from a smart contract. This observation is made precise in Theorem 6.

Theorem 6. Let $\mathcal{C}$ be a bundle with at least one non-reverting transaction f_p, and $((N_A \cup N_A, E, \delta), \mu, \eta)$ be a PTG of $\mathcal{C}$. If $a \in \mathbb{N}$ is the address of a smart contract or the zero-account, then $\forall p \in \mathsf{inputs}(\mathcal{C}, \mathbb{N}) \cdot \exists(\mu(p), l, \eta(p, a)) \in \delta$.

Proof. Recall that $[\![\mathcal{C}]\!]_{\mathcal{M}}$ reverts whenever `tx.origin` is the address of a smart contract or the zero-account. Then $\mathsf{client}(p, 0) = a^* \neq a$. By the assumption that f_p is non-reverting, there exists an $N \in \mathbb{N}$, $s, s' \in \mathsf{control}(\mathcal{C}, [N])$, and $\mathbf{u}, \mathbf{u}' \in \mathsf{user}(\mathcal{C}, [N])^N$ such that $(s', \mathbf{u}') = f_p(s, \mathbf{u})$. Construct $\mathbf{v} \in \mathsf{user}(\mathcal{C}, [N])$ such that $\mathsf{map}(\mathbf{v}_{a^*}) = \mathsf{map}(\mathbf{u}_{a^*})$, $\mathsf{id}(\mathbf{v}_{a^*}) = a$, and $\forall i \in [N]/\{a^*\} \cdot \mathbf{v}_i = \mathbf{u}_i$. Since $(\mathsf{client}(p, 0) = a^*) \wedge (\mathcal{M}(a^*) = a^*) \wedge (\mathsf{id}(\mathbf{v}_{a^*}) = a)$, then $f_p(s, \mathbf{v})$ must revert. As a direct result, $(s, \mathbf{v}) = f_p(s, \mathbf{v})$. Then, $(s, \mathbf{v})$ is a witness to the participation of a. By Theorem 5, a must appear in $\mathsf{pt}(\mathcal{C}, N, p)$. Therefore, by Definition 4.2.1, $\exists(\mu(p), l, \eta(p, a)) \in \delta$. □

4.3 The PTGBuilder Abstraction

For every PT, there are many over-approximating PTGs. The weakest PTG (see Fig. 4.1b) maps every action to one vertex, every user to another vertex, and then joins the two vertices through a single edge. The single edge is labelled by every participation class, including a single implicit class. Fig. 4.1a shows a simple, yet stronger, PTG for Fig. 3.1. First, note that there are three implicit participants, represented by addresses 0 to 2, with labels *implicit*@0 through to *implicit*@2. Next, observe that any arbitrary user can become the manager. Finally, the distinctions between actions are ignored. Thus, there are four user vertices, three which are mapped to the zero-account, `Auction`, and `TimedAuctionManager`, and another that is mapped to all other users. Such a PTG is constructed automatically using an algorithm named `PTGBuilder`.

Let $\mathsf{lits}(\mathcal{C})$ denote the literal addresses that appear in $\mathcal{C}$, together with an address for each contract and the zero-account. `PTGBuilder` takes a bundle $\mathcal{C}$ and returns a PTG for $\mathcal{C}$. The implicit classes are $L := \{\mathit{implicit}@a \mid a \in \mathsf{lits}(\mathcal{C})\}$, where $\mathit{implicit}@a$ signifies

[1]The MiniSol programs that do not satisfy this property are those that revert on every action. A simple example is a smart contract whose single method starts with `require(false)`. Such programs are excluded by the existence of f_p in Theorem 6

implicit participation from the user at address a. The vertices, edges, and labelling function are computed using taint analysis [39]. Tainted sources include input address variables (i.e., clients), state address variables (i.e., roles), and literal address variables. Sinks include memory writes, comparison expressions, and mapping accesses. `PTGBuilder` first performs this taint analysis to compute $(\text{Input}, \text{State}, \text{Literal})$, where:

1. Input is a set that contains the index of each input address variable that propagates to a sink;
2. State is a set that contains the index of each state variable that propagates to a sink;
3. Literal is a set that contains each literal address that propagates to a sink.

`PTGBuilder` then uses $(\text{Input}, \text{State}, \text{Literal})$ to construct a PTG, $((N_A \cup N_P, E, \delta), \mu, \eta)$, that is a least solution to the following rules:

1. Let $a^\star = \max_{a \in \text{Literal}}\{a\} + 1$ and $p^\star = (\mathbf{0}, \mathbf{0}) \in \mathsf{inputs}(\mathcal{C}, [1])$;
2. $N_A = \text{Literal} \cup \{a^\star\}$, $N_P = \{p^\star\}$, and $E = N_A \times N_P$;
3. $\delta = \{(e, \mathit{explicit@}i) \mid e \in E \wedge i \in \text{Input}\} \cup \{(e, \mathit{transient@}i) \mid e \in E \wedge i \in \text{State}\} \cup \{((a, p), \mathit{implicit@}a) \mid (a, p) \in E \wedge a \in \text{Literal}\}$;
4. $\mu = \{(p, p^\star) \mid p \in \mathsf{inputs}(\mathcal{C}, \mathbb{N})\}$;
5. $\eta = \{(p^\star, a, a^\star \mid a \in \mathbb{N} \setminus \text{Literal}\} \cup \{(p^\star, a, a) \mid a \in \text{Literal}\}$.

The correctness of `PTGBuilder` relies on the structure of MiniSol, and on the over-approximate nature of taint analysis. Recall that in MiniSol, communication is limited to address comparisons, map accesses, and storing addresses in state variables (i.e., assigning roles). Therefore, the taint analysis employed by `PTGBuilder` strictly over-approximates all communication in $\mathcal{C}$. The seven rules then formalize the intuition behind Fig. 4.1a. Rule 1 selects a unique address $a^\star \notin$ Literal and a unique action. Rule 2 ensures that every literal address has a unique vertex, and that all address vertices connect to p^*. The first term of rule 3 states that if an input address is never used, then the corresponding client is not an explicit participant. The second term of rule 3 states that if a state address is never used, then the corresponding user is not a transient participant. The third term of rule 3 states that if an address is not literal, then the corresponding user is not an implicit participant. Rules 4 and 5 define μ and η as expected. Note that in MiniSol, implicit participation

must result from literal addresses, since addresses do not support arithmetic operations, and since numeric expressions cannot be cast to addresses. This discussion is summarized by Theorem 7.

Theorem 7. Let $\mathcal{C}$ be a bundle. If (G, μ, η) is returned by `PTGBuilder` when applied to $\mathcal{C}$, then (G, μ, η) is a PTG for $\mathcal{C}$.

4.4 Summary

By re-framing smart contracts with rendezvous synchronization, each transaction can be interpreted as communication between several users. The communication patterns of these users are captured by the corresponding PT. A PTG over-approximates PTs of all transactions, and is automatically constructed using `PTGBuilder`. A PTG provides useful insight into the behaviour of a smart contract, such that each smart contract belongs to its own participation class (i.e., Theorem 6). In Chapter 5, it is shown that the PTG is crucial for smart contract PCMC, as it provides an upper bound on the number of equivalence class, and the users in each equivalence class.

Chapter 5

Locality in Smart Contracts

This chapter applies local symmetry analysis to MiniSol bundles. First, Section 5.1 defines a notion of locality, and shows that every smart contract can be executed locally. In this setting, locality is defined by the participants of a transaction. It is shown that given a global state $(c, \mathbf{u})$, the transaction f_p can be determined using the local state $(c, \mathbf{v})$, where $\mathbf{v}$ is the projection of $\mathbf{u}$ onto the participants of f_p. Second, Section 5.2 shows that non-implicit participants are locally symmetric and can be readdressed without impacting the outcome of a transaction. As a result, all local transactions can be executing using an abstract address domain of finite size. Third, Section 5.3 shows that each implicit participant can be permanently readdressed through source code manipulation, without impacting local behaviour. In practice, this means that given a MiniSol bundle $\mathcal{C}$, it is possible to construct a new bundle $\mathcal{C}'$, such that $\mathcal{C}'$ simulates $\mathcal{C}$, and $\mathsf{lits}(\mathcal{C}') = [\,|\mathsf{lits}(\mathcal{C})|\,]$. An overview of each symmetry is illustrated in Fig. 5.1.

The chapter concludes by defining local bundles abstractions. Intuitively, a local bundle is a non-deterministic, finite-state abstraction of a global smart contract bundle. The address of each user is selected from an abstract domain, and the data of each user is determined by a user-provided predicate. In Chapter 6, it is shown that smart contract PCMC reduces to the safety of local bundles. In Chapter 7, it is shown that local bundles can aid in bounded analysis.

5.1 Local Transaction Views

A MiniSol bundle C can be instantiated with an arbitrary number of users. However, the outcome of a transaction f_p depends only on the participants of p. As shown in

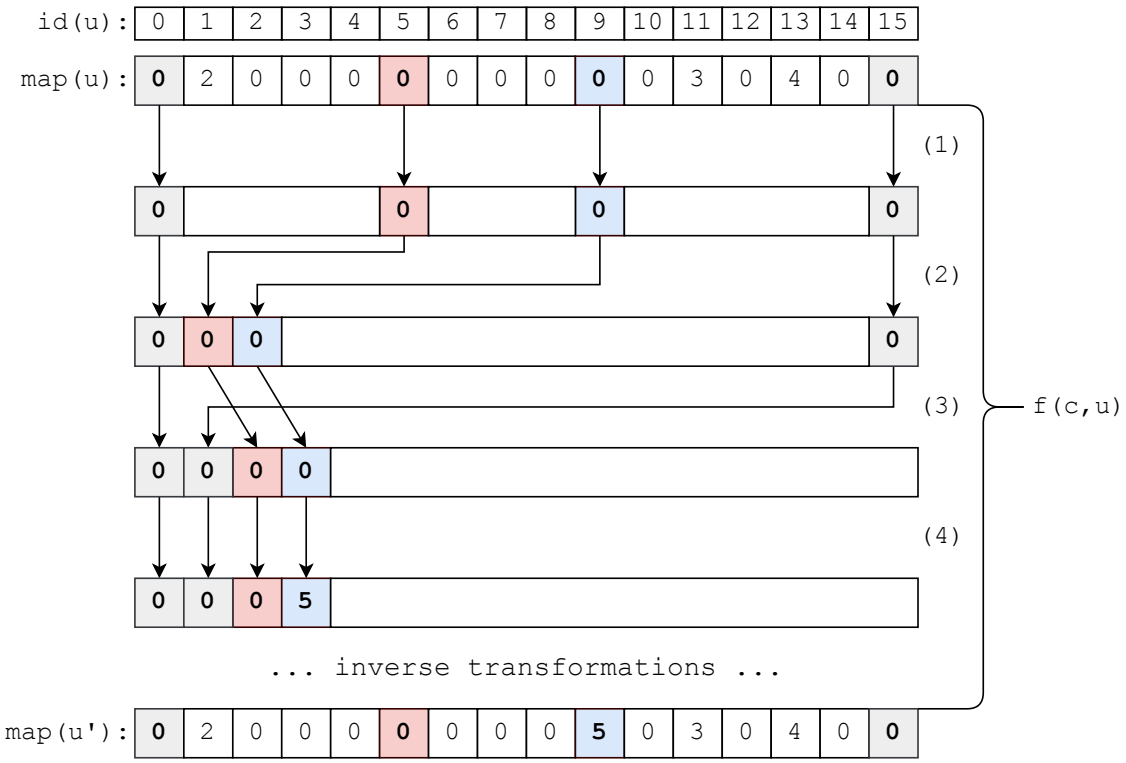


Figure 5.1: An illustration of a local transaction. A transaction $f_p(c, \mathbf{u})$ of `Auction` in Fig. 3.1 is depicted, such that in a 16-user network with owner at address 9 and auction at address 15, the user with address 5 increases their bid from 0 to 5. Step 1 performs a view projection as in Section 5.1. Step 2 constructs an abstract address domain as in Section 5.2. Step 3 applies smart contract normalization as in Section 5.3. Step 4 executes the transaction locally, and then the transformations are reversed to obtain $\mathbf{u}'$.

Chapter 4, the number of participants in f_p is bounded above by each PTG of C. This means that the outcome of f_p can be determined by inspecting a finite set of users, called the *transactional view* of p. The transactional view provides a natural notion of locality for MiniSol bundles.

Definition 5.1.1 (Transactional View)**.** Let $\mathcal{C}$ be a bundle, $N \in \mathbb{N}$, $p \in \mathsf{inputs}(\mathcal{C}, [N])$, $c \in \mathsf{control}(\mathcal{C}, [N])$, and $(E, T, I) = \mathsf{pt}(\mathcal{C}, N, p)$. The *transactional view of p relative to c* is the set of addresses $\mathsf{view}(p, c) := \{\mathsf{client}(p, i) \mid (i, a) \in E\} \cup \{\mathsf{role}(c, i) \mid (i, a) \in T\} \cup (I \cap [N])$.

Example 5.1.1. To illustrate Definition 5.1.1, a transactional view is constructed for Fig. 3.1 using a network size of 16. Let $\mathcal{C} = (\texttt{Auction})$ and assume for simplicity that the

address of `Auction` is 15. Consider the action $p = (\texttt{bid}, (5), (10)) \in \mathsf{inputs}(\mathcal{C}, [16])$, control state $c = ((9), (42, 42, 20, 7, \bot)) \in \mathsf{control}(\mathcal{C}, [N])$, and user configuration $\mathbf{u} \in \mathsf{user}(\mathcal{C}, [N])$. That is, the user at address 5 invokes `bid` with a message value of 10 while the auction is not stopped and the manager has address 9. As demonstrated by the running example in Section 4.1, this transaction has a single explicit participant (i.e., the sender), a single transient participant (i.e., the manager), and two implicit participants (i.e., the `Auction` contract and the zero-account). Relative to p and c the sender has address 5 and the manager has address 9. This gives a transactional view of $\mathsf{view}(p, c) = \{5\} \cup \{9\} \cup \{0, 15\}$. The outcome of p on $(c, \mathbf{u})$ is therefore determined by $\mathbf{u}_0$, $\mathbf{u}_5$, $\mathbf{u}_9$, $\mathbf{u}_{15}$. These four users form the *projection* of $\mathbf{u}$ onto $\mathsf{view}(p, c)$, as defined in Definition 5.1.2. □

Definition 5.1.2 (User Projection)**.** Let $\mathcal{C}$ be a bundle, $N \in \mathbb{N}$, $\mathcal{A} \subseteq [N]$, $M = |\mathcal{A}|$, and $\mathbf{u} \in \mathsf{user}(\mathcal{C}, [N])^N$. A configuration $\mathbf{v} \in \mathsf{user}(\mathcal{C}, \mathcal{A})^M$ is the *projection of* $\mathbf{u}$ *onto* $\mathcal{A}$, written $\mathbf{v} = \pi_{\mathcal{A}}(\mathbf{u})$, if there exists a total, injective, order-preserving mapping, $\lambda : [M] \to [N]$ such that $\mathcal{A} = \{\mathsf{id}(\mathbf{v}_i) \mid i \in [M]\}$ and $\forall i \in [M] \cdot \mathbf{v}_i = \mathbf{u}_{\lambda(i)}$.

Example 5.1.2. Recall $(p, c, \mathbf{u})$ from Example 5.1.1. Following Definition 5.1.2, the projection of $\mathbf{u}$ onto $\mathsf{view}(p, c)$ is $\mathbf{v} = (\mathbf{u}_0, \mathbf{u}_5, \mathbf{u}_9, \mathbf{u}_{15})$. In this example, $\lambda : [4] \to [16]$ such that $\lambda(0) = 0$, $\lambda(1) = 5$, $\lambda(2) = 9$, and $\lambda(3) = 15$. Given $(c, \mathbf{v})$, p, and λ, it is possible to compute the outcome of action p on state $(c, \mathbf{u})$. The key insight is that λ^{-1} defines a partial function from the addresses in $\mathsf{view}(p, c)$ to the indices of $\mathbf{v}$. For example, $\lambda^{-1}(5) = 1$ and $\lambda^{-1}(15) = 3$. This means that $[\![\mathcal{C}]\!]_{\lambda^{-1}}((c, \mathbf{v}), p)$ can be used to determine the projection of $[\![\mathcal{C}]\!]_{\mathcal{M}}((c, \mathbf{u}), p)$ onto $\mathsf{view}(p, c)$. For this reason, $((c, \mathbf{v}), p, \lambda)$ is called the *local transaction* of $((c, \mathbf{u}), p)$. In Theorem 8 it is proven that local transactions can be used to determine the outcome of a (global) transaction. An illustration is given by *Step 1* in Fig. 5.1. □

Lemma 1. Let $\mathcal{C}$ be a bundle, $N \in \mathbb{N}$, $(S, P, f, s_0) = \mathsf{global}(\mathcal{C}, N)$, $p \in P$, $(c, \mathbf{u}) \in S$, and $\mathbf{v} = \pi_{\mathsf{view}(c,p)}(\mathbf{u})$ with injection λ. If $f_p(c, \mathbf{u})$ succeeds, then $[\![\mathcal{C}]\!]_{\lambda^{-1}}((c, \mathbf{v}), p)$ succeeds.

Proof. Assume for the intent of contradiction that $[\![\mathcal{C}]\!]_{\lambda^{-1}}((c, \mathbf{v}), p)$ reverts. Clearly, $(c, \mathbf{u})$ and $(c, \mathbf{v})$ agree on the control state. Since f_p does not revert, then every address accessed by f_p must be in $[N]$. Then either, there exists an $a \in [N]$ accessed by $[\![\mathcal{C}]\!]_{\lambda^{-1}}$ such that $a \notin \mathsf{dom}(\lambda^{-1})$, or there exists an $a \in \mathsf{dom}(\lambda^{-1})$ such that $[\![\mathcal{C}]\!]_{\lambda^{-1}}$ and f_p disagree on the participant with address a. There are two cases to consider.

1. Assume for the intent of contradiction that a is accessed yet not in $\mathsf{dom}(\lambda^{-1})$. By construction of λ, $\mathsf{dom}(\lambda^{-1}) = \mathsf{view}(c, p)$. Then there exists $a \notin \mathsf{view}(c, p)$ such that a participates in f_p. However, by Definition 5.1.1, $\mathsf{view}(c, p)$ must contain the address of each participates in f_p, relative to c. By contradiction, $a \in \mathsf{dom}(\lambda^{-1})$.

2. Assume for the intent of contradiction that $[\![\mathcal{C}]\!]_{\lambda^{-1}}$ and f_p disagree on the participant with address a. By assumption, there exists an $a \in \mathsf{dom}(\lambda^{-1})$ such that $\mathbf{u}_a \neq \mathbf{v}_{\lambda^{-1}(a)}$. However, $\mathbf{v}_{\lambda^{-1}(a)}$ is defined to be $\mathbf{u}_{\lambda(\lambda^{-1}(a))} = \mathbf{u}_a$. By contradiction, $[\![\mathcal{C}]\!]_{\lambda^{-1}}$ and f_p agree the address of every participant during the transaction.

By contradiction, $[\![\mathcal{C}]\!]_{\lambda^{-1}}((c, \mathbf{v}), p)$ succeeds. □

Theorem 8. Let $\mathcal{C}$ be a bundle, $N \in \mathbb{N}$, $(S, P, f, s_0) = \mathsf{global}(\mathcal{C}, N)$, $p \in P$, $(c, \mathbf{u}) \in S$, $(c', \mathbf{u}') \in S$, $\mathcal{A} = \mathsf{view}(c, p)$, and $(\mathbf{v}, \mathbf{v}') = (\pi_{\mathcal{A}}(\mathbf{u}), \pi_{\mathcal{A}}(\mathbf{u}'))$ with injection λ. If $(c', \mathbf{u}') = f_p(c, \mathbf{u})$, then $\forall i \in [N] \backslash \mathcal{A} \cdot \mathbf{u}'_i = \mathbf{u}_i$ and $(c', \mathbf{v}') = [\![\mathcal{C}]\!]_{\lambda^{-1}}((c, \mathbf{v}), p)$.

Proof. The proof of Theorem 8 has two parts.

1. Assume for the intent of contradiction that $\forall i \in [N] \backslash \mathcal{A} \cdot \mathbf{u}'_i \neq \mathbf{u}_i$. Then $\exists i \in [N] \backslash \mathcal{A} \cdot \mathbf{u}'_i \neq \mathbf{u}_i$. Then by definition, $\mathbf{u}_i$ is influenced by f_p. By definition of $\mathsf{view}(c, p)$, $i \in \mathcal{A}$. By contradiction, $\forall i \in [N] \backslash \mathcal{A} \cdot \mathbf{u}'_i = \mathbf{u}_i$.

2. By definition of the transition function, $[\![\mathcal{C}]\!]_{\lambda^{-1}}$ is deterministic. Since $f_p(c, \mathbf{u})$ succeeds, then $[\![\mathcal{C}]\!]_{\lambda^{-1}}((c, \mathbf{v}), p)$ succeeds by Lemma 1. Assume for the intent of contradiction that $(c', \mathbf{v}') \neq [\![\mathcal{C}]\!]_{\lambda^{-1}}((c, \mathbf{v}), p)$. Since $[\![\mathcal{C}]\!]_{\lambda^{-1}}$ succeeds, only the users in $\mathbf{v}$ can participate in f_p. Since $[\![\mathcal{C}]\!]_{\lambda^{-1}}$ is deterministic, f_p must read a value that differs from $[\![\mathcal{C}]\!]_{\lambda^{-1}}$. However, the control states are the same, and $\mathbf{v} = \pi_{\mathcal{A}}(\mathbf{u})$. This contradicts the determinism of $[\![\mathcal{C}]\!]_{\lambda^{-1}}$. Therefore, $(c', \mathbf{v}') = [\![\mathcal{C}]\!]_{\lambda^{-1}}$.

Therefore, both claims of Theorem 8 must hold. □

5.2 User Symmetries and Abstract Address Domains

Theorem 8 reduces every MiniSol transaction to a bounded number of users. However, the address of each user is still parameterized by the size of the network. Therefore, the number of local transactions is also countably infinite. To enable reasoning about local transactions, this section identifies a finite set of equivalence classes that are closed under their transition relations. As a key result, it is shown that a representative for each equivalence class can be obtained by fixing a finite set of abstract addresses.

First, an equivalence relation is given relative to the numeric-typed data of a local transaction. For simplicity, $I \subseteq \mathsf{lits}(\mathcal{C})$ is used to denote that literal addresses in $\mathcal{C}$ that appear as implicits participants.

Definition 5.2.1 (Data Equivalence)**.** Let $\mathcal{C}$ be a bundle with m roles, d data, q clients, and r arguments, and $N \in \mathbb{N}$. Two local transactions of $\mathsf{global}(\mathcal{C}, N)$, $((c_u, \mathbf{u}), p_u, \lambda_u)$ and $((c_v, \mathbf{v}), p_v, \lambda_v)$ are *data equivalent*, written $((c_u, \mathbf{u}), p_u, \lambda_u) \sim_D ((c_v, \mathbf{v}), p_v, \lambda_v)$, when the following properties hold:

1. Both p_u and p_v invoke the same transaction;
2. $\forall i \in [r] \cdot \arg(p_u, i) = \arg(p_v, i)$;
3. $\forall a \in (I \cap [N]) \cdot \mathsf{map}(\mathbf{u}_{\lambda_u(a)}) = \mathsf{map}(\mathbf{v}_{\lambda_v(a)})$;
4. $\forall i \in [m] \cdot \mathsf{let}\ j = \mathsf{role}(c_u, i) \cdot \mathsf{let}\ k = \mathsf{role}(c_v, i) \cdot \mathsf{map}(\mathbf{u}_{\lambda_u(i)}) = \mathsf{map}(\mathbf{v}_{\lambda_v(j)})$;
5. $\forall i \in [q] \cdot \mathsf{let}\ j = \mathsf{client}(p_u, i) \cdot \mathsf{let}\ k = \mathsf{client}(p_v, i) \cdot \mathsf{map}(\mathbf{u}_{\lambda_u(j)}) = \mathsf{map}(\mathbf{v}_{\lambda_v(k)})$;
6. $\forall i \in [d] \cdot \mathsf{data}(c_u, i) = \mathsf{data}(c_v, i)$.

The rules of Definition 5.2.1 follow from the intuition of numeric data equivalence. Rule 1 states that the local transactions invoke the same method. Rule 2 states that the same arguments are passed to both invocations. Rules 3 through to 5 state that the local transactions assign the same mapping entries to users that share participation classes. Rule 6 states that the control process in both local transactions stores the same data. Since data equivalence is defined through equality, it follows naturally that data equivalence is an equivalence relation.

Lemma 2. Data equivalence is an equivalence relation.

Proof. If $\sim_D$ is reflexive, symmetric, and transitive, then $\sim_D$ is an equivalence relation.

1. Let $((c, \mathbf{u}), p, \lambda)$ be a local transaction. By definition, the domain of λ includes $I \cap [N]$, $\{\mathsf{role}(c, i) \mid i \in [m]\}$, and $\{\mathsf{client}(c, i) \mid i \in [q]\}$. Then all properties of $\sim_D$ are well-defined for $((c, \mathbf{u}), p, \lambda)$ and $((c, \mathbf{u},), p, \lambda)$. Since equality is reflexive, all properties are also satisfied. Then $((c, \mathbf{u}), p, \lambda) \sim_D ((c, \mathbf{u}), p, \lambda)$. Therefore, $\sim_D$ is reflexive.
2. Assume that $(s_1, p_1, \lambda_1) \sim_D (s_2, p_2, \lambda_2)$. Since equality is symmetric, then all properties of $\sim_D$ are also symmetric. Then $(s_2, p_2, \lambda_2) \sim_D (s_1, p_1, \lambda_1)$. Therefore, $\sim_D$ is symmetric.
3. Assume that $(s_1, p_1, \lambda_1) \sim_D (s_2, p_2, \lambda_2) \sim_D (s_3, p_3, \lambda_3)$. Since equality is transitive, then all properties of $\sim_D$ are also transitive. Then $((s_1, p_1, \lambda_1) \sim_D (s_3, p_3, \lambda_3)$. Therefore, $\sim_D$ is transitive.

Therefore, $\sim_D$ is an equivalence relation. □

Next, an equivalence relation is given relative to address-typed variables. This relation is weaker than data equivalence, since the address-type is nominal whereas the numeric-type is ordered. Intuitively, two local transactions are equivalent, relative to their address-typed variables, if they preserve equality between roles, clients, and implicit addresses. This ensures that the local transactions agree on the participation classes (i.e., names) assigned to each user.

Definition 5.2.2 (Name Equivalence)**.** Let $\mathcal{C}$ be a bundle with m roles and q clients, and $N \in \mathbb{N}$. Two local transaction of $\mathsf{global}(\mathcal{C}, N)$, $((c_u, \mathbf{u}), p_u, \lambda_u)$ and $((c_v, \mathbf{v}), p_v, \lambda_v)$, are *name equivalent*, written $((c_u, \mathbf{u}), p_u, \lambda_u) \sim_N ((c_v, \mathbf{v}), p_v, \lambda_v)$, when there exists a permutation $\tau : [N] \to [N]$ such that:

1. $\forall a \in (I \cap [N]) \cdot \tau(a) = a$;
2. $\forall i \in [m] \cdot \tau(\mathsf{role}(c_u, i)) = \mathsf{role}(c_v, i)$;
3. $\forall i \in [n] \cdot \tau(\mathsf{client}(p_u, i)) = \mathsf{client}(p_v, i)$.

In Definition 5.2.2, τ is used to map the participants of $((c_u, \mathbf{u}), p_u, \lambda_u)$ onto the participants of $((c_v, \mathbf{v}), p_v, \lambda_v)$. Rule 1 ensures that τ does not rename implicit participants, since implicit participants have fixed addresses. Rules 2 and 3 ensure that τ renames each role and client consistently. Since permutations preserve equality, τ ensures that the local transactions agree on participation classes. Furthermore, permutation properties of τ ensure that name equivalence is an equivalence relations.

Lemma 3. Address equivalence is an equivalence relation.

Proof. If $\sim_N$ is reflexive, symmetric, and transitive, then $\sim_N$ is an equivalence relation.

1. Let (s, p, λ) be a local transaction and τ be the identity permutation for $[N]$. Then $\forall a \in [N] \cdot \tau(a) = a$. By definition, $(s, p, \lambda) \sim_N (s, p, \lambda)$ with corresponding permutation τ. Therefore, $\sim_N$ is reflexive.
2. Let $(s_1, p_1, \lambda_1) \sim_N (s_2, p_2, \lambda_2)$ with corresponding permutation τ. Since permutations form a group, there exists an inverse permutation τ^{-1}. Since $\forall a \in (I \cap [N]) \cdot \tau(a) = a$, then $\forall a \in (I \cap [N]) \cdot \tau^{-1}(a) = a$. By definition, $(s_2, p_2, \lambda_2) \sim_N (s_1, p_1, \lambda_1)$ with corresponding permutation τ^{-1}. Therefore, $\sim_N$ is symmetric.

3. Let $(s_1, p_1, \lambda_1) \sim_N (s_2, p_2, \lambda_2)$ with corresponding permutation τ_1, and $(s_2, p_2, \lambda_2) \sim_N (s_3, p_3, \lambda_3)$ with corresponding permutation τ_2. Since permutations form a group, there exists a permutation $\tau = \tau_2 \circ \tau_1$. Since $\forall a \in (I \cap [N]) \cdot \tau_1(a) = \tau_2(a) = a$, then $\forall a \in (I \cap [N]) \cdot \tau(a) = a$. By definition, $(s_1, p_1, \lambda_1) \sim_N (s_3, p_3, \lambda_3)$ with corresponding permutation τ. Therefore, $\sim_N$ is transitive.

Therefore, $\sim_N$ is an equivalence relation. □

Now that $(\sim_D)$ and $(\sim_N)$ are defined, their closure under local transitions is considered. Theorem 9 shows that in general, $(\sim_D)$ is not closed under local transitions. A similar result can be proven for $(\sim_N)$. Theorem 10 shows that $(\sim_D) \cap (\sim_N)$ is closed under all local transitions. Since equivalence relations are closed under intersection, the relation $(\sim_D) \cap (\sim_N)$ is also an equivalence relation.

Theorem 9. Let $\mathcal{C}$ be a bundle with local transactions (s_u, p_u, λ_u) and (s_v, p_v, λ_v) such that $(s_u, p_u, \lambda_u) \sim_D (s_v, p_v, \lambda_v)$ and $(s_u, p_u, \lambda_u) \not\sim_N (s_v, p_v, \lambda_v)$. There exists an implementation of $[\![\mathcal{C}]\!]_{\mathcal{M}}$ such that $([\![\mathcal{C}]\!]_{\lambda_u}(s_u, p_u), p_u, \lambda_u) \not\sim_D ([\![\mathcal{C}]\!]_{\lambda_v}(s_v, p_v), p_v, \lambda_v)$.

Proof. Write $(c_u, \mathbf{u}) = s_u$ and $(c_v, \mathbf{v}) = s_v$. Assume that $(s_u, p_u, \lambda_u) \not\sim_N (s_v, p_v, \lambda_v)$. Then for some $a, x, y \in [N]$, there exists two properties of $\sim_N$ such that $\tau(a) = x$ and $\tau(a) = y$ with $x \neq y$. There are five cases to consider.

1. Assume that property 1 contradicts property 2. Then $a \in (I \cap [N])$, and there exists an $i \in [m]$ such that $\mathsf{role}(c_u, i) = a$ and $\mathsf{role}(c_v, i) \neq a$. Construct $[\![\mathcal{C}]\!]_{\mathcal{M}}$ such that the 0-th state variable is incremented if the i-th role has the address a.

2. Assume that property 1 contradicts property 3. Then $a \in (I \cap [N])$ and there exists an $i \in [n]$ such that $\mathsf{client}(p_u, i) = a$ and $\mathsf{client}(p_v, i) \neq a$. Construct $[\![\mathcal{C}]\!]_{\mathcal{M}}$ such that the 0-th state variable is incremented if the i-th client has the address a.

3. Assume that property 2 contradicts property 2. Then there exists $i, j \in [m]$ such that $\mathsf{role}(c_u, i) = \mathsf{role}(c_v, j)$ and $\mathsf{role}(c_u, i) \neq \mathsf{role}(c_v, j)$. Construct $[\![\mathcal{C}]\!]_{\mathcal{M}}$ such that the 0-th state variable is incremented if the i-th and j-th role are held by the same user.

4. Assume that property 2 contradicts property 3. Then there exists $i \in [m]$ and $j \in [n]$ such that $\mathsf{role}(c_u, i) = \mathsf{client}(p_u, j)$ and $\mathsf{role}(c_v, i) \neq \mathsf{client}(p_u, j)$. Construct $[\![\mathcal{C}]\!]_{\mathcal{M}}$ such that the 0-th state variable is incremented if the j-th client holds the i-th role.

5. Assume that property 3 contradicts property 3. Then there exists $i, j \in [n]$ such that $\mathsf{client}(p_u, i) = \mathsf{client}(p_u, j)$ and $\mathsf{client}(p_v, i) \neq \mathsf{client}(p_v, j)$. Construct $[\![\mathcal{C}]\!]_{\mathcal{M}}$ such that the 0-th state variable is incremented if the i-th and j-th clients are the same user.

In each case, $[\![\mathcal{C}]\!]_{\mathcal{M}}$ compares literal addresses, roles and clients. Such comparisons are expressible within the MiniSol language. Furthermore, $\mathsf{data}\,(c'_u, 2) \neq \mathsf{data}\,(c'_v, 2)$. Therefore, $([\![\mathcal{C}]\!]_{\lambda_u}(s_u, p_u), p_u, \lambda_u) \not\sim_D ([\![\mathcal{C}]\!]_{\lambda_v}(s_v, p_v), p_v, \lambda_v)$. □

Theorem 10. Let $\mathcal{C}$ be a bundle with local transactions (s_u, p_u, λ_u) and (s_v, p_v, λ_v) and $(\sim)$ be defined as $(\sim_D) \cap (\sim_N)$. If $(s_u, p_u, \lambda_u) \sim (s_v, p_v, \lambda_v)$ with permutation τ, then $([\![\mathcal{C}]\!]_{\lambda_u}(s_u, p_u), p_u, \lambda_u) \sim ([\![\mathcal{C}]\!]_{\lambda_v}(s_v, p_v), p_v, \lambda_v)$ with permutation τ.

Proof. Let $s'_u = [\![\mathcal{C}]\!]_{\lambda_u}(s_u, p_u)$, $s'_v = [\![\mathcal{C}]\!]_{\lambda_v}(s_v, p_v)$, $(c_u, \mathbf{u}) = s_u$ and $(s_v, \mathbf{v}) = s_v$. It must be shown that $(s'_u, p_u, \lambda_u) \sim_D (s'_v, p_v, \lambda_v)$ and $(s'_u, p_u, \lambda_u) \sim_N (s'_v, p_v, \lambda_v)$. However, the proof either case is symmetric, so assume for the intent of contradiction that $(s'_u, p_u, \lambda_u) \not\sim_D (s'_v, p_v, \lambda_v)$. Then $s'_u \neq s'_v$. Since $[\![\mathcal{C}]\!]_{\mathcal{M}}$ is deterministic, then $[\![\mathcal{C}]\!]_{\mathcal{M}}$ must differentiate between the transactions (s_u, p_u) and (s_v, p_v). Since $(s_u, p_u, \lambda_u) \sim_D (s_v, p_v, \lambda_u)$, then (s_u, p_u) and (s_v, p_v) assign the same value to each numeric-typed variable. Therefore, $[\![\mathcal{C}]\!]_{\mathcal{M}}$ must differentiate between the value of an address-typed variable in (s_u, p_u), and the same address-typed variable in (s_v, p_v). However, $(s_u, p_u, \lambda_u) \sim_N (s_v, p_v, \lambda_v)$, so (s_u, p_u) and (s_v, p_v) must also agree on the equality of all roles, clients, and implicit participants. Therefore, there exists an $i \in [m]$ and $a \in [N]$ such that $[\![\mathcal{C}]\!]_{\mathcal{M}}$ can identify that $\mathsf{role}(c_u, i) = a$ and $\mathsf{role}(c_v, i) \neq a$ (a symmetric argument holds for clients). By definition, a is an implicit participant, and as a result, $\tau(a) = a$. However, $\tau(\mathsf{role}(c_u, i)) = \mathsf{role}(c_v, i)$. By contradiction, $(s'_u, p_u, \lambda_u) \sim_D (s'_v, p_v, \lambda_v)$. In conclusion, $(s'_u, p_u, \lambda_u) \sim (s'_v, p_v, \lambda_v)$. □

Theorem 10 states that all local transactions can be analyzed by considering only the representatives of $(\sim_D) \cap (\sim_N)$. However, it is not yet clear whether the number of equivalence classes induced by $(\sim_D) \cap (\sim_N)$ is finite. Recall that the set of all local transactions is countably infinite due to the countably infinite set of addresses. Therefore, if all representatives of $(\sim_D) \cap (\sim_N)$ can be defined using a fixed set of abstract addresses, then $(\sim_D) \cap (\sim_N)$ has finitely many equivalence classes. Intuitively, the set of abstract addresses must be large enough to preserve equality between clients, roles, and literal addresses. This intuition is made precise by Theorem 11, with $\mathcal{A}$ denoting the set of abstract addresses.

Theorem 11. Let $\mathcal{C}$ be a bundle with m roles and q clients, $N \in \mathbb{N}$, $((c, \mathbf{u}), p, \lambda)$ be a local transaction of $\mathsf{global}(\mathcal{C}, N)$, and $\mathcal{A} \subseteq [N] \backslash I$. If $|\mathcal{A}| \geq \max\{m + q, |[N] \backslash I|\}$, then there exists a permutation $\tau : [N] \to [N]$ such that:

1. $\forall i \in [m] \cdot \tau(\mathsf{role}(c, i)) \in (I \cup \mathcal{A})$;
2. $\forall i \in [q] \cdot \tau(\mathsf{client}(p, i)) \in (I \cup \mathcal{A})$;
3. $\forall a \in I \cap [N] \cdot \tau(a) = a$.

Proof. Let (X, Y) be a partition of $\mathsf{view}(c, p)$ such that $Y = I \cap [N]$. Fix arbitrary orders over the elements of $\mathcal{A}$ and X, such that $\mathcal{A} = \{a_0, \ldots, a_{|\mathcal{A}|-1}\}$ and $X = \{x_0, \ldots, x_{|X|-1}\}$. By Definition 5.1.2, $|X| \leq m + q$ and $|X| \subseteq [N] \backslash I$. Then $|X| \leq \max\{m + q, |[N] \backslash I|\} \leq |\mathcal{A}|$. Since $|X| \leq |\mathcal{A}|$, there exists a relation $\tau \subseteq [N] \times [N]$ such that:

1. $\forall i \in [\,|X|\,] \cdot \tau(x_i) = a_i$;
2. $\forall a \in I \cap [N] \cdot \tau(a) = a$;
3. $\forall a \in [N] \backslash (X \cup Y) \cdot \tau(a) = a$.

Since $X \cap Y = \emptyset$ and $Y = I \cap [N]$, τ is a permutation. By Definition 5.1.2, if follows that $\forall i \in [m] \cdot \mathsf{role}(c, i) \in (X \cup Y)$ and $\forall i \in [q] \cdot \mathsf{client}(p, i) \in (X \cup Y)$. By construction of τ, $\forall a \in (X \cup Y) \cdot \tau(a) \in (I \cup \mathcal{A})$. Therefore, τ satisfies Theorem 11. □

Example 5.2.1. Recall $((c, \mathbf{v}), p, \lambda)$ from Example 5.1.2. In Example 5.1.1, it was shown that $\mathsf{view}(p, c) = \{0, 5, 9, 15\}$. This example constructs a local transaction $((c', \mathbf{v}'), p', \lambda')$, with abstract address domain $\mathcal{A} = \{1, 2\}$, such that $((c', \mathbf{v}), p, \lambda) \sim_N ((c', \mathbf{v}'), p', \lambda)$. The first step in this example is to find the permutation τ corresponding to $\sim_N$. A method to find τ is given in the proof of Theorem 11. Following the proof, $X = \{5, 9\}$ and $Y = \{1, 15\}$. Since the orderings placed on X and $\mathcal{A}$ are arbitrary, assume the standard ordering for integers. Then $x_0 = 5$, $x_1 = 9$, $a_0 = 1$ and $a_1 = 2$. Consequently, $\tau(5) = 1$, $\tau(9) = 2$, and $\forall a \in [16] \backslash \{5, 9\} \cdot \tau(a) = a$. Then $c' = ((\tau(9)), (42, 42, 20, 7, \bot))$, $p' = (\mathtt{bid}, (\tau(5)), (10))$, and $\mathbf{v}' = ((0, \mathsf{map}(\mathbf{v}_0)), (1, \mathsf{map}(\mathbf{v}_5)), (2, \mathsf{map}(\mathbf{v}_9)), (15, \mathsf{map}(\mathbf{v}_{15})))$. An illustration is given by (2) in Fig. 5.1. □

In Example 5.2.1, the control state c was transformed into c', the user configuration $\mathbf{v}$ was transformed into $\mathbf{v}'$, and the action p was transformed into p'. Each transformation is referred to as a τ-renaming, and is written $\mathsf{rename}(\mathcal{C}, \tau, \cdot)$. For c and p, the τ-renaming permutes address variables. In the case of $\mathbf{v}$, the τ-renaming also permutes the vector to preserve address ordering. This permutation is not necessary, but simplifies later proofs.

5.3 Renaming Implicit Participants

In Section 5.2, it was shown that $(\sim_D) \cap (\sim_N)$ induces a finite number of equivalence classes among local transactions, and that representatives for these equivalence classes can be found by enumerating an abstract address domain, $(\mathcal{A} \cup I)$. One limitation is that the abstract address domain may be non-consecutive. For example, in Example 5.2.1, $(\mathcal{A} \cup I) = [3] \cup \{15\}$. If non-consecutive addresses matter, then they must be taken into account during smart contract analysis. As a positive result, this section shows that for every MiniSol smart contract, a set of correctness-preserving source code transformations exist such that $(\mathcal{A} \cup I)$ is consecutive.

The key insight behind this result is that literal addresses are merely names for special users. If all instances of a literal address are replaced in the source code, then the set of implicit participants is changed accordingly. Usually, such an equivalence would be proven through bisimulation. However, changing implicit participation also changes the relation between states and actions, whereas standard simulations assume that actions are unchanged. In this section, a new notion of simulation is introduced that allows for actions to be similar relative to similar states. Such a simulation is called a *simulation under input relabelling*. First, the impact of such simulations on guarded safety properties is considered.

Usually, simulations are used to prove simulation-invariant properties. Unfortunately, guarded safety properties are not invariant to τ-renamings. For example, if $\tau(5) = 10$ and 5 is a literal guard of φ, then φ is not invariant to the renaming induced by τ. However, if the literal guards in φ were updated according to τ, then this new property would be preserved under simulation. This intuition is captured by simulation equivalent properties in Definition 5.3.1. In Definition 5.3.2, the τ-renaming of a guarded property is defined, and in Theorem 12 it is proven that a guarded safety property and its τ-renaming are simulation equivalent with respect to the τ-renaming of states.

Definition 5.3.1 (Simulation Equivalent)**.** Let (S_1, L_1, f, s_0) and (S_2, L_2, g, t_0) be LTSs, and $\sigma \subseteq S_1 \times S_2$. Properties φ_1 and φ_2 are *simulation equivalent* with respect to σ if $\forall\, ((s_1, t_1), \ldots, (s_k, t_k)) \in \sigma^* \cdot (s_1, \ldots, s_k) \models \varphi_1 \iff (t_1, \ldots, t_k) \models \varphi_2$.

Definition 5.3.2 (Guard Renaming)**.** Let φ be the guarded k-safety property given by (L, R, ξ), and τ be a permutation. The *τ-renaming of φ* is the guarded k-safety property given by (L', R, ξ), where $L' = \{(i, \tau(a)) \mid (i, a) \in L\}$.

Theorem 12. Let $\mathcal{C}$ be a bundle, $N \in \mathbb{N}$, φ be the guarded k-safety property given by (L, R, ξ), and $\tau : [N] \to [N]$ be a permutation. If φ_2 is the τ-renaming of φ_1, given by (L_2, R, ξ), then φ_1 and φ_2 are simulation equivalent with respect to $\sigma(s) = \mathsf{rename}(\mathcal{C}, \tau, s)$.

Proof. To prove that φ_1 and φ_2 are simulation equivalent, both directions of ($\iff$) must be proven. Since $L_1 = \{(i, \tau^{-1}(a)) \mid (i, a) \in L_2\}$, the ($\Rightarrow$) direction implies the ($\Leftarrow$) direction. Assume for the intent of contradiction, that the ($\Rightarrow$) does not hold. Then there exists $(s_0, \ldots, s_m) \in (S_1)^*$, such that $(s_0, \ldots, s_m) \models \varphi_1$ and $(\sigma(s_9), \ldots, \sigma(s_m)) \not\models \varphi_2$. Since φ_1 and φ_2 are safety properties, there exists an $i \in [m+1]$ such that $s_i \models \varphi_1$ and $\sigma(s_i) \not\models \varphi_2$. Then $\sigma(s_i)$ satisfies the guards of φ_2 and $\sigma(s_i) \not\models \xi$. Let c be the control state of s_i, $\mathbf{u}$ be the user configuration of s_i, and $\{j_1, \ldots, j_k\}$ be the indices of $\mathbf{u}$ that correspond to the counterexample. There are three cases to consider.

1. Assume that s_i satisfies the guards of φ_1. Then $s_i \models \xi$. Since ξ does not depend on address variables, $\sigma(s_i) \models \xi$. Consequently, $\sigma(s_i) \models \gamma \Rightarrow \xi$ for any choice of γ. Therefore, $\varphi_2(\sigma(c), \sigma(\mathbf{u}_{j_1}), \ldots, \sigma(\mathbf{u}_{j_k}))$ is valid. By contradiction, s_i must violate at least one guard of φ_1.

2. Assume that s_i violates a literal guard $(l, a) \in L_1$ and let $b = j_l$. Then $\mathbf{u}_b \neq a$. Since τ is a permutation, then $\tau(\mathbf{u}_b) \neq \tau(a)$. Consequently, $\sigma(s_i)$ violates the literal guard $(i, \tau(a)) \in L_2$. Therefore, $\varphi_2(\sigma(c), \sigma(\mathbf{u}_{j_1}), \ldots, \sigma(\mathbf{u}_{j_k}))$ is valid. By contradiction, s_i must violate at least one role guard of φ_1.

3. Assume that s_i violates a role guard $(l, r) \in R$ and let $b = j_l$. Then $\mathsf{id}(\mathbf{u}_b) = \mathsf{role}(c, r)$. Since τ is a permutation, then $\tau(\mathsf{id}(\mathbf{u}_b)) = \tau(\mathsf{role}(c, r))$. Consequently, $\sigma(s_i)$ violates the role guard $(l, r) \in R$. Therefore, $\varphi_2(\sigma(c), \sigma(\mathbf{u}_{j_1}), \ldots, \sigma(\mathbf{u}_{j_k}))$ is valid. By contradiction, $\sigma(s_i) \models \varphi_2$.

By contradiction, $(\sigma(s_0), \ldots, \sigma(s_m)) \models \varphi_2$. □

Next, the standard definitions of simulations and bisimulations (see Section 2.6) are generalized to similar inputs. The key difference is that for every $(s, t) \in \sigma$, there exists a function $\pi_{(s,t)}$ that relates actions of one LTS to actions of the other LTS. In the special case where $\forall (s, t) \in \sigma \cdot \pi_{(s,t)}(p) = p$, a standard simulation is obtained. In the case of MiniSol smart contract, $(s, t) \in \sigma$ if t is a particular τ-renaming of s. Respectively, $q = \pi_{(s,t)}(p)$ if q is also a τ-renaming of p.

Definition 5.3.3 (Simulation Under Relabelling)**.** Let (S_1, L_1, f, s_0) and (S_2, L_2, g, t_0) be LTSs. A relation $\sigma \subseteq S_1 \times S_2$ is a *simulation under input relabelling*, with respect to the family of total functions, $\{\pi_{(s,t)} : L_1 \to L_2 \mid (s, t) \in \sigma\}$, if the following requirements hold:

1. $(s_0, t_0) \in \sigma$;

2. $\forall s \in S_1 \cdot \exists t \in S_2 \cdot (s,t) \in \sigma$;

3. $\forall (s,t) \in \sigma \cdot \forall p \in L_1 \cdot \forall s' \in f(s,p) \cdot \exists t' \in g\left(t, \pi_{(s,t)}(p)\right) \cdot (s',t') \in \sigma$.

Definition 5.3.4 (Bisimulation Under Relabelling)**.** Let (S_1, L_1, f, s_0) and (S_2, L_2, g, t_0) be LTSs. A relation $\sigma \subseteq S_1 \times S_2$ is a *bisimulation under input relabelling*, with respect to the family of total functions, $R = \{\pi_{(s,t)} : L_1 \to L_2 \mid (s,t) \in \sigma\}$, if the following requirements hold:

1. For each $(s,t) \in \sigma$, the relation $\pi^{\star}_{(t,s)}(q) := {\pi_{(s,t)}}^{-1}(q)$ defines a total function;

2. σ is a simulation under input relabelling, with respect to R;

3. σ^{-1} is a simulation under input relabelling, with respect to $\{\pi^{\star}_{(t,s)} \mid (s,t) \in \sigma\}$.

Definition 5.3.5 (Normalized Bundle)**.** Let $\mathcal{C}$ be a bundle and a_i be the i-th largest address in $\mathsf{lits}(\mathcal{C})$. The *normalization* of $\mathcal{C}$, rewritten $\mathsf{norm}(\mathcal{C})$, is the bundle obtained by the source code transformation that replaces each instance of a_i with i.

The main result for the rest of this section is that $\mathsf{global}^{\circ}(\mathcal{C}, \mathcal{A})$ is bisimulated by $\mathsf{global}(\mathsf{norm}(\mathcal{C}), |\mathcal{A}|)$ under input relabelling (Theorem 13) while preserving guarded universal safety properties (Theorem 14)[1]. In order words, all local analysis can be performed using a consecutive abstract address domain $[\,|\mathcal{A}|\,]$. An example in given in Example 5.3.1, before proceeding with the corresponding proofs.

Example 5.3.1. Recall the `Auction` contract $\mathcal{C}$ with state $(c', \mathbf{v}')$, action p', and abstract address domain $\mathcal{A} = [3] \cup \{15\}$ in Example 5.2.1. By definition, $\mathcal{C}' = \mathsf{norm}(\mathcal{C})$ is the bundle obtained by taking `Auction` from Fig. 3.1, and then mapping the zero-account to itself (as 0 is the smallest address in $\mathsf{lits}(\mathcal{C})$), and the smart contract account to address 1 (as it has the only other literal address). Following from Theorem 11, all other addresses can then be mapped onto $\{2,3\}$ as desired. This corresponds to applying a renaming $\tau : [16] \to [16]$ such that $\tau(0) = 0$, $\tau(15) = 1$, $\tau(1) = 2$, and $\tau(2) = 3$. Then $\tau(\mathcal{A}) = [4]$, $c'' = \mathsf{rename}(\mathcal{C}', \tau, c') = ((3), (42, 42, 20, 7, \bot))$, $p'' = \mathsf{rename}(\mathcal{C}', \tau, p') = (\mathtt{bid}, (2), (10))$, and $\mathbf{u}'' = \mathsf{rename}(\mathcal{C}, \tau, \mathbf{v}') = ((0, \mathsf{map}(\mathbf{v}_0)), (1, \mathsf{map}(\mathbf{v}_{15})), (2, \mathsf{map}(\mathbf{v}_5)), (3, \mathsf{map}(\mathbf{v}_9)))$. An illustration is given by (3) in Fig. 5.1. □

Lemma 4. Let $\mathcal{C}$ be a bundle, a_i be the i-th smallest address in $\mathsf{lits}(\mathcal{C})$, $\mathcal{A} \subseteq \mathbb{N}$ such that $\mathsf{lits}(\mathcal{C}) \subseteq \mathcal{A}$, and $\tau : [\max(\mathcal{A})] \to [\max(\mathcal{A})]$ be a permutation such that $\tau(a_i) = i$

[1] Additional applications of simulations under input relabellings are discussed in Appendix C.

and $\tau(\mathcal{A}) = [\,|\mathcal{A}|\,]$. The relationship $\sigma(s) = \mathsf{rename}(\mathcal{C}, \tau, s)$ is a simulation under input relabelling with respect to the family $R = \{\pi_{(s,t)}(p) = \mathsf{rename}(\mathcal{C}, \tau, p) \mid (s,t) \in \sigma\}$, for $\mathsf{global}^{\circ}(\mathcal{C}, \mathcal{A})$ and $\mathsf{global}(\mathsf{norm}(\mathcal{C}), |\mathcal{A}|)$.

Proof. Let $(S, P, f, s_0) = \mathsf{global}^{\circ}(\mathcal{C}, \mathcal{A})$ and $(T, Q, g, t_0) = \mathsf{global}(\mathsf{norm}(\mathcal{C}), |\mathcal{A}|)$. There are three properties to consider.

1. Let $(c_u, \mathbf{u}) = s_0$ and $(c_v, \mathbf{v}) = t_0$. Since $0 \in \mathsf{lits}(\mathcal{C})$, and since 0 is the minimum element in $\mathbb{N}$, then $\tau(0) = 0$. Since all roles are zero-initialized, and since $\tau(0) = 0$, it follows that $\mathsf{rename}(\mathcal{C}, \tau, c_u) = c_v$. Since the τ-renaming of a user configuration preserves address ordering, it follows that $\forall i \in [N] \cdot \mathsf{id}(\mathbf{u}_i) = \mathsf{id}(\mathbf{v})$. Then $\mathsf{rename}(\mathcal{C}, \tau, s_0) = t_0$. Therefore, σ satisfies the first property of simulation under input relabelling.

2. Let $s \in S$. Since $\tau(\mathcal{A}) = [\,|\mathcal{A}|\,]$, then $\sigma(s) \in T$. Then $\forall s \in S \cdot \exists t \in T \cdot (s,t) \in \sigma$. Therefore, σ satisfies the second property of simulation under input relabelling.

3. Let $(s,t) \in \sigma$ and $p \in P$. By construction, there is a single $q \in \pi_{(s,t)}(p)$. Since f and g are deterministic, there is a single $s' \in f(s,p)$ and a single $t' \in g(t,q)$. Since $\mathsf{norm}(\mathcal{C})$ replaces all instances of $a \in \mathsf{lits}(\mathcal{C})$ with $\tau(a)$, and since all other addresses can be rearranged by Theorem 11, it follows that $g(t,q) = \mathsf{rename}(\mathcal{C}, \tau, f(s,p))$. Then, $(s', t') \in \sigma$. Therefore, σ satisfies the third property of simulation under input relabelling with respect to R.

Therefore, σ is a simulation under input relabelling with respect to R. □

Theorem 13. Let $\mathcal{C}$ be a bundle, a_i be the i-th smallest address in $\mathsf{lits}(\mathcal{C})$, $\mathcal{A} \subseteq \mathbb{N}$ such that $\mathsf{lits}(\mathcal{C}) \subseteq \mathcal{A}$, and $\tau : [\max(\mathcal{A})] \to [\max(\mathcal{A})]$ be a permutation such that $\tau(a_i) = i$ and $\tau(\mathcal{A}) = [\,|\mathcal{A}|\,]$. The relationship $\sigma(s) = \mathsf{rename}(\mathcal{C}, \tau, s)$ is a bisimulation under input relabelling with respect to the family $R = \{\pi_{(s,t)}(p) = \mathsf{rename}(\mathcal{C}, \tau, p) \mid (s,t) \in \sigma\}$, for $\mathsf{global}^{\circ}(\mathcal{C}, \mathcal{A})$ and $\mathsf{global}(\mathsf{norm}(\mathcal{C}), |\mathcal{A}|)$.

Proof. Property 1 holds, since σ is one-to-one. Property 2 holds by Lemma 4. For property 3, recall that permutations form a group. Therefore, there exists an inverse permutation τ^{-1} such that $\tau^{-1} \circ \tau$ is the identity permutation. Consequently, $\sigma^{-1}(s) = \mathsf{rename}(\mathcal{C}, \tau^{-1}, s)$ Then a proof that σ is a simulation under input relabelling with respect to R is also a proof that σ^{-1} is a simulation under input relabelling with respect to R. Therefore, σ is a bisimulation under input relabelling with respect to R. □

It is now shown that simulations under input relabellings preserve simulation equivalent properties. As a direct consequence, the τ-rewrite of a MiniSol bundle preserves the τ-renaming of guarded safety property. The proof precedes as follows. In Lemma 5, it is shown simulations under input relabellings preserve simulation-equivalent traces. This is used in the proof of Theorem 14 to show that if there is a trace that violates $P_2 \models \varphi_2$, then there exists a simulation equivalent trace that violates $P_1 \models \varphi_1$. In the case of bisimulations, traces are preserved in both directions, therefore $P_1 \models \varphi_1 \iff P_2 \models \varphi_2$.

Lemma 5. Let $P_1 = (S_1, L_1, f, s_0)$ and $P_2 = (S_2, L_2, g, t_0)$ be LTSs, $\sigma \subseteq S_1 \times S_2$, and $R = \{\pi_{(s,t)} : L_1 \to L_2 \mid (s,t) \in \sigma\}$. If σ is a simulation under input relabelling and $(s_0, \ldots, p_{k-1}, s_k)$ is a trace of P_1, then there exists a trace $(t_0, \ldots, q_{k-1}, t_k)$ of P_2 such that $\forall i \in [k+1] \cdot \sigma(s_i, t_i)$ and $\forall i \in [k] \cdot q_i = \pi_{(s_i,t_i)}(p_i)$.

Proof. Lemma 5 follows by induction on k. In the base case, $(s_0, t_0) \in \sigma$. As an inductive step, assume that Lemma 5 holds up to k. Consider a trace $(s_0, \ldots, p_{k-1}, s_k, p_k, s_{k+1})$ of P_1. By the inductive hypothesis, there exists a trace $(t_0, \ldots, q_{k-1}, t_k)$ of P_2 such that $\forall i \in [k+1] \cdot \sigma(s_i, t_i)$ and $\forall i \in [k] \cdot q_i = \pi_{(s_i,t_i)}(p_i)$. Let $q_k = \pi_{(s_k,t_k)}(p_k)$. By definition, $\exists t' \in g(t_k, q_k) \cdot \sigma(s', t')$. Then, there exists a trace $(t_0, \ldots, q_{k-1}, t_k, q_k, t')$ of P_2 such that $\forall i \in [k+2] \cdot \sigma(s_i, t_i)$ and $\forall i \in [k+1] \cdot q_i = \pi_{(s_i,t_i)}(p_i)$. Therefore, the inductive hypothesis holds. □

Theorem 14. Let $P_1 = (S_1, L_1, f, s_0)$ and $P_2 = (S_2, L_2, g, t_0)$ be LTSs, $\sigma \subseteq S_1 \times S_2$, and φ_1 and φ_2 be simulation equivalent properties with respect to σ. If σ is a simulation under input relabelling, then $P_2 \models \varphi_2 \Rightarrow P_1 \models \varphi_1$. If σ is a bisimulation under input relabelling, then $P_1 \models \varphi_1 \iff P_2 \models \varphi_2$.

Proof. First, consider the case in which σ is a simulation. Assume for contradiction that $P_1 \not\models \varphi_1$. Then there exists a trace $(s_0, \ldots, p_k, s_k)$ of P_1 such that $(s_0, \ldots, p_k, s_k) \not\models \varphi_1$. By Lemma 5, there exists a trace $(t_0, \ldots, p_k, s_k)$ such that $\forall i \in [k+1] \cdot (s_i, t_i) \in \sigma$. Since $P_2 \models \varphi_2$, φ_1 and φ_2 are not simulation equivalent properties with respect to σ. By contradiction, $P_2 \models \varphi_2 \Rightarrow P_1 \models \varphi_1$. Next, consider the case in which σ is a bisimulation. From the first case, $P_2 \models \varphi_2 \Rightarrow P_1 \models \varphi_1$. Since σ is a bisimulation, then σ^{-1} is a simulation. As a result, $P_1 \models \varphi_1 \Rightarrow P_2 \models \varphi_2$. Therefore, $P_1 \models \varphi_1 \iff P_2 \models \varphi_2$. □

5.4 Local Smart Contract Bundles

A local bundle is a finite-state abstraction of a smart contract bundle. At a high level, each local bundle is a non-deterministic LTS that abstracts away the state of certain

users. Formally, a local bundle is constructed from four components: a normalized smart contract, a network size, a concrete address region, and an abstract user domain[2]. The *concrete address region* associates each control state with zero or more users that can maintain their state. The *abstract user domain* assigns an abstract set of states to each user outside of the concrete address region. A local bundle executes transactions according to its normalized smart contract, and the corresponding network size.

Let $\mathcal{C}$ be a smart contract bundle and $N \in \mathbb{N}$. A concrete address region is a relation $\gamma \subseteq \mathsf{control}(\mathcal{C}, [N]) \times [N]$ and an abstract user domain is a relation $\theta \subseteq \mathsf{control}(\mathcal{C}, [N]) \times \mathsf{user}(\mathcal{C}, [N])$. If $(c, i) \in \gamma$, then in control state c, the user at index i is treated concretely by a local bundle. If $(c, i) \notin \gamma$, then the state of user u at index i is abstracted by all $v \in \mathsf{user}(\mathcal{C}, [N])$ such that $(c, v) \in \theta$ and $\mathsf{id}(u) = \mathsf{id}(v)$. A concrete example is given in Example 5.4.1, and a formal definition is given in Definition 5.4.1.

Example 5.4.1. Recall the normalized `Auction` bundle, $\mathcal{C}'$, and user $\mathbf{v}_2'' = (2, \mathsf{map}(\mathbf{v}_5))$, from Example 5.3.1. This example illustrates two potential user abstractions for $\mathbf{v}_2''$. Let $\gamma_\top = \mathsf{control}(\mathcal{C}, [4]) \times [4]$, $\gamma_\bot = \varnothing$, and $\theta_\top = \mathsf{control}(\mathcal{C}, [4]) \times \mathsf{user}(\mathcal{C}, [4])$. Since $\gamma_\top$ associates all control states with all indices, $\gamma_\top$ states that all users are concrete. Conversely, $\gamma_\bot$ does not associate indices with any control states, so $\gamma_\bot$ states that all users are abstract. The abstraction defined by $(\gamma_\top, \theta_\top)$ sends $\mathbf{v}_2''$ to itself, since all users are concrete. The abstraction defined by $(\gamma_\bot, \theta_\top)$ sends $\mathbf{v}_2''$ to any $(2, (v))$ such that $v \in \mathbb{D}$, since all users are abstract, and all possible mapping entries for user 2 are in the abstract domain $\theta_\top$. □

Definition 5.4.1 (User Abstraction[3])**.** Let $\mathcal{C}$ be a bundle, $N \in \mathbb{N}$, $S_C = \mathsf{control}(\mathcal{C}, [N])$, $S_U = \mathsf{user}(\mathcal{C}, [N])$, $\gamma \subseteq S_C \times [N]$, and $\theta \subseteq S_C \times S_U$. The *user abstraction* defined by (γ, θ) is a relation $g : (S_U) \to 2^{(S_U)}$ such that:

$$g(c, u) = \begin{cases} \{v \in S_U \mid \mathsf{id}(u) = \mathsf{id}(v) \wedge (c, v) \in \theta\} & \text{if } (c, \mathsf{id}(u)) \notin \gamma \\ \{u\} & \text{otherwise} \end{cases}$$

There are many choices of (γ, θ), although not all combinations are interesting. Interesting applications are provided in Chapter 6 and Chapter 7. To illustrate γ and θ, several useful examples are defined and their interpretations are given:

1. In $\gamma_\top := \mathsf{control}(\mathcal{C}, [N]) \times [N]$, all N users are concrete;

[2]This presentation diverges from [72]. In the [72], a non-consecutive neighbourhood $\mathcal{A} \subseteq \mathbb{N}$ was used in place of a network size. Due to Theorem 13, it is assumed, without loss of generality, that $\mathcal{A} = [N]$. Furthermore, [72] does not include a concrete address region. The concrete address region is optional, and is used to refine the balance relation in Chapter 6, and to extend local bundles to fuzzing in Chapter 7.

[3]An extension to k-dimensional maps is outlined in Appendix A.

2. In $\gamma_\bot := \varnothing$, all N users are abstract;
3. In $\gamma_{\text{Imp}} := \{(c, i) \in \gamma_\top \mid i \in \mathsf{Iits}(\mathcal{C})\}$, all implicit participants are concrete;
4. In $\gamma_{\text{Role}} := \{(c, i) \in \gamma_\top \mid \exists j \in \mathbb{N} \cdot \mathsf{role}(c, j) = i\}$, all transient participants concrete;
5. The weakest abstraction is $\theta_\top := \mathsf{control}(\mathcal{C}, [N]) \times \mathsf{user}(\mathcal{C}, [N])$;
6. The strongest abstraction in $\theta_\top := \varnothing$.

The local bundle corresponding to $(\mathcal{C}, N, \gamma, \theta)$ is defined using a special relation called an *N-user interference relation*. Intuitively, the N-user interference relation for (γ, θ) sends an N-user smart contract state to the set of all N-user smart contract states reachable under the user abstraction of (γ, θ). A state is reachable under the user abstraction of (γ, θ) if the control state is unchanged, each address is unchanged, and all user data satisfies the user abstraction. A concrete example is given in Example 5.4.2, and a formal definition is given in Definition 5.4.2. Note that if the interference relation for (γ, θ) fails to relate $(c, \mathbf{u})$ to itself, then $(c, \mathbf{u})$ violates (γ, θ).

Example 5.4.2. Recall the normalized `Auction` bundle, $\mathcal{C}'$, and user configuration $\mathbf{v}''$, from Example 5.3.1 In Example 5.4.1, the user abstractions $(\gamma_\top, \theta_\top)$ and $(\gamma_\bot, \theta_\top)$ were applied to the user $\mathbf{v}''_2$. This example extends $(\gamma_\bot, \theta_\top)$ to all users of $\mathbf{v}''$ via an interference relation, and then illustrates an interference violation with the user abstraction $(\gamma_\bot, \theta_\bot)$. Recall that $(\gamma_\bot, \theta_\top)$ treats all users abstractly, and allows for all possible mapping states. Therefore, the interference of $(\gamma_\bot, \theta_\top)$ maps $\mathbf{v}''$ to $A_1 = \{\mathbf{u} \in \mathsf{user}(\mathcal{C}', [4]) \mid \forall i \in [4] \cdot \mathsf{id}(\mathbf{u}_i) = i\}$. Clearly, $\mathbf{v}''$ satisfies the interference of $(\gamma_\bot, \theta_\top)$, since $\mathbf{v}'' \in A_1$. Next, consider the interference of $(\gamma_\bot, \theta_\bot)$. Since $\theta_\bot = \varnothing$, then the interference of $(\gamma_\bot, \theta_\bot)$ maps $\mathbf{v}''$ to $A_2 = \varnothing$. Clearly, $\mathbf{v}''$ violates the interference of $(\gamma_\bot, \theta_\top)$, since $\mathbf{v}'' \notin A_2$. □

Definition 5.4.2 (Interference)**.** Let $\mathcal{C}$ be a bundle, $N \in \mathbb{N}$, $(S, P, f, s_0) = \mathsf{global}(\mathcal{C}, N)$, and g be a user abstraction. The *N-user interference relation* for g is a relation $h : S \to 2^S$ such that $h(c, \mathbf{u}) = \{(c, \mathbf{v}) \in S \mid \forall i \in [N] \cdot \mathbf{v}_i \in g(c, \mathbf{u}_i)\}$.

Each state of the *local bundle* for $(\mathcal{C}, N, \gamma, \theta)$ is a tuple $(c, \mathbf{u})$ where c is the control state and $\mathbf{u}$ is an N-user configuration. The transition relation of the local bundle is defined in terms of the (global) transition function f for $\mathsf{norm}(\mathcal{C})$. First, the transition relation applies f. If the application of f is closed under the user abstraction of (γ, θ), then the user abstraction is applied. Otherwise, the state remains unchanged. Intuitively, the user abstraction defines a safe envelop under which the abstraction is compositional (see Chapter 6). Note that since local bundles operate on normalized smart contracts, they require normalized properties.

Definition 5.4.3 (Local Bundle)**.** Let $\mathcal{C}$ be a bundle, $N \in \mathbb{N}$, γ be a concrete region for $(\mathcal{C}, N)$, θ be a user abstraction for $(\mathcal{C}, N)$, h be the N-user interference relation for the user abstraction defined by (γ, θ), and $(S, P, f, s_0) = \mathsf{global}(\mathsf{norm}(\mathcal{C}), N)$. A local bundle is an LTS $\mathsf{local}(\mathcal{C}, N, \gamma, \theta) := (S, P, \hat{f}, s_0)$ such that $\hat{f} : S \times P \to 2^S$ defined by:

$$\hat{f}((c, \mathbf{u}), p) := \begin{cases} h\left(f_p(c, \mathbf{u})\right) & \text{if } f_p(c, \mathbf{u}) \in h\left(f_p(c, \mathbf{u})\right) \\ \{f_p(c, \mathbf{u})\} & \text{otherwise} \end{cases}$$

Definition 5.4.4 (Normalized Property)**.** Let $\mathcal{C}$ be a bundle, φ be a guarded safety property with literal guards L, a_i be i-th largest address in $\mathsf{lits}(\mathcal{C})$, and b_j be the j-th largest address in $L \backslash \mathsf{lits}(\mathcal{C})$. The $\mathcal{C}$-normalization of φ, written $\mathsf{norm}(\mathcal{C}, \varphi)$, is the guard renaming of φ that maps a_i to i and b_i to $|\mathsf{lits}(\mathcal{C})| + j$.

Example 5.4.3. This example briefly illustrates the transition relation of Definition 5.4.3 using `Auction` of Fig. 3.1. For simplicity of presentation, the local bundle abstraction is assumed to have a network size of 4. All users implicit and transient users are treated concretely, and the abstract user domain is given by **Prop. 7**: "*The zero-account never has an active bid, while all other users can have active bids.*" Formally, $(S, P, \hat{f}, s_0) = \mathsf{local}\left(\mathcal{C}, 4, \gamma_1, \theta_1\right)$ where $\gamma_1 = (\gamma_{\text{Role}} \cup \gamma_{\text{Imp}})$ and θ_1 is defined by:

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

Let g be the 4-user interference relation defined by (γ_1, θ_1). Consider applying $\hat{f}$ to $(c, \mathbf{u}) \in S$ with action $p \in P$, such that $c = ((2), (0, 0, 0, 0, \bot))$, $\forall i \in [4] \cdot \mathsf{map}(\mathbf{u}_i) = \mathbf{0}$, and $p = (\texttt{bid}, (3), (10))$. That is, the abstract user at address 3 places a bid of 10 into an ongoing auction with a manager at address 2 and leading bid of 0. By γ_1, the users at addresses 0 through to 2 are concrete, as 0 is the zero-account, 1 is the smart contract account, and 2 is the manager.

Action p must succeed, since the sender is not the manager, and the leading bid is less than 10. It follows that if $(s', \mathbf{v}) = f_p(s, \mathbf{u})$, then the leading bid is updated to 10, and the bid of the sender is updated to 10. Furthermore, since the bid of the zero-account is unchanged by f_p, then $(s', \mathbf{v}) \in g(s', \mathbf{v})$, and therefore, $g(s', \mathbf{v}) = \hat{f}\left((s, \mathbf{u}), p\right)$. A successor state is selected from $g(s', \mathbf{v})$ as follows. By γ_1, the bids of user 0 through to user 3 are unchanged. By θ_1, the single abstract user at address 3 must be assigned an arbitrary value that satisfies θ_1. In this example, 100 is selected, since $100 \geq 0$. This yields a user configuration such that $\mathbf{u}' = ((0, 0), (1, 0), (2, 0), (3, 100))$.

In conclusion, a successor state to $(s, \mathbf{u})$ is $(s', \mathbf{u})$ such that the leading bid is updated to 10 and the bid of user at address 3 is set to 100. Note that $(s', \mathbf{u}')$ is not reachable in $\mathsf{global}(\mathcal{C}, 4)$, as the bid of user 3 exceeds the leading bid. □

5.5 Summary

This section has identified a set of local symmetries among MiniSol transactions. In Theorem 8, a notion of locality was defined that reduces each transaction to its participating users. This locality is useful, as it is sufficient to determine the outcome of the transaction. In Theorem 11 and Theorem 13, it is show that all local transactions are symmetric to a set of abstract transactions from a finite domain of consecutive addresses. In Section 5.4, these insights were used to define the finite-state *local bundle abstraction* for a MiniSol smart contract. The section concluded with an example (Example 5.4.3) that walked thought an execution of a local bundle. In Chapter 6 and Chapter 7, results are proven about local bundles, with applications to PCMC and bounded analysis.

Chapter 6

Parameterized Composition Reasoning for Local Smart Contracts

This chapter presents an application of local transaction analysis to the parameterized verification of MiniSol smart contracts. At a high-level, this section reduces parameterized smart contract verification to safety of a local bundle abstraction, through PCMC. Section 6.1 proves that local transaction symmetries form a balance relation. This means that MiniSol smart contracts are amenable to PCMC. Section 6.2 extends this result by showing the balance relation preserve certain aggregate properties. As a main result, Section 6.3 provides sound proof rules for compositionality and parameterized k-safety through the verification of local bundles. This reduction is novel compared to prior work in PCMC, as a compositional cutoff is never used explicitly.

6.1 A Smart Contract Balance Relation

This section relates local transaction symmetries to PCMC. First, it is proven that local transaction symmetries form balance relations for MiniSol bundles. It is shown that these balance relations respect guarded k-safety properties, even though guarded k-safety properties might induce new equivalence classes. The new equivalence classes are characterized, and use to motivate a syntactic form for interference invariants called *split interference invariants*.

Recall from Section 5.2, the local transaction symmetry relation $(\sim_D) \cap (\sim_N)$. In Theorem 15 it is shown that local transaction symmetries provide a balance relation for

MiniSol bundles. It then follows by Theorem 11 that this balance relation has finitely many equivalence classes.

Theorem 15. The relation $(\sim_D) \cap (\sim_N)$ is a balance relation with witnessing bijection defined by the corresponding bijection for $(\sim_N)$.

Proof. Assume that $(s_u, p_u, \lambda_u) \sim_D (s_v, p_v, \lambda_v)$ and $(s_u, p_u, \lambda_u) \sim_N (s_v, p_v, \lambda_v)$ with corresponding bijection β. By Lemma 2 and Lemma 3, both $\sim_D$ and $\sim_N$ are equivalence relations. In must be shown that $((s_u, p_u, \lambda_u), \beta, (s_v, p_v, \lambda_v))$ satisfy the properties of a balance relation.

1. Since $\sim_D$ is an equivalence relation, then $(s_v, p_v, \lambda_v) \sim_D (s_u, p_u, \lambda_u)$. Since $\sim_N$ is an equivalence relation, then $(s_v, p_v, \lambda_v) \sim_N (\lambda_u, p_u, s_u)$ with witnessing bijection β^{-1}. Therefore, $((s_v, p_v, \lambda_v), \beta^{-1}, (s_u, p_u, \lambda_u))$ is also in the relation, as desired.

2. By definition of $(\sim_D) \cap (\sim_N)$, a pair of control states are locally symmetric if and only if their users are locally symmetric. These symmetries preserve participation classes by definition of $\sim_N$, and process state by $\sim_D$. Furthermore, by the network semantics of MiniSol, a control process is adjacent to many user processes, and a user process is adjacent to a single control process. This means that a bijection between control processes (resp. user processes) ensures the existence of a bijection between adjacent user processes (resp. control processes).

Therefore, $(\sim_D) \cap (\sim_N)$ is a balance relation with witnessing bijection defined by the corresponding bijection for $(\sim_N)$. □

It is proven in Theorem 16 that the balance relation in Theorem 15 respects guarded k-universal safety properties. However, a guarded k-universal safety property might induce new equivalence classes. For example, k-universal quantification requires at least k explicit representatives to check that it is satisfied (else the property can be vacuously true). Similarly, each role guard requires a transient participant, and each literal guard requires a literal participant (if they are not already in the PTG of the bundle). Examples of how each case can go wrong are given in Example 6.1.1. These problems are resolved in Section 6.3 via *saturating network sizes*. To simplify presentation until Section 6.3, Theorem 16 assumes a sufficiently large PTG.

Example 6.1.1. This example considers three properties for `Auction` of Fig. 3.1 that have clear violation:

Prop. 8 : *For any pair of users, only one user has an active bid.*

Prop. 9 : *The manager always has a bid of* 10*.*

Prop. 10 : *The user with address* 100 *cannot bid.*

If there is only one explicit participant, then all other participants may not bid, and therefore, **Prop. 8** is satisfied. To find a counterexample to **Prop. 8**, there must be at least two explicit participants. If the manager is not a transient participant, then the guard of **Prop. 9** is never satisfied, and therefore, **Prop. 9** is vacuously true. To find a counterexample to **Prop. 9**, there must be a transient participant. If the user with address 100 is not a participant, then **Prop. 10** is vacuously true. To find a counterexample to **Prop. 10**, address 100 must be treated as a literal address value. □

Theorem 16. Let φ be a guarded k-universal safety property, and $\mathcal{C}$ be a bundle with an implicit participant for each role guard of φ. The relation $(\sim_D) \cap (\sim_N)$ for $\mathcal{C}$ respects φ.

Proof. Assume that $((c_u, \mathbf{u}), p_u, \lambda_u) \sim_D (c_v, \mathbf{v}), p_v, \lambda_v)$, $((c_u, \mathbf{u}), p_u, \lambda_u) \sim_N ((c_v, \mathbf{v}), p_u, \lambda_u)$ with corresponding permutation τ, and $(c_u, \mathbf{u}) \models \varphi$. Let L be the literal guards of φ. Since there is an implicit participant in $\mathcal{C}$ for each role guard of φ, then $\forall (i, a) \in L \cdot \tau(a) = a$ by definition of $(\sim_N)$. Then by Theorem 12, $(c_v, \mathbf{v}) \models \varphi$. □

Similar to safety properties in PCMC, a valid interference invariant must also respect the network topology of a bundle. This means that an interference invariant might depend on user addresses, but only if these dependencies are respected by the balance relation. To rule out interference invariants that violate the balance relation, a syntactic form called a *split interference invariant* is introduced. A split interference invariant is built from a list of 1-universal predicates (recall Section 3.2), each guarded by a single constraint. The final predicate is guarded by the negation of all other constraints. Intuitively, each 1-universal predicate summarizes the class of users that satisfy its guard. The split interference invariant is the conjunction of all (guarded predicate) clauses. A formal definition is given in Definition 6.1.1 and a practical illustration is given in Example 6.1.2. Note that since a split compositional invariant is syntactic, rather than semantic, the term *candidate split compositional invariant* is used to describe a predicate that fits the syntactic definition in Definition 6.1.1, regardless of whether is satisfies the semantic definition in Section 2.7.

Definition 6.1.1 (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 1-universal properties, μ is a list of n 1-universal properties, and ξ is a 1-universal property, such that:

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

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

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

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

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

Example 6.1.2. The abstract user domain, θ_1, defined in Example 5.4.3 is an example of a split interference invariant. Recall that informally, θ_1 is defined by **Prop. 7**: "*The zero-account never has an active bid, while all other users can have active bids.*" Formally, θ_1 is defined by:

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

Following Definition 6.1.1, θ_1 is determined by $\mathit{Inv} = (\mathcal{A}_L, \varnothing, (\xi_1), \varnothing, \xi_2)$, where $\mathcal{A}_L = \{0\}$, $\xi_1 := (\mathsf{map}(\mathbf{u}_0)_0 = 0)$, 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 Definition 6.1.1, 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 $\psi_{\text{Else}}(s, \mathbf{u}) := (\mathsf{id}(\mathbf{u}_0) \neq 0) \Rightarrow (\mathsf{map}(\mathbf{u}_0)_0 \geq 0)$. □

6.2 Local Symmetries for Aggregate Properties

This section extends smart contract PCMC to aggregate properties. The results in Theorem 16 are limited to guarded k-universal properties. However, certain aggregate properties are also preserved by the balance relation in Theorem 15. This section prove that reachable assignments for point-wise aggregates are over-approximated by smart contract PCMC. Furthermore, it is shown that for many interesting aggregates, such as sums and monotonic extrema, the analysis is exact.

The local analysis of point-wise aggregates proceeds in two steps. First, a source code transformation is presented to maintain point-wise aggregate approximations. Second, it

```
1  contract Auction {
2    mapping(address => uint) bids;
3    uint sum = 0;
4    // ...
5    function bid() payable {
6      uint pre = bids[msg.sender];
7
8      require(!stopped);
9      require(msg.sender != manager);
10
11     uint oldBid = bids[msg.sender];
12     uint newBid = oldBid + msg.sender;
13     require(oldBid == 0 || oldBid < leadingBid);
14
15     bids[msg.sender] = newBid;
16     leadingBid = newBid;
17
18     uint post = bids[msg.sener];
19     sum = sum - pre + post;
20   }
21   // ...
22 }
```

(a) Instrumented code to compute sum(`bids`).

```
1  contract Auction {
2    mapping(address => uint) bids;
3    uint max = 0;
4    bool monotonic = true;
5    // ...
6    function bid() payable {
7      uint pre = bids[msg.sender];
8
9      require(!stopped);
10     require(msg.sender != manager);
11
12     uint oldBid = bids[msg.sender];
13     uint newBid = oldBid + msg.sender;
14     require(oldBid == 0 || oldBid < leadingBid);
15
16     bids[msg.sender] = newBid;
17     leadingBid = newBid;
18
19     uint post = bids[msg.sener];
20     if (pre == max && post < max) { monotonic = false; }
21     else if (post >= max) { max = post; }
22   }
23   // ...
24 }
```

(b) Instrumented code to compute max(`bids`).

Figure 6.1: Local instrumentation of sum(`bids`) and max(`bids`) for `bid()` of Fig. 3.1. All modifications are highlighted.

is shown that these approximations can be used to verify aggregate properties exclusively through control data. Since local transactions preserve control data, all results extend to smart contract PCMC. As in previous sections, examples are given with respect to `Auction` of Fig. 3.1.

Let $\mathcal{C}$ be a bundle and f be the point-wise aggregate defined by (g, h, z). Recall from Section 3.2, that g sends each mapping entry to an output domain D, h is a binary operator for D used to aggregate the elements, and $z \in D$ is the base case for the aggregation. Schemes were given in Section 3.2 to locally approximate point-wise aggregates based on Abelian groups and lattices. These schemes motivate a way to instrument $\mathcal{C}$ with aggregate approximations.

In the case of aggregates based on Abelian groups, a single variable is added to the bundle to track the sum of the mapping (this variable must have the same domain as the Abelian group, and may require an arithmetic integer, as opposed to a 256-bit integer). The sum variable is initially set to z. For each write to a mapping, `map[i] = x`, the sum is updated by first adding the inverse of `map[i]` to sum, and then adding `x` to sum. An example is given for sum(`bids`) in Fig. 6.1a. In this example, `sum` is added to `Auction` to track sum(`bids`). At line 3, `sum` is set to 0, since $z = 0$ in the case of sum$(\cdot)$. An update to `sum` is shown at lines 18–19.

In the case of aggregates based on lattices, two variables are added to the bundle: the

first variable tracks the supremum, and the second variable is true so long as all writes are monotonically increasing, relative to the lattice. The supremum variable is initially set to $\sup\{z, g(\mathbf{0})\}$, and the monotonic variable is initially set to `true` (a singleton sequence is monotonically increasing). For each write to a mapping, `map[i] = x`, the supremum is updated by taking its supremum with `x`. Furthermore, if the change from `map[i]` to `x` was decreasing, then the monotonic variable is set to `false`. An example is given for max(`bids`) in Fig. 6.1b. In this example, `max` and `monotonic` are added to `Auction` to track max(`bids`) and monotonicity. At line 3, `max` is set to 0, since $z = 0$ in the case of $\max(\cdot)$. Updates to `sum` and `monotonic` for `bid()` are shown at lines 19–21.

By performing this instrumentation, an aggregate for $\mathcal{C}$ can be approximated without inspecting an entire mapping. In particular, an aggregate can be extracted from a local transaction without knowing the global state of the network. The local aggregate can then appear within any guarded k-universal safety property, as already supported by smart contract PCMC. Theorem 17 proves that a local aggregate based on an Abelian group is equal to the global sum. Theorem 18 proves that a local aggregate based on a lattice over-approximates the global supremum, and in the case that each update is monotonic, equals the global supremum.

Theorem 17. Let $\mathcal{C}$ be a bundle instrumented with point-wise aggregate (g, h, z) at datum i, such that $g : \mathbb{D} \to D$, (D, h) is an Abelian group with identity $g(0)$. For any $N \in \mathbb{N}$, if $(c, \mathbf{u})$ is reachable in $\mathsf{global}(\mathcal{C}, N)$, then $\mathsf{data}(c, i) = \mathsf{fold}(h, z, g(\mathsf{map}(\mathbf{u})))$.

Proof. For simplicity of presentation, let $x \circ y = f(x, y)$. Fix an $N \in \mathbb{N}$. It follows by induction on the length of a trace to $(c, \mathbf{u})$, that Theorem 17 holds.

1. In the base case, a trace has length 0. The only state reachable in 0 transitions is $(c, \mathbf{u}) = s_0$. By definition, $\mathsf{data}(c, i) = z$ and $\mathbf{u}$ is zero-initialized. Therefore, $\mathsf{fold}(h, z, g(\mathsf{map}(\mathbf{u}))) = g(0) \circ \cdots \circ g(0) \circ z$. Since $g(0)$ is the identity of $(D, \circ)$, then $\mathsf{fold}(h, z, g(\mathsf{map}(\mathbf{u}))) = z$. Therefore, $\mathsf{data}(c, i) = \mathsf{fold}(h, z, g(\mathsf{map}(\mathbf{u})))$ for all states reachable in a length 0 trace.

2. Assume that $(c, \mathbf{u})$ is reachable in a length k trace, and that $(c', \mathbf{u}')$ is a successor to $(c, \mathbf{u})$. By the inductive hypothesis, $\mathsf{data}(c, i) = \mathsf{fold}(h, z, g(\mathsf{map}(\mathbf{u})))$. By instrumentation, $\mathsf{data}(c', i) = \mathsf{data}(c, i) \circ g^{-1}(\mathsf{map}(\mathbf{u}_0)) \circ g((\mathbf{u}_0)) \circ \cdots \circ g^{-1}(\mathsf{map}(\mathbf{u}_{N-1})) \circ g(\mathsf{map}(\mathbf{u}_{N-1}))$. By Theorem 3, $\mathsf{data}(c', i) = \mathsf{fold}(h, z, g(\mathsf{map}(\mathbf{u})))$.

By induction, Theorem 17 is established. □

Lemma 6. Let $\mathcal{C}$ be a bundle instrumented with point-wise aggregate (g, h, z) at datum j (`monotonic`), such that $g : \mathbb{D} \to D$ and $(D, \leq)$ is a lattice with supremum h. For any $N \in \mathbb{N}$, if $\left((c_0, \mathbf{u}^0), \ldots, \left(c_k, \mathbf{u}^k\right)\right)$ is a trace of $\mathsf{global}(\mathcal{C}, N)$ and $\mathsf{data}(c_k, j) = \top$, then for all $i \in [N]$, $\left(g\left(\mathsf{map}\left(\mathbf{u}_i^0\right)\right), \ldots, g\left(\mathsf{map}\left(\mathbf{u}_i^k\right)\right)\right)$ is monotonically increasing relative to $\leq$.

Proof. For simplicity of presentation, let $x \circ y = f(x, y)$. Fix an $N \in \mathbb{N}$. It follows by induction on the length of a trace, that Lemma 6 holds.

1. In the base case, a trace has length 0. The only length 0 trace is (s_0) where $(c, \mathbf{u}) = s_0$. It is vacuously true that for any $i \in [N]$, $(g\left(\mathsf{map}\left(\mathbf{u}_i\right)\right))$ is monotonically increasing relative to $\leq$. As desired, $\mathsf{data}(c, j) = \top$ in the initial state.

2. Assume that $\left((c_0, \mathbf{u}^0), \ldots, \left(c_k, \mathbf{u}^k\right)\right)$ is a trace of $\mathsf{global}(\mathcal{C}, N)$, $\mathsf{data}(c_k, j) = \top$, and the inductive hypothesis holds up to k. Assume that $(c', \mathbf{u}')$ is a successor state of $\left(c_k, \mathbf{u}^k\right)$ such that for some $i \in [N]$, $\left(g\left(\mathsf{map}\left(\mathbf{u}_i^0\right)\right), \ldots, g\left(\mathsf{map}\left(\mathbf{u}_i^k\right)\right), g\left(\mathsf{map}\left(\mathbf{u}_i'\right)\right)\right)$ is not monotonically increasing relative to $\leq$. By hypothesis, the sequence is monotonically increasing up to $g\left(\mathsf{map}\left(\mathbf{u}_i^k\right)\right)$, and therefore $g\left(\mathsf{map}\left(\mathbf{u}_i'\right)\right) < g\left(\mathsf{map}\left(\mathbf{u}_i^k\right)\right)$. Then by instrumentation of $\mathcal{C}$, it follows that $\mathsf{data}(c', j) = \bot$. Through proof by contrapositive, the inductive hypothesis holds.

By induction, Lemma 6 is established. □

Theorem 18. Let $\mathcal{C}$ be a bundle instrumented with point-wise aggregate (g, h, z) at data i (`max`) and j (`monotonic`), such that $g : \mathbb{D} \to D$ and $(D, \leq)$ is a lattice with supremum h. For any $N \in \mathbb{N}$, if $(c, \mathbf{u})$ is reachable in $\mathsf{global}(\mathcal{C}, N)$, then $\mathsf{data}(c, i) \leq \mathsf{fold}(h, z, \mathsf{map}(\mathbf{u}))$. Furthermore, if $\mathsf{data}(c, j) = \top$, then the equality is strict.

Proof. For simplicity of presentation, let $\sup\{x, y\} = h$. Fix an $N \in \mathbb{N}$. It follows by induction on the length of a trace to $(c, \mathbf{u})$, that Theorem 18 holds.

1. In the base case, a trace has length 0. The only state reachable in 0 transitions is $(c, \mathbf{u}) = s_0$. By definition, $\mathsf{data}(c, i) = \sup\{z, g(\mathbf{0})\}$, $\mathsf{data}(c, j) = \top$, and $\mathbf{u}$ is zero-initialized. Then $\mathsf{fold}(h, z, g(\mathsf{map}(\mathbf{u}))) = \max\{z, g(\mathbf{0}), \ldots, g(\mathbf{0})\} = \sup\{z, g(\mathbf{0})\}$. Therefore, $\mathsf{fold}(h, z, g(\mathsf{map}(\mathbf{u}))) = \mathsf{data}(c, i)$ and $\mathsf{data}(c, j) = \top$, as desired.

2. Assume that $(c, \mathbf{u})$ is reachable in a length k trace, and that $(c', \mathbf{u}')$ is a successor to $(c, \mathbf{u})$. By the inductive hypothesis, $\mathsf{data}(c, i) \leq \mathsf{fold}(h, z, g(\mathsf{map}(\mathbf{u})))$, and if $\mathsf{data}(c, j) = \top$, then the equality is strict. By instrumentation, $\mathsf{data}(c', i) = \sup\{\mathsf{data}(c, i), g((\mathbf{u}_0')), \ldots, g((\mathbf{u}_N'))\}$. First, assume that $\mathsf{data}(c, j) = \bot$. Then by

Theorem 4, $\mathsf{data}(c', i) \leq \mathsf{fold}(h, z, g(\mathsf{map}(\mathbf{u})))$. Next, assume that $\mathsf{data}(c', j) = \top$. By Lemma 6, $\mathsf{data}(c, j) = \top$ and all updates from $\mathbf{u}$ to $\mathbf{u}'$ were monotonically increasing relative to $\leq$. Then by Theorem 4, $\mathsf{data}(c', i) = \mathsf{fold}(h, z, g(\mathsf{map}(\mathbf{u})))$.

By induction, Theorem 18 is established. □

6.3 Reduction to Software Model Checking

Prior work on fully-automated PCMC (e.g., [52, 54]) discovers compositional invariants by first finding a cutoff size for the network, and then analyzing all networks up to the given cutoff[1]. However, for MiniSol bundles, the notion of a cutoff is less clear. For example, the maximal sum of all bids in Fig. 3.1 increases as the number of users in the network increases, and therefore, there is no clear cutoff point. In this section, a new approach to PCMC is given though local bundle abstraction. Intuitively, a local bundle is used to simulate an arbitrary neighbourhood under the interference of all other users. This means that the abstract user domain is the interference invariant. The concrete address region is used to refine the abstraction, and can be selected anywhere from $\gamma_\perp$ (i.e., all users are abstract) through to $(\gamma_{\text{Imp}} \cup \gamma_{\text{Role}})$ (i.e., transient and implicit participants are concrete). An inductive invariant for a local bundle, as obtained by any off-the-shelf model checker, then provides an inductive invariant for the SCUN.

The first step in this reduction is to determine the number of users in the local bundle. From Theorem 11 and Theorem 13, it can be assumed, without loss of generality, that all addresses are consecutive. This means that only the size of the local transaction is necessary. This is called a *saturating network size*, as it is large enough to fill all participation classes within a transaction. For example, the network size in Example 5.4.3 was saturating by design.

Rather than analyze a bundle manually, the saturating network size can be extracted automatically from a PTG. Formally, a network size is saturating if its corresponding local bundle contains a representative from each participation class of a PTG, and for all role guards ($\mathcal{A}_R \subseteq \mathbb{N}$) and literal guards ($\mathcal{A}_L \subseteq \mathbb{N}$) of interest. In the case of `PTGBuilder`, this requires counting all explicit, transient, and implicit labels in the PTG.

Definition 6.3.1 (Saturating Network Size)**.** Let $\mathcal{A}_R, \mathcal{A}_L \subseteq \mathbb{N}$ be finite, $\mathcal{C}$ be a bundle, (G, μ, η) be the `PTGBuilder` PTG of $\mathcal{C}$, and $G = (V, E, \delta)$. A *saturating network size* for $(\mathcal{A}_R, \mathcal{A}_L, (G, \mu, \eta))$ is a number $(N_{\text{Exp.}} + N_{\text{Trans.}} + N_{\text{Impl.}})$ such that:

[1] [53] and [5] provide alternative solutions. However, [53] is manual and [5] requires an oracle.

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

The next step in this reduction is to select a concrete address region. A key observation is at any point during execution, there is a single choice for each transient and implicit participant. There is a single choice for each transient participant, as a smart contract assigns each role to a single user at any one time. There is a single choice for each implicit participant, as an implicit participant is identified by a fixed address. However, a purely explicit participant can be exchanged with any other purely explicit participant, without changing the control state (as proven in Theorem 10). This topological argument shows that all transient and implicit participants can be treated as concrete, whereas all other participants are abstract. In other words, the strongest possible concrete address region for smart contract PCMC is $(\gamma_{\text{Role}} \cup \gamma_{\text{Imp}})$, and the weakest possible region is $\gamma_\bot$.

The correct choice of concrete address region and network size reduces compositionality and k-safety proofs to the safety of local bundles. As stated before, the concrete address region must respect the network topology of a bundle $\mathcal{C}$. The network size depends on whether the proof is for compositionality or k-safety. For the compositionality of θ_U, consider a local bundle with $(N+1)$ users, where N is the saturating network size for $\mathcal{C}$ and the guards of θ_U. Then the local bundle contains a representative for: each participation class of $\mathcal{C}$; each role and literal guard distinguished by θ_U; an arbitrary user under interference. Intuitively, the local bundle computes the reachable control states (i.e., an inductive invariant) relative to interference invariant θ_U. The details are given in the proof of Theorem 19.

Theorem 19. Let $\mathcal{C}$ be a bundle, G be a PTG for $\mathcal{C}$, θ_U be a candidate split interference invariant with role guards $\mathcal{A}_R$ and literal guards $\mathcal{A}_L \in \mathsf{lits}(\mathcal{C})$, and N be a saturating neighbourhood for $(\mathcal{A}_R, \mathcal{A}_L, G)$. Then, $\mathsf{local}\,(\mathcal{C}, N+1, (\gamma_{\text{Role}} \cup \gamma_{\text{Imp}})\,, \theta_U) \models \theta_U$ if and only if θ_U is an interference invariant for $\mathcal{C}$.

Proof. Recall **Initialization**, **Consecution**, and **Interference** from Section 2.7. Let $(S, P, \hat{f}, s_0) = \mathsf{local}\,(\mathcal{C}, N+1, (\gamma_{\text{Role}} \cup \gamma_{\text{Imp}})\,, \theta_U)$. The user configurations in S each have a representative for each equivalence class in $\mathcal{C}$, in addition to an arbitrary representative under interference. Therefore, every transition of $(S, P, \hat{f}, s_0)$ captures both consecution and interference. The proof of Theorem 19 proceeds in two cases:

(⇒ Case) Assume that θ_U is compositional. Then θ_U satisfies **Initialization**, **Consecution**, and **Interference**. It follows by induction on the length of a trace, that $(S, P, \hat{f}, s_0) \models \theta_U$. In the base case, there is a single length 1 trace, given by (s_0). By **Initialization**, $s_0 \models \theta_U$. As an inductive step, assume that all length i traces satisfy θ_U. Let $(c, \mathbf{u})$ be the i-th state of an arbitrary trace. By hypothesis, $(c, \mathbf{u}) \models \theta_U$. As θ_U is closed under **Consecution** and **Interference**, and as $(\gamma_{\text{Role}} \cup \gamma_{\text{Imp}})$ respects the network topology of $\mathcal{C}$, it must be the case that $\forall p \in P \cdot \forall (c', \mathbf{u}') \in \hat{f}_p(c, \mathbf{u}) \cdot (c', \mathbf{u}') \models \theta_U$. Therefore, the inductive step holds, and all traces of $(S, P, \hat{f}, s_0)$ satisfy θ_U. In conclusion, $(S, P, \hat{f}, s_0) \models \theta_U$.

(⇐ Case) Assume that $(S, P, \hat{f}, s_0) \models \theta_U$. Then $s_0 \models \theta_U$, and **Initialization** holds. By construction, $(S, P, \hat{f}, s_0)$ computes the reachable states of the control process, relative to $(\gamma_{\text{Role}} \cup \gamma_{\text{Imp}})$ and θ_U. Since $(\gamma_{\text{Role}} \cup \gamma_{\text{Imp}})$ respects the network topology of $\mathcal{C}$, then $(S, P, \hat{f}, s_0)$ computes an inductive invariant for the control processes relative to θ_U. Assume that $(c, \mathbf{u})$ is reachable in $(S, P, \hat{f}, s_0)$. Then the control process, and all concrete users satisfy the inductive invariant, whereas all abstract users satisfy θ_U. By definition, $\forall p \in P \cdot [\![\mathsf{norm}(\mathcal{C})]\!]_{\mathcal{M}}((c, \mathbf{u}), p) \in \hat{f}_p(c, \mathbf{u})$. Then by assumption, $[\![\mathsf{norm}(\mathcal{C})]\!]_{\mathcal{M}}((c, \mathbf{u}), p) \models \theta_U$. As a result, θ_U satisfies **Consecution** (for the participating abstract users) and **Interference** (for the abstract users under interference). Therefore, θ_U is an interference invariant.

Therefore, both directions of Theorem 19 hold. □

Now assume that θ_U is compositional, and that φ is a guarded k-safety property. As in Theorem 19, Theorem 20 also uses a saturating network size that incorporates the guards of φ. Since this proof rule proves inductiveness, rather than compositionality, the local bundle no longer requires an arbitrary user under interference. However, a k-universal property can distinguish between k users at once, rather than one. Thus, the network size must allow for k arbitrary representatives, rather than one[2]. Full details are given in the proof of Theorem 20.

Theorem 20. Let φ be guarded k-universal safety property with role guard $\mathcal{A}_R$ and literal guards $\mathcal{A}_L$, $\mathcal{C}$ be a bundle, θ_U be an interference invariant for $\mathcal{C}$, G be a PTG for $\mathcal{C}$, $(N_{\text{Exp.}} + N_{\text{Trans.}} + N_{\text{Impl.}})$ be a saturating network size for $(\mathcal{A}_R, \mathcal{A}_L, G)$. Define $N = (\max(N_{\text{Exp.}}, k) + N_{\text{Trans.}} + N_{\text{Impl.}})$. If $\mathsf{local}(\mathcal{C}, N, (\gamma_{\text{Role}} \cup \gamma_{\text{Imp}}), \theta_U) \models \mathsf{norm}(\mathcal{C}, \varphi)$, then $\forall M \in \mathbb{N} \cdot \mathsf{global}(\mathcal{C}, M) \models \varphi$.

[2]Theorem 19 did not require an explicit $\max(N_{\text{Exp.}}, 1)$ as the existence of a sender ensures $N_{\text{Exp.}} \geq 1$.

Proof. Let $\mathsf{LTS} = \mathsf{local}\left(\mathcal{C}, N, \left(\gamma_{\text{Role}} \cup \gamma_{\text{Imp}}\right), \theta_U\right)$. By definition of a saturating network size, the user configurations in LTS have representatives for each participation class in $\mathcal{C}$. Since $\left(\gamma_{\text{Role}} \cup \gamma_{\text{Imp}}\right)$ respects the network topology of $\mathcal{C}$, then LTS computes an inductive invariant for the control processes relative to θ_U. Therefore, verifying LTS is an instantiation of PCMC. It will now be shown that all additional equivalence classes, induced by φ, are represented by LTS. By including the guards of φ in the saturating network size, there is an additional representative for the extra transient and implicit participation classes induced by φ. By taking $\max\left(N_{\text{Exp.}}, k\right)$, rather than $N_{\text{Exp.}}$, there are also enough explicit participants to cover all existential quantifiers in φ. By Theorem 15 and Theorem 16, if $\mathsf{LTS} \models \varphi$, then $\forall M \in \mathbb{N} \cdot \mathsf{global}(\mathsf{norm}(\mathcal{C}), M) \models \mathsf{norm}(\mathcal{C}, \varphi)$. By Theorem 12 and Theorem 13, $(\forall M \in \mathbb{N} \cdot \mathsf{global}(\mathsf{norm}(\mathcal{C}), M) \models \mathsf{norm}(\mathcal{C}, \varphi)) \iff (\forall M \in \mathbb{N} \cdot \mathsf{global}(\mathcal{C}, M) \models \varphi)$ In conclusion, if $\mathsf{LTS} \models \mathsf{norm}(\mathcal{C}, \varphi)$, then $\forall M \in \mathbb{N} \cdot \mathsf{global}(\mathcal{C}, M) \models \varphi$. □

Theorem 20 enables proofs of φ_1 through to φ_6 in Chapter 3. For φ_4 and φ_5, the sum$(\cdot)$ and max$(\cdot)$ aggregates are instrumented according to Section 6.2. Details follow for a proof of φ_1. Recall $\mathsf{local}\left(\mathcal{C}, 4, \gamma_1, \theta_1\right)$ from Example 5.4.3. Since φ_1 is a 1-universal property, and since the `PTGBuilder` PTG for $\mathcal{C}$ has one explicit participant, it follows that $\max\left(N_{\text{Exp.}}, 1\right) = 1$. Using a model checker, $\mathsf{local}\left(\mathcal{C}, 4, \gamma_1, \theta_1\right) \models \varphi_1$ is certified by an inductive strengthening θ_1^*. Then by Theorem 20, $\mathcal{C}$ is also safe for 2^{160} users.

By definition, both the local and global bundles have state spaces that are 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 parameter. Even more remarkably, θ_1^* is an inductive invariant from Section 2.7, as it summarizes the safe control states that are closed under the interference of (γ_1, θ_1). Therefore, this section has achieved an exponential reduction in verification, and has automated the discovery of an inductive invariant (relative to an interference invariant).

Chapter 7

Bounded Analysis for Local Smart Contracts

In prior work, local symmetry analysis has been used as a tool for parameterized verification. However, local symmetry also provides insight into the structure of a smart contract. This section addresses the question of whether local bundle abstractions can be applied outside of parameterized smart contract verification. This section answers affirmatively, by offering an application to bounded analysis techniques, such as bounded model checking or fuzzing. A motivating example based on fuzzing is provided in Example 7.0.1.

Example 7.0.1. Let $\mathcal{C} =$ `Auction` from Fig. 3.1. Consider the property φ that states a user with address 100 never calls `withdraw`. To observe a counterexample, there must be a user with address 100 and a manager to stop the auction. Furthermore, the manager must not have address 100, else the user with address 100 cannot call withdraw. Therefore, $(\mathsf{global}(\mathcal{C}, N) \not\models \varphi) \Rightarrow (N \geq 101)$. However, a local bundle can abstract this execution to the zero-account, the smart contract account, a manager, and address 100. Therefore, $(\mathsf{local}(\mathcal{C}, N, \gamma_\top, \theta_\bot) \not\models \mathsf{norm}(\mathcal{C}, \varphi)) \Rightarrow (N \geq 4)$. In conclusion, local fuzzing can refute φ with only 4 users, whereas global fuzzing requires 101 users. □

A few points should be noted from Example 7.0.1. First, the local bundle abstraction is defined by $\gamma_\top$ and $\theta_\bot$. The choice of $\gamma_\top$ was necessary, as all users are concrete in the context of fuzzing. The choice of $\theta_\bot$, on the other hand, was arbitrary since all users are concrete and are not subjected to interference. Second, it should be noted that global fuzzing required 101 users, whereas local fuzzing required only 4 users. This gain was achieved through implicit address relabelling, as the smallest counterexample to φ required a user with literal address 100.

The rest of this chapter generalizes Example 7.0.1 to all instances of local contract fuzzing. Section 7.1 proves that counterexamples in MiniSol are monotonic (i.e., increasing the network size preserves counterexamples) and uses this to show that fuzzing with a normalized smart contract is meaningful. Section 7.2 analyzes the potential costs and benefits of local smart contract fuzzing.

7.1 Monotonicity of Counterexamples

Local bundle abstractions change the order of users in smart contracts. For example, the counterexample found through local fuzzing in Example 7.0.1 corresponds to an address space $\mathcal{A} = \{0, 1, 2, 100\}$. Clearly, $\mathcal{A}$ is non-consecutive, and does not correspond to any instance of global fuzzing. However, $\mathcal{A}$ is a superset of the address space [3] and a subset of the address space [101]. Therefore, if counterexamples are preserved monotonically by both consecutive and non-consecutive address spaces, then local fuzzing is sound.

The section proceeds as follows. Lemma 7 proves that if $\mathcal{A}_2 \subset \mathcal{A}_1$, then $\mathsf{global}^\circ(\mathcal{C}, \mathcal{A}_2)$ is simulated by $\mathsf{global}^\circ(\mathcal{C}, \mathcal{A}_1)$. The key insight in this simulation is that each state of $\mathsf{global}^\circ(\mathcal{C}, \mathcal{A}_2)$ is a projection onto $\mathcal{A}_2$ from multiple states in $\mathsf{global}^\circ(\mathcal{C}, \mathcal{A}_1)$. The simulation then follows directly from Theorem 8, which states that transactions can be executed locally through view projections. In Theorem 21, it is shown that this simulation preserves guarded universal safety properties. As a direct result, counterexamples to guarded universal safety properties are preserved monotonically.

Lemma 7. Let $\mathcal{C}$ be a bundle, $\mathcal{A}_1 \subseteq \mathbb{N}$, $\{0\} \subseteq \mathcal{A}_2 \subseteq \mathcal{A}_1$, $(S, P, f, s_0) = \mathsf{global}^\circ(\mathcal{C}, \mathcal{A}_1)$, and $(T, Q, g, t_0) = \mathsf{global}^\circ(\mathcal{C}, \mathcal{A}_2)$. If $\sigma = \{((c_t, \mathbf{u}), (c_s, \mathbf{v})) \in T \times S \mid c_t = c_s \wedge \mathbf{u} = \tau_{\mathcal{A}_2}(\mathbf{v})\}$, then $\mathsf{global}^\circ(\mathcal{C}, \mathcal{A}_2)$ is simulated by $\mathsf{global}^\circ(\mathcal{C}, \mathcal{A}_1)$ under the relation σ.

Proof. Since $0 \in \mathcal{A}_2$ and s_0 is zero-initialized, then $t_0 = \pi_{\mathcal{A}_2}(s_0)$. Therefore, $(s_0, t_0) \in \sigma$. Next, assume that $(t, s) \in \sigma$, $p \subseteq P$, and $f_p(t)$ does not revert. Let c be the control state of s and $\mathbf{u}$ be the user configuration for s. By definition of σ, c is the control state of t and $\pi_{\mathcal{A}_2}(\mathbf{u})$ is the user configuration of t. Since $\mathcal{A}_2 \subseteq \mathcal{A}_1$ and $f_p(s)$ does not revert, then $\pi_{\mathsf{view}(c,p)}(\mathbf{u}) = \pi_{\mathsf{view}(c,p)}(\pi_{\mathcal{A}_2}(\mathbf{u}))$. By Theorem 8, if $(c', \mathbf{u}') = f((c, \mathbf{u}), p)$, then $(c', \pi_{\mathcal{A}_2}(\mathbf{u}')) = g((c, \pi_{\mathcal{A}_2}(\mathbf{u}')), p)$. As a consequence, $(g(t, q), f(s, q)) \in \sigma$. Therefore, σ is a simulation relation. □

Theorem 21. Let $\mathcal{A}_1 \subseteq \mathbb{N}$, $\{0\} \subseteq \mathcal{A}_2 \subseteq \mathcal{A}_1$, and φ be a guarded universal safety property. If $\mathsf{global}^\circ(\mathcal{C}, \mathcal{A}_1) \models \varphi$, then $\mathsf{global}^\circ(\mathcal{C}, \mathcal{A}_2) \models \varphi$.

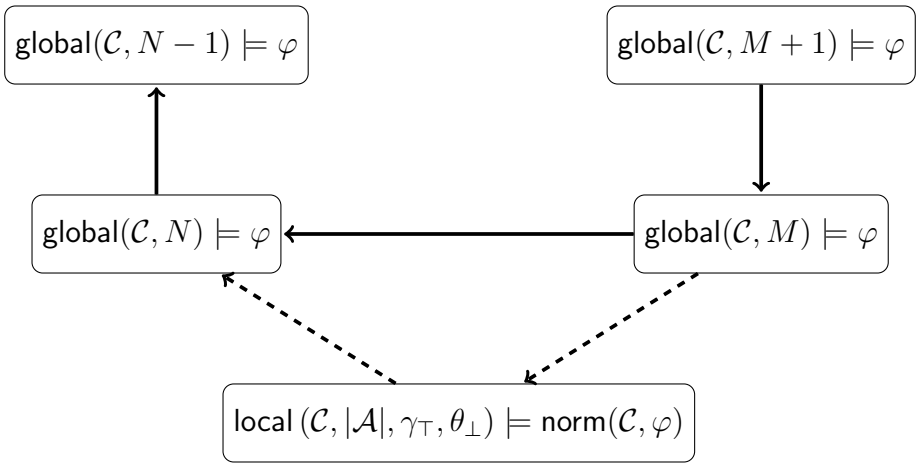


Figure 7.1: An illustration of Theorem 22. This figure highlights that a local bundle abstraction over-approximates all global bundles below some cutoff N, and under-approximates all global bundles above some cutoff M. Recall that in this example, $\mathcal{A} = (\mathsf{lits}(\mathcal{C}) \cup [N])$ and $M = \max(\mathcal{A})$.

Proof. Let φ be the guarded universal k-safety property given by (L, R, ξ). Assume for the intent of contradiction that $\mathsf{global}^\circ(\mathcal{C}, \mathcal{A}_1) \models \varphi$ and $\mathsf{global}^\circ(\mathcal{C}, \mathcal{A}_2) \not\models \varphi$. Let σ be the simulation relation defined in Lemma 7, and λ be the order-preserving mapping that corresponds to $\pi_{\mathcal{A}_2}$. Then there exists traces $(s_0, \ldots, s_m)$ and $(t_0, \ldots, t_m)$ of $\mathsf{global}^\circ(\mathcal{C}, \mathcal{A}_1)$ and $\mathsf{global}^\circ(\mathcal{C}, \mathcal{A}_2)$ such that $(s_0, \ldots, s_m) \models \varphi$, $(t_0, \ldots, t_m) \not\models \varphi$, and $\forall i \in [m+1] \cdot (t_i, s_i) \in \sigma$. Since φ is a safety property, there exists an $i \in [m+1]$ such that $s_i \models \varphi$ and $t_i \not\models \varphi$. Then t_i satisfies the guards of φ and $t_i \not\models \varphi$. Consequently, $|\mathcal{A}_2| \geq k$, else $t_i \models \varphi$ would be vacuously true. Let c be the control state of both s_i and t_i, $\mathbf{u}$ be the user configuration of s_i, and $\{j_1, \ldots, j_k\}$ be the indices of $\pi_{\mathcal{A}_2}(\mathbf{u})$ that violate φ. Then $\mathbf{u}_{\lambda(j_1)}$ through to $\mathbf{u}_{\lambda(j_k)}$ satisfy the guards of φ and $\xi\left(c, \mathbf{u}_{\lambda(j_1)}, \ldots, \mathbf{u}_{\lambda(j_k)}\right)$. However, $\lambda(j_1)$ through to $\lambda(j_k)$ are in $[\,|\mathcal{A}_1|\,]$, and therefore, $s_i \not\models \varphi$. By contradiction, $t_i \models \varphi$. □

From Theorem 21, it follows that $\mathsf{local}(\mathcal{C}, 4, \gamma_\top, \theta_\bot)$ in Example 7.0.1 over-approximates $\mathsf{global}(\mathcal{C}, 3)$ and under-approximates $\mathsf{global}(\mathcal{C}, 101)$. This simultaneous over-approximation and under-approximation is true of local fuzzing in general, as proven below in Theorem 22. Note that in this theorem, if $N \geq \max(\mathsf{lits}(\mathcal{C}))$, then $M = N$. In this case, local fuzzing is equivalence to global fuzzing. An illustration of this theorem can be found in Fig. 7.1.

Theorem 22. Let $\mathcal{C}$ be a bundle, $N \in \mathbb{N}$, $\mathcal{A} = (\mathsf{lits}(\mathcal{C}) \cup [N])$, $M = \max(\mathcal{A})$, and

φ be a guarded universal safety property. If $\mathsf{local}\,(\mathcal{C}, |\mathcal{A}|, \gamma_\top, \theta_\bot) \models \mathsf{norm}(\mathcal{C}, \varphi)$, then $\mathsf{global}(\mathcal{C}, N) \models \varphi$, else $\mathsf{global}(\mathcal{C}, M) \not\models \varphi$.

Proof. Let $\mathsf{lts} = \mathsf{local}\,(\mathcal{C}, |\mathcal{A}|, \gamma_\top, \theta_\bot)$. By Theorem 21, since $[N] \subseteq \mathcal{A} \subseteq [M]$, then:

$$(\mathsf{global}(\mathcal{C}, M) \models \varphi) \Rightarrow (\mathsf{global}^\circ(\mathcal{C}, \mathcal{A}) \models \varphi) \Rightarrow (\mathsf{global}(\mathcal{C}, N) \models \varphi)$$

By Theorem 12, Theorem 13, and Theorem 14, $(\mathsf{lts} \models \mathsf{norm}(\mathcal{C}, \varphi)) \iff (\mathsf{global}^\circ(\mathcal{C}, \mathcal{A}) \models \varphi)$. Therefore, if $\mathsf{lts} \models \mathsf{norm}(\mathcal{C}, \varphi)$, then $\mathsf{global}(\mathcal{C}, N) \models \varphi$, else $\mathsf{global}(\mathcal{C}, M) \not\models \varphi$. □

7.2 Impact on Counterexample Sizes

Section 7.1 proved that local bundle abstractions can be used in bounded analysis. This section analyzes the effectiveness of using local bundle abstractions in bounded analysis. To compare the cost of global fuzzing to local fuzzing, a notion of minimal counterexample is required. For global fuzzing, this is simply the smallest network size such that a counterexample appears. We call this a *counterexample cutoff* (see Definition 7.2.1). In the case of local fuzzing, this is the smallest address space associated with a counterexample. We call this a *minimal refutation* (see Definition 7.2.2). Whereas a counterexample cutoff is unique, there are many minimal refutations due to local symmetry.

Definition 7.2.1 (Counterexample Cutoff)**.** Let $\mathcal{C}$ be a bundle and φ be a guarded universal safety property. A *counterexample cutoff* of φ for $\mathcal{C}$ is a minimal $N \in \mathbb{N}$ such that $\mathsf{global}(\mathcal{C}, N) \not\models \varphi$.

Definition 7.2.2 (Minimal Refutation)**.** Let $\mathcal{C}$ be a bundle and φ be a guarded universal safety property. A *minimal refutation* of φ for $\mathcal{C}$ is a minimal set $\mathcal{A} \subseteq \mathbb{N}$ such that $\mathsf{global}^\circ(\mathcal{C}, \mathcal{A}) \not\models \varphi$.

It follows from monotonicity that a counterexample cutoff is an upper bound on the size of a minimal refutation. This is because, given any cutoff N, the address space $[N]$ must be a refutation (see Theorem 23). However, this does not prove that local fuzzing is always more efficient than global fuzzing. Since local bundle abstractions are normalized, they necessarily include all literal addresses, whereas a minimal refutation may require a subset of literal addresses. Theorem 24 relates the number of literal addresses to the cost of local fuzzing.

Theorem 23. Let $\mathcal{C}$ be a bundle and φ be a guarded universal safety property. If N is a counterexample cutoff of φ for $\mathcal{C}$ and $\mathcal{A}$ is a minimal witness of φ for $\mathcal{C}$, then $|\mathcal{A}| \leq N$.

Proof. Assume for the intent of contradiction that $|\mathcal{A}| > N$. If N is a counterexample cutoff of φ, then $\mathsf{global}^{\circ}(\mathcal{C}, [N]) \not\models \varphi$. Then $[N]$ is a refutation of φ for $\mathcal{C}$. Furthermore, $|[N]| < |\mathcal{A}|$. Therefore, $\mathcal{A}$ is not minimal. By contradiction, $|\mathcal{A}| \leq N$. □

Theorem 24. Let $\mathcal{C}$ be a bundle and φ be a guarded universal safety property. If $\mathcal{A}$ is a minimal refutation of φ for $\mathcal{C}$, then local fuzzing will find a counterexample within a network size of $|\mathcal{A} \cup \mathsf{lits}(\mathcal{C})|$.

Proof. Since $\mathcal{A}$ is a minimal refutation of φ for $\mathcal{C}$, then $\mathsf{global}^{\circ}(\mathcal{C}, \mathcal{A}) \not\models \varphi$. By Theorem 21, if $\mathsf{global}^{\circ}(\mathcal{C}, \mathcal{A}) \not\models \varphi$, then $\mathsf{global}^{\circ}(\mathcal{C}, \mathcal{A} \cup \mathsf{lits}(\mathcal{C})) \not\models \varphi$. By Theorem 12, Theorem 13, and Theorem 14:

$$(\mathsf{local}\,(\mathcal{C}, |\mathcal{A} \cup \mathsf{lits}(\mathcal{C})|, \gamma_{\top}, \theta_{\bot}) \models \mathsf{norm}(\mathcal{C}, \varphi)) \iff (\mathsf{global}^{\circ}(\mathcal{C}, \mathcal{A} \cup \mathsf{lits}(\mathcal{C})) \models \varphi)$$

Therefore, local fuzzing will find a counterexample within a network size of $|\mathcal{A} \cup \mathsf{lits}(\mathcal{C})|$. □

Theorem 24 shows that the effectiveness of local fuzzing depends on both the smart contract and the property. To illustrate this, let $\mathcal{C}$ be a bundle, φ be a guarded universal safety property, N be a counterexample cutoff of φ for $\mathcal{C}$, and $\mathcal{A}$ be a minimal witness of φ for $\mathcal{C}$. In the worst case, local fuzzing requires a network size of $|\mathcal{A}| + |\mathsf{lits}(\mathcal{C})|$. As shown in Theorem 23, this is at most $N + |\mathsf{lits}(\mathcal{C})|$. Therefore, the worst-case overhead of local fuzzing is constant with respect to the property φ. Often $\mathsf{lits}(\mathcal{C})$ is small for real-world smart contracts. In the best case, local fuzzing requires a network size of $|\mathcal{A}|$. As shown in Theorem 23, $|\mathcal{A}| \leq N$, and therefore, local fuzzing can also permit smaller network size. This happens when a counterexample requires one or more implicit participants with large addresses. It is concluded that local fuzzing can accelerate smart contract fuzzing, but in the worst case, incurs an (often small) overhead determined by the source text of $\mathcal{C}$.

Chapter 8

Implementation

This chapter describes SmartACE, an open-source tool[1] for Solidity smart contract analysis. SmartACE supports an extended version of MiniSol with structures, inheritance, libraries, enums, strings, and logging statements such as `emit`. An example program is given in Fig. 8.1.

The goal of SmartACE is to bring state-of-the art program analyzers from the LLVM-community to Solidity smart contracts. LLVM is a compiler development framework with emphasis on program transformation and program analysis [45]. Over the past two decades, many languages have adopted LLVM-based compilers, such as C++ [45], CUDA [74], Haskell [68], and Rust. This has motivated the software engineering community to develop many popular tools for the development, testing, and verification of LLVM programs. This section focuses on model checkers such as SeaHorn [28] and SMACK [61], along with greybox fuzzers (i.e., coverage-guided fuzzers [78]) such as libFuzzer [47] and AFL [76].

8.1 Architecture

SmartACE is a smart contract analysis framework guided by local symmetry analysis. As opposed to other smart contract tools, SmartACE performs all analysis against a local bundle abstraction for a provided smart contract. The abstraction is obtained through source-to-source translation from Solidity to an LLVM-based *harness*. To avoid reinventing the wheel, SmartACE makes use of off-the-shelf analyzers from the LLVM-community. The design of SmartACE is guided by four key principles.

[1] https://github.com/contract-ace

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

  modifier canParticipate() {
    require(msg.sender != manager);
    require(!stopped);
    _;
  }

  function bid() public payable canParticipate() {
    require(msg.value > leadingBid);
    bids[msg.sender] = msg.value;
    leadingBid = msg.value;
  }
  function withdraw() public canParticipate() {
    require(bids[msg.sender] != leadingBid);
    bids[msg.sender] = 0;
  }

  // Fallback, constructor, computeMinBid(), and stop().
}
```

```
contract Mgr {
  Auction auction;
  constructor() public {
    auction = new Auction(address(this));
  }
  function stop() public { auction.stop(); }
}

contract TimedMgr is Mgr {
  event Stopped(address _by, uint _block);
  uint start = block.number;
  uint dur;

  constructor(uint _d) public { dur = _d; }
  function stop() public {
    require(start + dur < block.number);
    emit Stopped(msg.sender, block.number);
    super.stop();
  }
  function check() public returns (bool, uint) {
    if (start + dur < block.number) {
      return (false, block.number - dur - start);
    } else { return (true, 0); }
  }
}
```

Figure 8.1: An extension to Fig. 3.1 that exercises additional language support.

1. **Reusability:** The framework should support state-of-the-art and off-the-shelf analyzers to minimize the risk of incorrect analysis results.

2. **Reciprocity:** The framework should produce intermediate artifacts that can be used as benchmarks for off-the-shelf analyzers.

3. **Extensibility:** The framework should extend to new analyzers without modifying existing features.

4. **Testability:** The intermediate artifacts produced by the framework should be executable, to support both validation and interpretation of results.

These principles are achieved through the architecture in Fig. 8.2. SmartACE takes as input a smart contract with assertions and optionally an *interference invariant*. The inputs are passed to a source-to-source translator, to obtain an LLVM-based, sequential model of the smart contract and its environment (see Section 8.2). This model is called a *harness*. Harnesses use an interface called libVerify to integrate with arbitrary analyzers, and are therefore analyzer-agnostic (see Section 8.3). When an analyzer is chosen, CMake is used to automatically compile the harness, the analyzer, and its dependencies, into an executable program. Analysis results for the program are returned by SmartACE.

The SmartACE architecture achieves its guiding principles as follows. To ensure *reusability*, SmartACE uses state-of-the-art tools for build automation (CMake) and

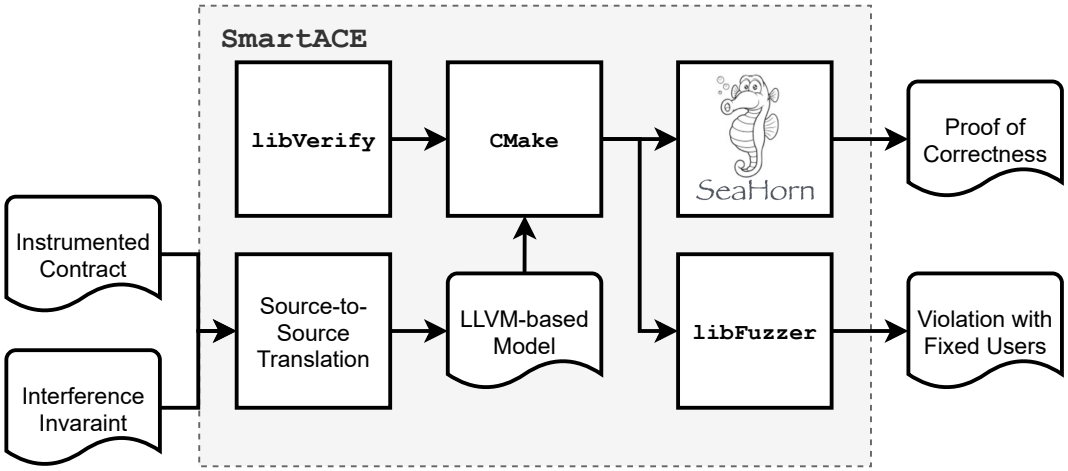


Figure 8.2: The architecture of SmartACE for integration with SeaHorn for model checking and libFuzzer for greybox fuzzing.

program analysis (e.g., SeaHorn and libFuzzer). The source-to-source translation is implemented within the Solidity compiler to utilize existing source-code analysis (e.g., AST construction, type resolution). To ensure *reciprocity*, the SmartACE architecture integrates third-party tools entirely through intermediate artifacts. To ensure *extensibility*, the libVerify interface is used together with CMake build scripts to orchestrate smart contract analysis. A new analyzer can be added to SmartACE by first creating a new implementation of libVerify, and then adding a build target to the CMake build scripts. Finally, *testability* is achieved by ensuring all harnesses are executable. As shown in Section 8.4, executable harnesses provide many benefits, such as validating counterexamples from model checkers, and manually inspecting harness behaviour.

8.2 Contract Modelling

This section describes the translation from a smart contract with annotations, to an LLVM-based harness. A high-level overview is provided by Fig. 8.3. First, static analysis is applied to a smart contract, such as resolving inheritance and over-approximating user participation (see Section 8.2.1). Next, the analysis results are used to convert each `contract` to LLVM structures and functions (see Section 8.2.2). Finally, these functions are combined into a harness that schedules an unbounded sequence of smart contract transactions (see Section 8.2.3).

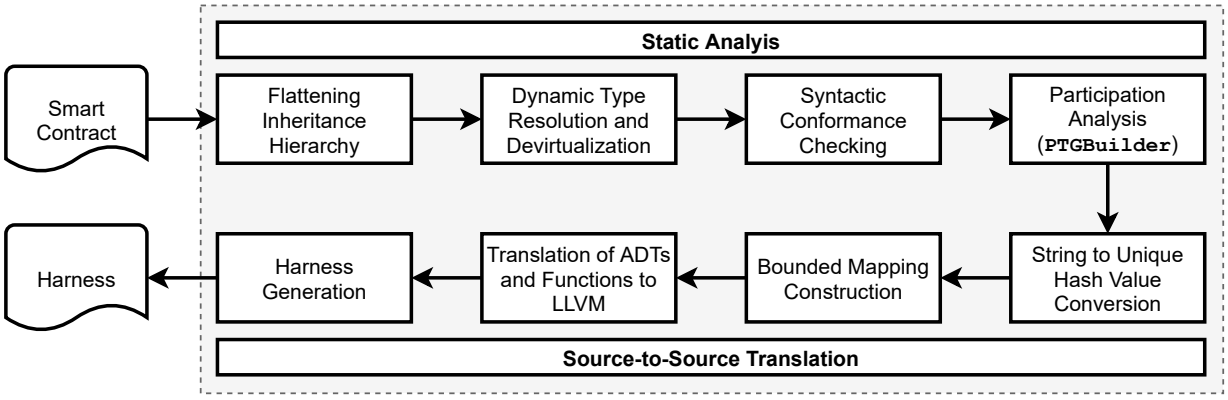


Figure 8.3: The analysis and transformations performed by SmartACE.

8.2.1 Static Analysis

The static analysis in SmartACE is illustrated by the top row of Fig. 8.3. At a high-level, static analysis ensures that a bundle conforms to the restrictions of Chapter 3, and extracts facts about the bundle required during the source-to-source translation. Bundle facts include a flat inheritance hierarchy [9], the dynamic type of each contract-typed variable, the devirtualization of each call (e.g., [7]), and the PTG of the bundle.

SmartACE obtains a PTG through `PTGBuilder`, and over-approximates conformance checks through syntactic rules. Therefore, it is possible for SmartACE to reject valid smart contracts due to imprecision. For this reason, SmartACE uses incremental passes to restrict the code surface that reaches the conformance checker. The first pass flattens the inheritance hierarchy by duplicating member variables and specializing methods. The second pass resolves the dynamic type of each contract-typed variable, by identifying its allocation sites. For example, the dynamic type for state variable `auction` in `TimedMgr` of Fig. 8.1 is `Auction` due to the allocation at line 29. The third pass uses the dynamic type of each contract-typed variable, to resolve all virtual calls in the smart contract. For example, `super.stop` at line 43 devirtualizes to method `stop` of contract `Mgr`. The fourth pass constructs a call graph for the `public` and `external` methods of each smart contract. Only methods in the call graph are subject to the conformance checker.

8.2.2 Source-to-Source Translation

Source-to-source translation relies on the call graph and PTG obtained through static analysis. The translation is illustrated by the bottom row of Fig. 8.3. A translation for

```
1 struct Map_1 { sol_uint256_t data_0; /* ... */ sol_uint256_t data_4; };
2 struct Auction {
3   sol_address_t model_address;  sol_uint256_t model_balance;
4   struct Map_1 user_bids;
5   sol_address_t user_manager;   sol_uint256_t user_leadingBid;
6   sol_bool_t user_stopped;      sol_uint256_t user___sum;
7 };
8 void TimedMgr_Method_stop(
9   struct TimedMgr *self, sol_address_t sndr, sol_uint256_t value,
10  sol_uint256_t bnum, sol_uint256_t time, sol_bool_t paid, sol_address_t orig) {
11  sol_require(self->user_start.v + self->user_dur.v < bnum.v, 0);
12  sol_emit("Stopped(msg.sender, block.number)");
13  Mgr_Method_For_TimedMgr_stop(self, /*...*/ time, Init_sol_bool_t(0), orig);
14 }
15 sol_bool_t TimedMgr_Method_check(
16  struct TimedMgr *self, /*...globals...*/, sol_uint256_t *out_1) {
17  (*out_1) = Init_sol_uint256_t(0);
18  if (self->user_start.v + self->user_dur.v < bnum.v) {
19    out_1->v = bnum.v - self->user_dur.v - self->user_start.v;
20    return Init_sol_bool_t(0);
21  }
22  return Init_sol_bool_t(1);
23 }
24 void Auction_Method_1_bid(struct Auction *self, /*...globals...*/) {
25  sol_require(value.v > self->user_leadingBid.v, 0);
26  Write_Map_1(&self->user_bids, sndr, value);
27  self->user_leadingBid = value;
28 }
29 void Auction_Method_bid(struct Auction *self, /*...globals...*/) {
30  if (paid.v == 1) self->model_balance.v += value.v;
31  sol_require(sndr.v != self->user_manager.v, 0);
32  sol_require(!self->user_stopped.v, 0);
33  Auction_Method_1_bid(self, /*...globals...*/);
34 }
```

Figure 8.4: Partial modelling of the types and methods in Fig. 8.1 as C code (LLVM).

Fig. 8.1 is given in Fig. 8.4. Note that the C language is used in Fig. 8.4, rather than LLVM, as C is more human-readable.

Abstract Data Types (ADTs). An ADT is either a `struct` or a `contract`. Each `struct` is translated directly to an LLVM structure. The name of the structure is prefixed by the name of its containing `contract` to avoid name collisions. Each `contract` is translated to an LLVM structure of the same name, with a field for its address (`model_address`), a field for its balance (`model_balance`), and a field for each user-defined member variable. An example is given for `Auction` at line 3.

Primitive Types. Primitive types include all integer types, along with `bool`, `address`, and `enum` (unbounded arrays are not yet supported in SmartACE). Integer types are mapped to singleton structures, according to their signedness and bit-width. For example, the type of `leadingBid` is mapped to `sol_uint256_t` (see line 5). Each `bool` type is mapped to the singleton structure `sol_bool_t`, which contains the same underlying type as `uint8` (see line 6). Each `address` type is mapped to the singleton structure `sol_address_t`, which contains the same underlying type as `uint160` (see line 5). Each `enum` is treated as an unsigned integer of the nearest containing bit-width. Benefits of singleton structures, and their underlying types, are discussed in Section 8.3.

Functions. Methods and modifiers are translated to LLVM functions. Methods are specialized according to the flattened inheritance hierarchy, and modifiers are specialized to each method. To avoid name collisions, each function is renamed according to the contract that defines it, the contract that is calling it, and its position in the chain of modifiers. For example, the specialization of method `Mgr.stop` for `TimedMgr` is `Mgr_Method_For_TimedMgr_stop`. Likewise, the specializations of method `Auction.bid` and its modifier `canParticipate` are `Auction_Method_1_bid` and `Auction_Method_bid`, respectively. Extra arguments are added to each method to represent the current call state (see `self` through to `orig` on line 9). Specifically, `self` is `this`, `sndr` is `msg.sender`, `value` is `msg.value`, `bnum` is `block.number`, `time` is `block.timestamp`, and `orig` is `msg.origin`. A special argument, `paid`, indicates if `msg.value` has been added to a contract's balance (see line 13, where `paid` is set to false). If `paid` is true, then the balance is updated before executing the body of the method (see line 30). Multiple return values are handed through the standard practice of output variables. For example, the argument `out_1` in `TimedMgr_Method_check` represents the second return value of `check`.

Statements and Expressions. Most expressions map directly from Solidity to LLVM (as both are typed imperative languages). Special cases are outlined. Each `assert` maps to `sol_assert` from libVerify, which causes a program failure given argument `false`. Each `require` maps to `sol_require` from libVerify, which reverts a transaction given argument `false` (see line 31). For each `emit` statement, the arguments of the event are expanded out, and then a call is made to `sol_emit` (see line 12). For each method call, the devirtualized call is obtained from the call graph, and the call state is propagated (see line 13 for the devirtualized called to `super.stop`). For `external` method calls, `paid` and `msg.sender` are reset.

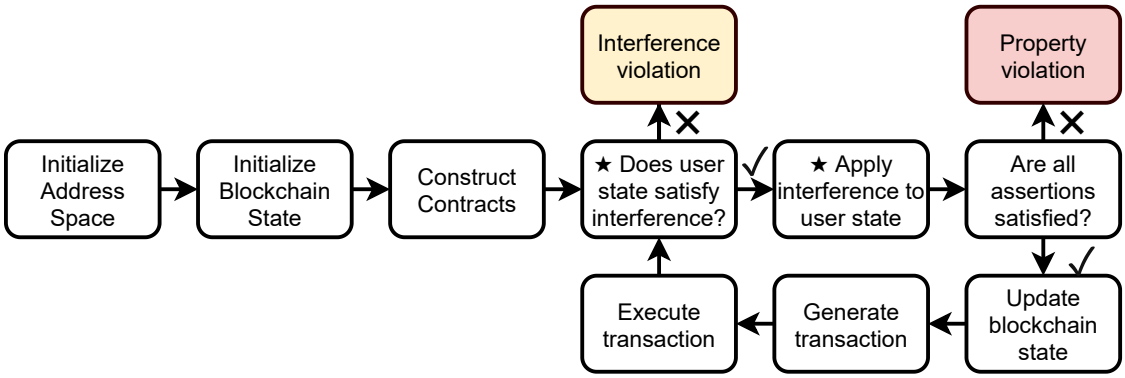


Figure 8.5: The control-flow of a test harness. Each ⋆ denotes an optional step.

Mappings. Each `mapping` is translated to an LLVM structure. This structure represents a bounded mapping with an entry for each participant in the PTG. For example, if a PTG has N participants, then a one-dimensional `mapping` will have N entries, and a two-dimensional `mapping` will have N^2 entries. Since `mapping` types are unnamed, the name of each LLVM structure is generated according to declaration order. For example, `bids` of `Auction` is the first mapping in Fig. 8.1, and translates to `Map_1` accordingly (see line 1). Accesses to `Map_1` are encapsulated by `Read_Map_1` and `Write_Map_1` (see line 26).

Strings. Each string literal is translated to a unique integer value. This model supports string equality, but disallows string manipulation. Note that string manipulation is avoided by many smart contracts due to gas costs.

Addresses. Implicit participation is induced by literal addresses. This means that the value of a literal address is unimportant, so long as it is unique and constant. For reasons outlined in Section 8.2.3, it is important to set the value of each literal address programmatically. Therefore, each literal address is translated to a unique global variable. For example, `address(0)` translates to `g_literal_address_0`.

8.2.3 Harness Design

A harness provides an entry-point for LLVM analyzers. Currently, SmartACE implements a single harness that models a blockchain from an arbitrary state, and then schedules an unbounded sequence of transactions for contracts in a bundle. A high-level overview of this harness is given in Fig. 8.5. The harness for `Auction` in Fig. 8.1 is depicted in Fig. 8.6.

```
sol_bool_t paid; paid.v = 1;
// Address space initialization.
struct TimedMgr sc_1;
struct Auction *sc_2;
sc_2 = &sc_1.user_auction;
g_literal_address_0 = 0;
sc_1.model_address.v = 1;
sc_2.model_address.v = 2;
// Blockchain initialization.
sol_uint256_t bnum;
bnum.v = ND_UINT(1,256,"bnum");
sol_uint256_t time;
time.v = ND_UINT(2,256,"time");
// Contract construction.
sol_address_t sndr;
sndr.v = ND_RANGE(3,3,5,"sndr");
sol_uint256_t value; value.v = 0;
sol_uint256_t arg___d;
arg___d.v = ND_UINT(4,256,"_d");
Init_TimedMgr(
  &sc_1, sndr, value, bnum, time,
  paid, sndr, arg___d);
// Transaction Loop.
while (sol_continue()) {
  sol_on_transaction();
  // Interference.
  if (sol_can_interfere()) {/*...*/}
  // Update blockchain state.
  if (ND_RANGE(5,0,2,"inc_time")) {
    bnum.v =
      ND_INCREASE(6,bnum.v,1,"bnum");
    time.v =
      ND_INCREASE(7,time.v,1,"time");
  }
  // Generate transaction.
  switch (ND_RANGE(8,0,6,"call")) {
  case 0: {
    /*...generate arguments...*/
    TimedMgr_Method_stop(/*...*/);
    break;
  } /*...other public methods...*/
  }
}
```

Figure 8.6: The harness for Fig. 8.1. Logging is omitted to simplify the presentation.

Modelling Nondeterminism. All nondeterministic choices are resolved by interfaces from LIBVERIFY. `ND_INT(id,bits,msg)` and `ND_UINT(id,bits,msg)` choose integers of a desired signedness and bit-width. `ND_RANGE(id,lo,hi,msg)` chooses values between `lo` (inclusively) and `hi` (exclusively). `ND_INCREASE(id,old,msg)` chooses values larger than `old`. In all cases, `id` is an identifier for the callsite, and `msg` is used for logging purposes.

Address Space. An abstract address space restricts the number of addresses in a harness. It assigns abstract address values to each contract and literal address global variable. Assume that there are N contracts, M literal addresses, and K non-implicit participants. The corresponding harness has abstract addresses 0 to $N + M + K - 1$. Constraints are placed on address assignments to prevent impossible address spaces, such as two literal addresses sharing the same value, two contracts sharing the same value, or a contract having the same value as the zero-account. The number of constraints must be minimized, to simplify symbolic analysis. In SMARTACE, the following partitioning is used. `Address(0)` is always mapped to abstract address 0 (see line 6). Abstract addresses 1 to N are assigned to contracts according to declaration order (see lines 7–8). Literal addresses are assigned arbitrary values from 1 to $N + M$. This allows contracts to have literal addresses. Disequality constraints ensure each assignment is unique. Senders are then chosen from the range of non-contract addresses (see line 16).

Blockchain Model. SmartACE models `block.number`, `block.timestamp`, `msg.value`, `msg.sender`, and `msg.origin`. The block number and timestamp are maintained across transactions by `bnum` at line 10 and `time` at line 12. Before transaction generation, `bnum` and `time` may be incremented in lockstep (see lines 29–33). Whenever a method is called, `msg.sender` is chosen from the non-contract addresses (e.g., line 16). The value of `msg.sender` is also used for `msg.origin` (e.g., the second argument on line 22). If a method is `payable`, then `msg.value` is chosen by `ND_UINT`, else `msg.value` is set to 0 (e.g., line 17).

Transaction Loop. Transactions are scheduled by the loop at line 24. The loop terminates if `sol_continue` from libVerify returns `false` (this does not happen for most analyzers). Upon entry to the loop, `sol_on_transaction` from libVerify provides a hook for analyzer specific bookkeeping. Interference is then checked and re-applied, provided that `sol_can_interfere` returns `true` at line 27. A transaction is picked at line 36 by assigning a consecutive number to each valid method, and then choosing a number from this range. Arguments for the method are chosen using `ND_INT` and `ND_UINT` for integer types, and `ND_RANGE` for bounded types such as `address`, `bool` and `enum` (see lines 15–19 for an example).

Interference. A harness may be instrumented with interference invariants to enable modular reasoning, such as PCMC. As illustrated in Fig. 8.5, interference is checked and then re-applied before executing each transaction. Note that checking interference after a transaction would be insufficient, as this would fail to check the initial state of each contract. To apply interference, a harness chooses a new value for each mapping entry, and then assumes that these new values satisfy their interference invariants. To check interference, a harness chooses an arbitrary entry from a mapping, and asserts that the entry satisfies its interference invariant. Note that asserting each entry explicitly would challenge symbolic analyzers. For example, a two-dimensional mapping with 16 participants would require 256 assertions.

Limitations. The harness has three key limitations. First, as gas is unlimited, the possible transactions are over-approximated. Second, there is no guarantee that time must increase (i.e., a fairness constraint), so time-dependent actions may be indefinitely postponed. Third, reentrancy is not modeled, though this is sufficient for EECF contracts.

Interface	Description
`sol_continue()`	Returns `true` if the transaction execution loop should continue.
`sol_can_interfere()`	Returns `true` if interference should be applied and validated.
`sol_require(cond, msg)`	If `cond` is `false`, then `msg` is logged and the transaction aborts.
`sol_assert(cond, msg)`	If `cond` is `false`, then `msg` is logged and the program fails.
`sol_emit(expr)`	Performs analyzer-specific processing for a call to `emit expr`.
`ND_INT(id, n, msg)`	Returns an `n`-bit signed integer. The choice is logged as "`msg`: *value*".
`ND_UINT(id, n, msg)`	Returns an `n`-bit unsigned integer. The choice is logged as "`msg`: *value*".
`ND_RANGE(id, lo, hi, msg)`	Returns an 8-bit unsigned integer between `lo` (inclusively) and `hi` (exclusively). The choice is logged as "`msg`: *value*".
`ND_INCREASE(id, cur, strict, msg)`	Returns a 256-bit unsigned integer that is greater than or equal to `cur`. If `strict` is true, then the integer is strictly larger than `cur`. The choice is logged as "`msg`: *value*".
`smartace_log(msg)`	Logs `msg`.

Table 8.1: Summary of the libVerify interface.

```
1 int rv = *;
2 assume(lo <= rv && rv < hi);
3 return rv;
```

(a) An implementation for SeaHorn.

```
1 int rand = *;
2 int rv = lo + (rand % (hi - lo));
3 return rv;
```

(b) An implementation for libFuzzer.

Figure 8.7: Possible implementations of `ND_RANGE(n,lo,hi,msg)`.

8.3 Integration with Analyzers

CMake and libVerify are used to integrate LLVM analyzers with SmartACE. Functions from libVerify, as described in Table 8.1, provide an interface between a harness and an analyzer (usage of these functions is described in Section 8.2). Each implementation of libVerify codifies how a certain analyzer should interact with a harness. Build details are resolved using CMake scripts. For example, CMake arguments are used to switch the implementation of primitive singleton structures between native C integers and Boost multiprecision integers. To promote extensibility, certain interfaces in libVerify are designed with many analyzers in mind. Two key examples are bounded value selection and the placement of non-determinism.

8.3.1 Challenge: Bounded Value Selection

In libVerify, the functions `ND_INT` and `ND_UINT` are used as sources of non-determinism. In principle, all selections could be implemented using these interfaces. However, certain operations, such as "*increase the current block number*," or "*select an address between* 3

and 5," require specialized implementations, depending on the mode of analysis. For this reason, libVerify provides multiple interfaces for non-determinism, such as `ND_INCREASE` and `ND_RANGE`. To illustrate this design choice, the implementations of `ND_RANGE` for SeaHorn and libFuzzer are discussed.

The interface `ND_RANGE(id,lo,hi,msg)` returns a value between `lo` (inclusively) and `hi` (exclusively). Efficient implementations are given for SeaHorn and libFuzzer in Fig. 8.7a and Fig. 8.7b, respectively. The SeaHorn implementation is correct, since failed assumptions in symbolic analysis simply restricts the domain of each symbolic variable. Intuitively, assumptions made in the future can influence choices made in the past. This design does not work for libFuzzer, as failed assumptions in libFuzzer simply halt execution. This is because all values in libFuzzer are concrete. Instead, a value is constructed between `lo` and `hi` through modular arithmetic. In contrast, many symbolic analyzers struggle with non-linear constraints such as modulo. Therefore, neither implementation is sufficient for both model checking and fuzzing.

8.3.2 Challenge: Placement of Non-determinism

Not all analyzers provide non-determinism in the same way. In SeaHorn, external functions are used as sources of non-determinism. For counterexample generation, it is useful to have a unique external function for each callsite. In libFuzzer, generated values are passed to the harness through an input array. This presents a challenge, since the external functions for SeaHorn should be exposed to the callsite, whereas the input array used by libFuzzer should be hidden as an implementation detail. However, if the external functions are called unconditionally, then the harness would not compile for libFuzzer.

This problem is solved in libVerify by adding a "hint" to each value selection interface. For SeaHorn, the hint is used as a source of non-determinism, whereas for libFuzzer, the hint is ignored. The call is wrapped by a preprocessor macro that automatically generates a hint. For example, `ND_RANGE(id,lo,hi,msg)` is a macro wrapper to `nd_range`. For SeaHorn, `ND_RANGE` passes the return value from external function `nd_{id}` as a hint, whereas for libFuzzer, the hint defaults to 0. Alternatively, SmartACE could generate an implementation for each external function when SeaHorn is not in use, thought this solution is specific to SeaHorn and counter to the principle of extensability.

8.4 Modes of Analysis

SmartACE has been instantiated for greybox fuzzing and PCMC. The current implementation of libVerify supports libFuzzer for fuzzing and SmartACE for model checking. Other analyzers, such as AFL and SMACK, can be supported by extending libVerify.

Interactive Test Harness. A default implementation of libVerify provides an interactive test harness. Non-determinism, and the return values for `sol_continue`, are resolved through standard input. Events such are `smartace_log` and `sol_emit` are printed to standard output. The `sol_on_transaction` hook is used to collect test metrics, such as the number of transactions. As mentioned in Section 8.1, providing an interactive harness improves the testability of SmartACE.

Greybox Fuzzing. In greybox fuzzing, the harness is instantiated with N participants, and each participant is treated concretely. As opposed to other smart contract fuzzing techniques, SmartACE performs all fuzzing against a local bundle abstraction. This ensures that all implicit participants are in the address space.

PCMC. PCMC for a provided interference invariant either proves a bundle is safe for any number of users, or finds a counterexample to compositionality. The harness is instantiated from the PTG, with three choices of concretization ($\gamma_\bot$, γ_{Imp}, or $\gamma_{\text{Imp}} \cup \gamma_{\text{Role}}$). Increasing the number of concrete participants increases the precision of the analysis, but also increases the size of the state space. As the harness is executable, SmartACE is able to compile and execute counterexamples found by a model checker. With SeaHorn, integers can be bit-precise [42], or over-approximated by linear integer arithmetic [10]. The predicate synthesis feature in SeaHorn can also be used to infer interference invariants. Synthesized predicates may optionally depend on all control data, or no control data.

Chapter 9

Evaluations

This chapter reports on the effectiveness of SmartACE in analyzing real-world smart contracts. The evaluation answers the following research questions:

RQ1: Compliance. Can MiniSol represent real-world smart contracts?

RQ2: Effectiveness. Is SmartACE effective for MiniSol smart contracts?

- **RQ2a: Verification.** Is smart contract PCMC able to verify real-world properties?
- **RQ2b: Fuzzing.** Is local fuzzing able to find bugs in real-world smart contracts?
- **RQ2c: Synthesis.** Can predicate synthesis automate smart contract PCMC?

RQ3: Performance. Is SmartACE competitive with other verification techniques?

To answer the above research questions, we have curated a benchmark of 89 properties across 15 smart contracts (see Table 9.1). Contracts `Alchemist` through to `Mana` are from VerX [58]. The VerX benchmark set consists of real-world smart contracts that were provided to ChainSecurity for auditing purposes. Contracts `Fund` and `Auction` were added to offset the lack of parameterized properties in existing benchmarks. The `QSPStaking` contract comprises the Quantstamp Assurance Protocol[1] for which we checked real-world properties provided by Quantstamp. In Table 9.1, *Time* is the total analysis time, *Inv. Size* is the number of clauses in an interference invariant, *Users* is the number of users required by `PTGBuilder`, ✓ is the number successful evaluations, and ✗ is the number of unsuccessful evaluations.

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

Contracts			VerX	PCMC			Fuzzing		Synthesis		
Name	Prop.	LOC	Time	Time	Inv. Size	Users	✓	X	✓	X	Time
Alchemist	3	401	29	7	0	7	0	1	3	0	208
ERC20	9	599	158	12	1	5	9	0	9	0	103
Melon	16	462	408	30	0	7	14	2	16	0	979
MRV	5	868	887	2	0	7	5	0	5	0	428
Overview	4	66	211	4	0	8	3	1	4	0	9
PolicyPal	4	815	20 773	26	0	8	1	2	4	0	3 118
Zebi	5	1 209	77	8	0	7	5	0	5	0	487
Zilliqa	5	377	94	8	0	7	4	0	5	0	501
Brickblock	6	549	191	13	0	10	4	2	6	0	1 214
Crowdsale	9	1 198	261	223	0	8	6	1	9	0	238
ICO	8	650	6 817	371	0	16	5	3	0	8	—
VUToken	5	1 120	715	19	0	10	2	3	1	4	17
Mana	4	885	41 409	—	—	—	—	—	—	—	—
Fund	2	38	—	1	0	6	1	1	—	—	—
Auction	1	42	—	1	1	5	1	0	1	0	1
QSPStaking	4	1 550	—	3	7	8	—	—	—	—	—

Table 9.1: Summary of results for all SmartACE case studies.

9.1 Case Studies

Five case studied were performed using the data in Table 9.1. All artifacts described in these case studies are publicly available[2].

VerX Verification Study. This study evaluates the *Compliance*, *Verification Effectiveness*, and *Performance* of SmartACE. The goal of this study is to compare SmartACE to benchmarks from a closely-related and state-of-the-art smart contract verifier. The VerX benchmarks were selected for this study, since VerX is also a semi-automated[3] tool that verifies past-time LTL properties. We report on the number of compliant benchmarks, the proportion of compliant benchmarks that could be verified by SmartACE using PCMC, and the time required for verification.

- **Compliance:** We found that 8 out of 13 benchmarks are compliant after removing dead code. With manual abstraction, 12 out of 13 benchmarks are compliant. The manual abstractions are as follows. In `Brickblock`, inline assembly is used to revert transactions with smart contract senders. To comply with SmartACE, we remove the assembly as an over-approximation. To support `Crowdsale`, we manually resolve dynamic calls not supported by SmartACE. In `ICO`, calls can be made to arbitrary

[2] https://github.com/contract-ace

[3] End-users must provide predicate abstractions [35] to VerX.

contracts (by address). However, these calls adhere to EECF 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. In `VUToken`, 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 methods, and do not impact reachability.

- **Verification:** All compliant VerX benchmarks were verified by SmartACE. However, it was also discovered that most VerX properties are not parameterized. Specifically, 40 properties lack parameterization in general, 19 properties are parameterized but do not involve user data, and 23 properties involve user data but are not universally quantified.

- **Performance:** For each smart contract, the reported "average" time of VerX was compared to the total time of SmartACE. Note that it is unclear how the reported times of VerX were obtained. The authors of VerX were contacted, but were unable to provide the original data. A VerX client was obtained from the authors, but it failed to connect to the VerX servers. Therefore, we conservatively assume that all times reported by VerX are total. All VerX experiments were performed using a faster a processor (4 cores at 2.8GHz versus 4 cores at 3.4GHz) and additional RAM (16GB versus 64GB). In each case, SmartACE significantly outperformed VerX, achieving a speedup of at least 10x for all but 2 contracts.

Auction Study. This study evaluates the *Verification Effectiveness* of SmartACE. The goal of this study is to validate SmartACE against universal properties. In this case study, an Auction smart contract was designed (similar to Fig. 3.1), and SmartACE was used to verify that "*Each bid is at most the maximum bid.*" Each artifact obtained from SmartACE was audited by hand, and validated, including the abstract address space. The discovery of an interference invariant was semi-automated, and aided by counterexamples from SeaHorn. First, the interference invariant was assumed to be $\theta_\top$. SeaHorn provided a counterexample in which a user spuriously obtained a bid of 1 before any bids had been placed. The failed assertion stated `bids[i] <= maxBid`, so this was taken to be the new interference invariant. A compositionality check proved that this new invariant was compositional. Finally, a k-universal safety check proved that this interference invariant was also adequate.

Quantstamp Assurance Protocol Study. This study evaluates the *Compliance* and *Verification Effectiveness* of SmartACE. The goal of this study is to validate SmartACE in the context of large-scale smart contract development. In this study, we were provided private access to a large specification for the `QSPStaking` contract by Quantstamp. From this specification, 4 properties were selected at random and then verified. Verification required a novel harness, for refinement checking (see [1]). 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 the manual abstraction of unbounded arrays. In total, 10 clauses were required to abstract each array, such as:

- **Clause 1**: The length of array `stakers` is always the length of array `powersOf100`;
- **Clause 2**: Each element of `stakers` is at most the length of `stakers`;
- **Clause 3**: All elements of `powersOf100` are greater than 0.

The abstraction of arrays and discovery of an interference invariant were semi-automated, and aided by counterexamples from SeaHorn. For example, the discovery of **Clause 3** was aided by a counterexample in which 0 appeared spuriously in `powersOf100`, resulting in a division-by-zero error.

Fuzzing Study. This study evaluates the *Fuzzing Effectiveness* of SmartACE. The goal of this study is to determine if local fuzzing is capable of detecting bugs in smart contracts. In this study, faults were injected into smart contracts from Table 9.1. For each fault, 3 000 000 trials of local fuzzing were performed, using 5 users and a timeout of 15 seconds per trial. A benchmark was included in this study if a property-specific fault could be injected without violating the static analysis of SmartACE. It was found that 60 out of 76 faults were discovered by SmartACE. For faults that were missed, a minimal refutation required a sequence of multiple transactions with correlated arguments.

Synthesis Study. This study evaluates the *Synthesis Effectiveness* and *Performance* of SmartACE. The goal of this study is to determine if SeaHorn's predicate synthesis is an appropriate tool for PCMC automation. In this study, predicate synthesis was applied to all compliant smart contracts in Table 9.1. Interference invariants were allowed to depend on all control data. Table 9.1 reports the time required to solve each verification problem. All benchmarks, except for `ICO` and `VUToken`, were solved within 10x of the report VerX time. For 3 benchmarks, predicate synthesis outperform VerX by a speedup of at

least 1.5x. For all unsolved benchmarks, predicate synthesis failed to find either a solution or a counterexample. It is hypothesized that predicate synthesize failed to solve `VUToken` due to the number of control data (21 variables), and `ICO` due to the total number of mapping entries (1 796 entries across 8 mappings). Note that `Fund` was excluded due to a lack of user data, and `QSPStaking` was excluded due to its novel verification harness.

9.2 Results and Discussion

RQ1: Compliance. In the *VerX Verification Study*, SmartACE was applied to 13 real-world smart contracts. With manual abstraction, all but one smart contract was compliant. Most abstractions could be automated, such as identifying patterns in assembly code, supporting dynamic calls, and generating interfaces for calls from EECF contracts. In the *Quantstamp Assurance Protocol Study*, SmartACE was applied to a large-scale, real-world smart contract without prior auditing. The `QSPStaking` contract was compliant after manual abstraction of unbounded arrays through PCMC. In future work, the abstraction of unbounded arrays could be automated. We conclude that the restrictions of MiniSol are reasonable, provided that key abstractions are automated. We suggest that a larger-scale compliance study is required to identify meaningful abstractions.

RQ2a: Verification. In both the *VerX Verification Study* and *Quantstamp Assurance Protocol Study*, SmartACE was able to verify all provided specifications. In the *Auction Study*, SmartACE was also able to verify universal properties. Experience from the *Quantstamp Assurance Protocol Study* shows the overhead of applying SmartACE to a new contract is minimal (i.e., 3 person-days for a novel harness) and can be aided by counterexamples from SeaHorn. We conclude that SmartACE is suitable for high-assurance contracts, and with proper automation, can be integrated into contract development.

RQ2b: Fuzzing. In the *Fuzzing Study*, local fuzzing was shown to perform moderately well by achieving a true positive rate of 79%. The faults that were not detected involved multiple transactions with correlated arguments. Prior work has shown that specialized whitebox fuzzing is often required to find such faults in smart contracts [77, 32]. An interesting question for future work is whether an off-the-shelf whitebox fuzzer from the LLVM-community, such as Klee [13], is capable of detecting these faults.

RQ2c: Synthesis. In the *Synthesis Study*, predicate synthesis verified 85% of all properties. For 10% of unverified properties, the failure was attributed to the total number of mapping entries. For 5% of unverified properties, the failure was attributed to the number of control data. However, the number of mapping entries could be reduced by a more precise `PTGBuilder`, whereas the impact of control data could be reduced by partitioning the signature of an interference invariant. Alternatively, the underlying techniques used to solve predicate synthesis in SeaHorn could be improved (see [29]). We conclude that predicate synthesis is a promising approach to automated smart contract PCMC.

RQ3: Performance. In the *VerX Verification Study*, semi-automated SmartACE outperformed VerX on all compliant benchmarks. On all but two benchmarks, SmartACE achieved a speedup of at least 10x. Therefore, SmartACE is competitive with state-of-the-art techniques. In the *Synthesis Study*, fully-automated SmartACE outperformed VerX on 3 benchmarks. However, the comparison is not entirely fair, since VerX is semi-automated. It is suspected that one bottleneck for SmartACE is the number of users, as each user extends the state space. A more precise `PTGBuilder` would help to reduce the number of users. Upon manual inspect of `Melon` and `Alchemist` (in a single bundle), it was noted that 28% of user state was due to over-approximation. Details on refining `PTGBuilder` can be found in Appendix C. We conclude that SmartACE can scale.

Chapter 10

Conclusion

This thesis has presented an approach to local smart contract analysis. The analysis was enabled in Chapter 3 by defining MiniSol, a subset of Solidity with restricted user interactions and parameterized network semantics. The locality studied in this thesis deals with user interactions during a single smart contract transaction. To this end, Chapter 4 introduced the PT to provide explicit semantics to user interactions, and PTGs to over-approximate all PTs during automated analysis. A detailed study of local transaction symmetries were presented in Chapter 5, resulting in an abstract address domain for MiniSol transactions and the local bundle abstraction for MiniSol smart contracts. In Chapter 6 and Chapter 7 it was shown that local bundle abstractions can be used to ameliorate state explosion in parameterized smart contract verification, and to accelerate counterexample search in bounded analysis. Special attention was given in Chapter 6 to extend PCMC to point-wise aggregate properties through local approximation. To validate this theory in practice, local bundle abstractions were implemented within the Solidity compiler. The experimental results obtained on the VerX and Quantstamp benchmarks show that the SmartACE approach to local smart contract analysis is effective for a sample of real-world smart contracts.

10.1 Related Work

Synchronized Control-User Networks. The use of SCUNs in parameterized verification was first proposed in [23]. The interference and inductive invariants used throughout this paper follow from the application of PCMC to SCUNs in [54]. Both [23] and [54] assume that transition relations are independent of addresses. In this thesis, addresses

can be stored by control process state, and the addresses of participants can be compared, leading to different equivalence classes than those obtained in prior work. The addresses used in this paper, and their restrictions, are similar to the scalarsets of [34]. However, the results of [34] are not directly applicable to SCUNs, and do not allow for addresses to be stored by process state.

Parameterized Compositional Model Checking. PCMC is an abstraction technique that allows for "local," rather than "global" symmetry reductions. Early work on symmetry reduction (e.g., [15, 18]) was "global" in the sense that analysis considered the entire topology of a network, and processes were permuted on this topology. In contrast, PCMC performs all analysis against abstract topologies, known as neighbourhoods [54]. Local symmetry reductions can be exponentially larger than global symmetry reductions [54], and supports all of CTL* (a superset of LTL) [55]. PCMC requires both a network topology and compositional cutoff. Recent work has considered inferring topologies from first-order specifications [5]. In this thesis, topologies are inferred from source code, and compositional cutoffs are avoided by analyzing abstract neighbourhoods directly.

Smart Contract Verification. Most research on smart contract analysis focuses on verifying generic rules, such as freedom from overflow (e.g., [44, 69]), freedom from re-entrancy (e.g., [21, 27, 49]), and correct access patterns (e.g., [12, 56, 70]). A small selection of work has focused on user-provided properties (i.e., [30, 37, 40, 41, 58, 65, 50, 66]). Of these works, only [40] considers smart contracts as parameterized systems. However, in [40], verification is limited to networks of bounded size. In contrast, this thesis verifies networks of arbitrary size through application of PCMC.

Aggregate Properties. Several authors have studied aggregate extensions to first-order logic (e.g., [20, 33, 48]). However, this research is primarily disconnected from smart contract research. For example, the smart contract verification techniques in [30, 58, 65] are limited to ad-hoc solutions for the sum$(\cdot)$ aggregate. In [46], aggregate properties are treated more rigorously, though the results are limited to optimizing runtime checks, and are complementary to this thesis. At the time of writing this thesis, [17] instantiated an aggregate logic for verifying smart contract properties with sum$(\cdot)$. It is currently unclear how [17] relates to local aggregate approximations, and whether this would could be extended to support lattice-based point-wise aggregates.

10.2 Future Work

This section outlines potential directions for future work on local smart contract analysis.

Relaxing MiniSol. The restrictions placed on MiniSol induced the symmetries studied throughout this thesis. However, not all Solidity smart contracts conform to MiniSol. An interesting direction for future work is to study extensions of MiniSol that still permits a finite number of equivalence classes, while also supporting a wider set of Solidity smart contracts. Note that in Chapter 4, the definition of implicit labels was not bound to literal addresses, in order to permit such extensions. One important extension for MiniSol is to support the batch transfer pattern, in which a loop iterates over a list of clients, with only a finite number of these clients participating at any one time (see Appendix D). Another important extension for MiniSol is to support ERC721-like protocols, in which special numeric-typed variables are used as mapping indices (see Appendix E).

Point-Wise Aggregates. This thesis studied the sum$(\cdot)$ and max$(\cdot)$ aggregates in smart contract analysis. These aggregates inspired the notion of point-wise aggregates, and their local approximations. It was shown that the relevant properties of sum$(\cdot)$ and max$(\cdot)$ generalize to point-wise aggregates defined from Abelian groups and lattices, respectively. An interesting topic for future work is to identify other aggregates with point-wise definitions, that are not defined from Abelian groups or lattice. Important questions are whether these new aggregates permit local approximations, and whether these approximations can be used to verify interesting properties.

Refining Local Bundle Abstractions. In Chapter 5, it was shown by Theorem 11 that the users of a MiniSol smart contract can be partitioned into finitely many equivalence classes. However, the analysis in this theorem is coarse-grained, and does not provide a tight upper-bound on the number of equivalence classes. Consequently, the local bundle in Definition 5.4.3 is also coarse-grained, and includes many executions that are not required for the soundness of PCMC in Chapter 6. For example, all non-transient and non-implicit users are symmetric, and the order of their addresses could be fixed a-priori, whereas Definition 5.4.3 considers all possible orderings (see Appendix C for an example). A promising direction for future work is to refine the set of equivalence classes, and to find an encoding of these equivalence classes that is efficient for model checking.

Empirical Evaluations. The scope of evaluations in Chapter 9 was limited by the time required to manually specify program-specific properties. An important future direction for empirical research is to evaluate SmartACE on a wider collection of real-world smart contracts. It is suspected that such a evaluation would be possible by adapting techniques from behavioural simulations to infer program-specific and user-sensitive smart contract properties (e.g., [8]). Note that this study would also require a full specification language and automatic property instrumentation.

References

[1] Martín Abadi and Leslie Lamport. The existence of refinement mappings. In *Proceedings of the Third Annual Symposium on Logic in Computer Science (LICS '88), Edinburgh, Scotland, UK, July 5-8, 1988*, pages 165–175. IEEE Computer Society, 1988.

[2] Parosh Aziz Abdulla, A. Prasad Sistla, and Muralidhar Talupur. Model checking parameterized systems. In Edmund M. Clarke, Thomas A. Henzinger, Helmut Veith, and Roderick Bloem, editors, *Handbook of Model Checking*, pages 685–725. Springer, 2018.

[3] Frances E. Allen. Control flow analysis. In *Proceedings of a Symposium on Compiler Optimization*, pages 1–19, New York, NY, USA, 1970. Association for Computing Machinery.

[4] Sidney Amani, Myriam Bégel, Maksym Bortin, and Mark Staples. Towards verifying ethereum smart contract bytecode in isabelle/hol. In June Andronick and Amy P. Felty, editors, *Proceedings of the 7th ACM SIGPLAN International Conference on Certified Programs and Proofs, CPP 2018, Los Angeles, CA, USA, January 8-9, 2018*, pages 66–77. ACM, 2018.

[5] Rylo Ashmore, Arie Gurfinkel, and Richard J. Trefler. Local reasoning for parameterized first order protocols. In Julia M. Badger and Kristin Yvonne Rozier, editors, *NASA Formal Methods - 11th International Symposium, NFM 2019, Houston, TX, USA, May 7-9, 2019, Proceedings*, volume 11460 of *Lecture Notes in Computer Science*, pages 36–53. Springer, 2019.

[6] Nicola Atzei, Massimo Bartoletti, and Tiziana Cimoli. A survey of attacks on ethereum smart contracts (sok). In Matteo Maffei and Mark Ryan, editors, *Principles of Security and Trust - 6th International Conference, POST 2017, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2017, Uppsala,*

Sweden, April 22-29, 2017, Proceedings, volume 10204 of *Lecture Notes in Computer Science*, pages 164–186. Springer, 2017.

[7] David F. Bacon and Peter F. Sweeney. Fast static analysis of C++ virtual function calls. In Lougie Anderson and James Coplien, editors, *Proceedings of the 1996 ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages & Applications (OOPSLA '96), San Jose, California, USA, October 6-10, 1996*, pages 324–341. ACM, 1996.

[8] Sidi Mohamed Beillahi, Gabriela F. Ciocarlie, Michael Emmi, and Constantin Enea. Behavioral simulation for smart contracts. In Alastair F. Donaldson and Emina Torlak, editors, *Proceedings of the 41st ACM SIGPLAN International Conference on Programming Language Design and Implementation, PLDI 2020, London, UK, June 15-20, 2020*, pages 470–486. ACM, 2020.

[9] Dirk Beyer, Claus Lewerentz, and Frank Simon. Impact of inheritance on metrics for size, coupling, and cohesion in object-oriented systems. In Reiner R. Dumke and Alain Abran, editors, *New Approaches in Software Measurement, 10th International Workshop, IWSM 2000, Berlin, Germany, October 4-6, 2000, Proceedings*, volume 2006 of *Lecture Notes in Computer Science*, pages 1–17. Springer, 2000.

[10] Nikolaj Bjørner and Arie Gurfinkel. Property directed polyhedral abstraction. In Deepak D'Souza, Akash Lal, and Kim Guldstrand Larsen, editors, *Verification, Model Checking, and Abstract Interpretation - 16th International Conference, VMCAI 2015, Mumbai, India, January 12-14, 2015. Proceedings*, volume 8931 of *Lecture Notes in Computer Science*, pages 263–281. Springer, 2015.

[11] Sam Blackshear, Evan Cheng, David L. Dill, Victor Gao, Ben Maurer, Todd Nowacki, Alistair Pott, Shaz Qadeer, Rain, Dario Russi, Stephane Sezer, Tim Zakian, and Runtian Zhou. Move: A language with programmable resources, 2019.

[12] Lexi Brent, Neville Grech, Sifis Lagouvardos, Bernhard Scholz, and Yannis Smaragdakis. Ethainter: a smart contract security analyzer for composite vulnerabilities. In Alastair F. Donaldson and Emina Torlak, editors, *Proceedings of the 41st ACM SIGPLAN International Conference on Programming Language Design and Implementation, PLDI 2020, London, UK, June 15-20, 2020*, pages 454–469. ACM, 2020.

[13] Cristian Cadar, Daniel Dunbar, and Dawson R. Engler. KLEE: unassisted and automatic generation of high-coverage tests for complex systems programs. In Richard

Draves and Robbert van Renesse, editors, *8th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2008, December 8-10, 2008, San Diego, California, USA, Proceedings*, pages 209–224. USENIX Association, 2008.

[14] Adrien Champion, Arie Gurfinkel, Temesghen Kahsai, and Cesare Tinelli. Cocospec: A mode-aware contract language for reactive systems. In Rocco De Nicola and eva Kühn, editors, *Software Engineering and Formal Methods - 14th International Conference, SEFM 2016, Held as Part of STAF 2016, Vienna, Austria, July 4-8, 2016, Proceedings*, volume 9763 of *Lecture Notes in Computer Science*, pages 347–366. Springer, 2016.

[15] Edmund M. Clarke, Somesh Jha, Reinhard Enders, and Thomas Filkorn. Exploiting symmetry in temporal logic model checking. *Formal Methods Syst. Des.*, 9(1/2):77–104, 1996.

[16] Reinhard Diestel. *Graph Theory*. Springer, New York, 2 edition, 2000.

[17] Neta Elad, Sophie Rain, Neil Immerman, Laura Kovács, and Mooly Sagiv. Summing up smart transitions. In Alexandra Silva and K. Rustan M. Leino, editors, *Computer Aided Verification - 33rd International Conference, CAV 2021, Virtual Event, July 20-23, 2021, Proceedings, Part I*, volume 12759 of *Lecture Notes in Computer Science*, pages 317–340. Springer, 2021.

[18] E. Allen Emerson and A. Prasad Sistla. Symmetry and model checking. In Costas Courcoubetis, editor, *Computer Aided Verification, 5th International Conference, CAV '93, Elounda, Greece, June 28 - July 1, 1993, Proceedings*, volume 697 of *Lecture Notes in Computer Science*, pages 463–478. Springer, 1993.

[19] William Entriken, Dieter Shirley, Jacob Evans, and Nastassia Sachs. Eip-721: Erc-721 non-fungible token standard. *Ethereum Improvement Proposals*, no. 721, January 2018. [Online serial]. Available: https://eips.ethereum.org/EIPS/eip-721.

[20] Kousha Etessami. Counting quantifiers, successor relations, and logarithmic space. *J. Comput. Syst. Sci.*, 54(3):400–411, 1997.

[21] Josselin Feist, Gustavo Grieco, and Alex Groce. Slither: a static analysis framework for smart contracts. In *Proceedings of the 2nd International Workshop on Emerging Trends in Software Engineering for Blockchain, WETSEB@ICSE 2019, Montreal, QC, Canada, May 27, 2019*, pages 8–15. IEEE / ACM, 2019.

[22] Joesph A. Gallian. *Contemporary Abstract Algebra*. Cengage Learning, Boston, 9 edition, 2016.

[23] Steven M. German and A. Prasad Sistla. Reasoning about systems with many processes. *J. ACM*, 39(3):675–735, 1992.

[24] Yossi Gilad, Rotem Hemo, Silvio Micali, Georgios Vlachos, and Nickolai Zeldovich. Algorand: Scaling byzantine agreements for cryptocurrencies. *IACR Cryptol. ePrint Arch.*, 2017:454, 2017.

[25] Neville Grech, Michael Kong, Anton Jurisevic, Lexi Brent, Bernhard Scholz, and Yannis Smaragdakis. Madmax: surviving out-of-gas conditions in ethereum smart contracts. *Proc. ACM Program. Lang.*, 2(OOPSLA):116:1–116:27, 2018.

[26] Ilya Grishchenko, Matteo Maffei, and Clara Schneidewind. A semantic framework for the security analysis of ethereum smart contracts. In Lujo Bauer and Ralf Küsters, editors, *Principles of Security and Trust - 7th International Conference, POST 2018, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2018, Thessaloniki, Greece, April 14-20, 2018, Proceedings*, volume 10804 of *Lecture Notes in Computer Science*, pages 243–269. Springer, 2018.

[27] Shelly Grossman, Ittai Abraham, Guy Golan-Gueta, Yan Michalevsky, Noam Rinetzky, Mooly Sagiv, and Yoni Zohar. Online detection of effectively callback free objects with applications to smart contracts. *Proc. ACM Program. Lang.*, 2(POPL):48:1–48:28, 2018.

[28] Arie Gurfinkel, Temesghen Kahsai, Anvesh Komuravelli, and Jorge A. Navas. The seahorn verification framework. In Daniel Kroening and Corina S. Pasareanu, editors, *Computer Aided Verification - 27th International Conference, CAV 2015, San Francisco, CA, USA, July 18-24, 2015, Proceedings, Part I*, volume 9206 of *Lecture Notes in Computer Science*, pages 343–361. Springer, 2015.

[29] Arie Gurfinkel, Sharon Shoham, and Yuri Meshman. Smt-based verification of parameterized systems. In Thomas Zimmermann, Jane Cleland-Huang, and Zhendong Su, editors, *Proceedings of the 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering, FSE 2016, Seattle, WA, USA, November 13-18, 2016*, pages 338–348. ACM, 2016.

[30] Ákos Hajdu and Dejan Jovanovic. solc-verify: A modular verifier for solidity smart contracts. In Supratik Chakraborty and Jorge A. Navas, editors, *Verified Software.*

Theories, Tools, and Experiments - 11th International Conference, VSTTE 2019, New York City, NY, USA, July 13-14, 2019, Revised Selected Papers, volume 12031 of *Lecture Notes in Computer Science*, pages 161–179. Springer, 2019.

[31] Klaus Havelund and Grigore Rosu. Efficient monitoring of safety properties. *Int. J. Softw. Tools Technol. Transf.*, 6(2):158–173, 2004.

[32] Jingxuan He, Mislav Balunovic, Nodar Ambroladze, Petar Tsankov, and Martin T. Vechev. Learning to fuzz from symbolic execution with application to smart contracts. In Lorenzo Cavallaro, Johannes Kinder, XiaoFeng Wang, and Jonathan Katz, editors, *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security, CCS 2019, London, UK, November 11-15, 2019*, pages 531–548. ACM, 2019.

[33] Lauri Hella, Leonid Libkin, Juha Nurmonen, and Limsoon Wong. Logics with aggregate operators. In *14th Annual IEEE Symposium on Logic in Computer Science, Trento, Italy, July 2-5, 1999*, pages 35–44. IEEE Computer Society, 1999.

[34] C. Norris Ip and David L. Dill. Better verification through symmetry. In David Agnew, Luc J. M. Claesen, and Raul Camposano, editors, *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*, volume A-32 of *IFIP Transactions*, pages 97–111. North-Holland, 1993.

[35] Ranjit Jhala, Andreas Podelski, and Andrey Rybalchenko. Predicate abstraction for program verification: Safety and termination. In Edmund M. Clarke, Thomas A. Henzinger, Helmut Veith, and Roderick Bloem, editors, *Handbook of Model Checking*, pages 447–491. Springer, 2018.

[36] Bo Jiang, Ye Liu, and W. K. Chan. Contractfuzzer: fuzzing smart contracts for vulnerability detection. In Marianne Huchard, Christian Kästner, and Gordon Fraser, editors, *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering, ASE 2018, Montpellier, France, September 3-7, 2018*, pages 259–269. ACM, 2018.

[37] Sukrit Kalra, Seep Goel, Mohan Dhawan, and Subodh Sharma. 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.

[38] Theodoros Kasampalis, Dwight Guth, Brandon M. Moore, Traian-Florin Serbanuta, Yi Zhang, Daniele Filaretti, Virgil Nicolae Serbanuta, Ralph Johnson, and Grigore Rosu. IELE: A rigorously designed language and tool ecosystem for the blockchain. In Maurice H. ter Beek, Annabelle McIver, and José N. Oliveira, editors, *Formal Methods - The Next 30 Years - Third World Congress, FM 2019, Porto, Portugal, October 7-11, 2019, Proceedings*, volume 11800 of *Lecture Notes in Computer Science*, pages 593–610. Springer, 2019.

[39] Gary A. Kildall. A unified approach to global program optimization. In Patrick C. Fischer and Jeffrey D. Ullman, editors, *Conference Record of the ACM Symposium on Principles of Programming Languages, Boston, Massachusetts, USA, October 1973*, pages 194–206. ACM Press, 1973.

[40] John Kolb. *A Languge-Based Approach to Smart Contract Engineering*. PhD thesis, University of California at Berkeley, USA, 2020.

[41] Aashish Kolluri, Ivica Nikolic, Ilya Sergey, Aquinas Hobor, and Prateek Saxena. Exploiting the laws of order in smart contracts. In Dongmei Zhang and Anders Møller, editors, *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2019, Beijing, China, July 15-19, 2019*, pages 363–373. ACM, 2019.

[42] Anvesh Komuravelli, Arie Gurfinkel, and Sagar Chaki. Smt-based model checking for recursive programs. In Armin Biere and Roderick Bloem, editors, *Computer Aided Verification - 26th International Conference, CAV 2014, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 18-22, 2014. Proceedings*, volume 8559 of *Lecture Notes in Computer Science*, pages 17–34. Springer, 2014.

[43] Yves Lafont. Towards an algebraic theory of boolean circuits. *Journal of Pure and Applied Algebra*, 184(2):257–310, 2003.

[44] Enmei Lai and Wenjun Luo. Static analysis of integer overflow of smart contracts in ethereum. In *ICCSP 2020: 4th International Conference on Cryptography, Security and Privacy, Nanjing, China, January 10-12, 2020*, pages 110–115. ACM, 2020.

[45] Chris Lattner and Vikram S. Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *2nd IEEE / ACM International Symposium on Code Generation and Optimization (CGO 2004), 20-24 March 2004, San Jose, CA, USA*, pages 75–88. IEEE Computer Society, 2004.

[46] Ao Li, Jemin Andrew Choi, and Fan Long. Securing smart contract with runtime validation. In Alastair F. Donaldson and Emina Torlak, editors, *Proceedings of the 41st ACM SIGPLAN International Conference on Programming Language Design and Implementation, PLDI 2020, London, UK, June 15-20, 2020*, pages 438–453. ACM, 2020.

[47] LibFuzzer—A library for coverage-guided fuzz testing. https://llvm.org/docs/LibFuzzer.html.

[48] Leonid Libkin. Logics with counting, auxiliary relations, and lower bounds for invariant queries. In *14th Annual IEEE Symposium on Logic in Computer Science, Trento, Italy, July 2-5, 1999*, pages 316–325. IEEE Computer Society, 1999.

[49] Loi Luu, Duc-Hiep Chu, Hrishi Olickel, Prateek Saxena, and Aquinas Hobor. Making smart contracts smarter. In Edgar R. Weippl, Stefan Katzenbeisser, Christopher Kruegel, Andrew C. Myers, and Shai Halevi, editors, *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, Vienna, Austria, October 24-28, 2016*, pages 254–269. ACM, 2016.

[50] Matteo Marescotti, Rodrigo Otoni, Leonardo Alt, Patrick Eugster, Antti E. J. Hyvärinen, and Natasha Sharygina. Accurate smart contract verification through direct modelling. In Tiziana Margaria and Bernhard Steffen, editors, *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*, volume 12478 of *Lecture Notes in Computer Science*, pages 178–194. Springer, 2020.

[51] Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system, 2008.

[52] Kedar S. Namjoshi and Richard J. Trefler. Analysis of dynamic process networks. In Christel Baier and Cesare Tinelli, editors, *Tools and Algorithms for the Construction and Analysis of Systems - 21st International Conference, TACAS 2015, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2015, London, UK, April 11-18, 2015. Proceedings*, volume 9035 of *Lecture Notes in Computer Science*, pages 164–178. Springer, 2015.

[53] Kedar S. Namjoshi and Richard J. Trefler. Loop freedom in aodvv2. In Susanne Graf and Mahesh Viswanathan, editors, *Formal Techniques for Distributed Objects, Components, and Systems - 35th IFIP WG 6.1 International Conference, FORTE 2015, Held as Part of the 10th International Federated Conference on Distributed Computing*

Techniques, DisCoTec 2015, Grenoble, France, June 2-4, 2015, Proceedings, volume 9039 of *Lecture Notes in Computer Science*, pages 98–112. Springer, 2015.

[54] Kedar S. Namjoshi and Richard J. Trefler. Parameterized compositional model checking. In Marsha Chechik and Jean-François Raskin, editors, *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*, volume 9636 of *Lecture Notes in Computer Science*, pages 589–606. Springer, 2016.

[55] Kedar S. Namjoshi and Richard J. Trefler. Symmetry reduction for the local mu-calculus. In Dirk Beyer and Marieke Huisman, editors, *Tools and Algorithms for the Construction and Analysis of Systems - 24th International Conference, TACAS 2018, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2018, Thessaloniki, Greece, April 14-20, 2018, Proceedings, Part II*, volume 10806 of *Lecture Notes in Computer Science*, pages 379–395. Springer, 2018.

[56] Ivica Nikolic, Aashish Kolluri, Ilya Sergey, Prateek Saxena, and Aquinas Hobor. 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*, pages 653–663. ACM, 2018.

[57] Susan S. Owicki and David Gries. An axiomatic proof technique for parallel programs I. *Acta Informatica*, 6:319–340, 1976.

[58] Anton Permenev, Dimitar Dimitrov, Petar Tsankov, Dana Drachsler-Cohen, and Martin T. Vechev. Verx: Safety verification of smart contracts. In *2020 IEEE Symposium on Security and Privacy, SP 2020, San Francisco, CA, USA, May 18-21, 2020*, pages 1661–1677. IEEE, 2020.

[59] Nir Piterman and Amir Pnueli. Temporal logic and fair discrete systems. In Edmund M. Clarke, Thomas A. Henzinger, Helmut Veith, and Roderick Bloem, editors, *Handbook of Model Checking*, pages 27–73. Springer, 2018.

[60] Shaz Qadeer and Dinghao Wu. KISS: keep it simple and sequential. In William Pugh and Craig Chambers, editors, *Proceedings of the ACM SIGPLAN 2004 Conference on Programming Language Design and Implementation 2004, Washington, DC, USA, June 9-11, 2004*, pages 14–24. ACM, 2004.

[61] Zvonimir Rakamaric and Michael Emmi. SMACK: decoupling source language details from verifier implementations. In Armin Biere and Roderick Bloem, editors, *Computer*

Aided Verification - 26th International Conference, CAV 2014, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 18-22, 2014. Proceedings, volume 8559 of *Lecture Notes in Computer Science*, pages 106–113. Springer, 2014.

[62] Steven Roman. *Lattices and Ordered Sets*. Springer, New York, 1 edition, 2008.

[63] Ilya Sergey, Vaivaswatha Nagaraj, Jacob Johannsen, Amrit Kumar, Anton Trunov, and Ken Chan Guan Hao. Safer smart contract programming with scilla. *Proc. ACM Program. Lang.*, 3(OOPSLA):185:1–185:30, 2019.

[64] Joseph Sifakis. Property preserving homomorphisms of transition systems. In Edmund M. Clarke and Dexter Kozen, editors, *Logics of Programs, Workshop, Carnegie Mellon University, Pittsburgh, PA, USA, June 6-8, 1983, Proceedings*, volume 164 of *Lecture Notes in Computer Science*, pages 458–473. Springer, 1983.

[65] Sunbeom So, Myungho Lee, Jisu Park, Heejo Lee, and Hakjoo Oh. 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*, pages 1678–1694. IEEE, 2020.

[66] Jon Stephens, Kostas Ferles, Benjamin Mariano, Shuvendu Lahiri, and Isil Dillig. Smartpulse: Automated checking of temporal properties in smart contracts. In *42nd IEEE Symposium on Security and Privacy*. IEEE, 2021.

[67] Nick Szabo. Smart contracts: Building blocks for digital markets, 1996.

[68] David A. Terei and Manuel M. T. Chakravarty. An llvm backend for GHC. In Jeremy Gibbons, editor, *Proceedings of the 3rd ACM SIGPLAN Symposium on Haskell, Haskell 2010, Baltimore, MD, USA, 30 September 2010*, pages 109–120. ACM, 2010.

[69] Christof Ferreira Torres, Julian Schütte, and Radu State. Osiris: Hunting for integer bugs in ethereum smart contracts. In *Proceedings of the 34th Annual Computer Security Applications Conference, ACSAC 2018, San Juan, PR, USA, December 03-07, 2018*, pages 664–676. ACM, 2018.

[70] Petar Tsankov, Andrei Marian Dan, Dana Drachsler-Cohen, Arthur Gervais, Florian Bünzli, and Martin T. Vechev. Securify: Practical security analysis of smart contracts. In David Lie, Mohammad Mannan, Michael Backes, and XiaoFeng Wang, editors, *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security, CCS 2018, Toronto, ON, Canada, October 15-19, 2018*, pages 67–82. ACM, 2018.

[71] Fabian Vogelsteller and Vitalik Buterin. Eip-20: Erc-20 token standard. *Ethereum Improvement Proposals*, no. 20, November 2015. [Online serial]. Available: https://eips.ethereum.org/EIPS/eip-20.

[72] Scott Wesley, Maria Christakis, Jorge A. Navas, Richard J. Trefler, Valentin Wüstholz, and Arie Gurfinkel. Compositional verification of smart contracts through communication abstraction (extended). *CoRR*, abs/2107.08583, 2021.

[73] Gavin Wood. Ethereum: A secure decentralised generalised transaction ledger, 2014.

[74] Jingyue Wu, Artem Belevich, Eli Bendersky, Mark Heffernan, Chris Leary, Jacques A. Pienaar, Bjarke Roune, Rob Springer, Xuetian Weng, and Robert Hundt. gpucc: an open-source GPGPU compiler. In Björn Franke, Youfeng Wu, and Fabrice Rastello, editors, *Proceedings of the 2016 International Symposium on Code Generation and Optimization, CGO 2016, Barcelona, Spain, March 12-18, 2016*, pages 105–116. ACM, 2016.

[75] Valentin Wüstholz and Maria Christakis. Harvey: a greybox fuzzer for smart contracts. In Prem Devanbu, Myra B. Cohen, and Thomas Zimmermann, editors, *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*, pages 1398–1409. ACM, 2020.

[76] Michal Zalewski. Technical "whitepaper" for AFL. http://lcamtuf.coredump.cx/afl/technical_details.txt.

[77] William Zhang, Sebastian Banescu, Leonardo Pasos, Steven T. Stewart, and Vijay Ganesh. Mpro: Combining static and symbolic analysis for scalable testing of smart contract. In Katinka Wolter, Ina Schieferdecker, Barbara Gallina, Michel Cukier, Roberto Natella, Naghmeh Ramezani Ivaki, and Nuno Laranjeiro, editors, *30th IEEE International Symposium on Software Reliability Engineering, ISSRE 2019, Berlin, Germany, October 28-31, 2019*, pages 456–462. IEEE, 2019.

[78] Yan Zhang, Junwen Zhang, Dalin Zhang, and Yongmin Mu. Survey of directed fuzzy technology. In *2018 IEEE 9th International Conference on Software Engineering and Service Science (ICSESS)*, pages 696–699. IEEE, 2018.

APPENDICES

Appendix A

Multidimensional Mappings

This section presents extensions of MiniSol for multidimensional maps. Results from previous sections (i.e., Chapter 4 and Chapter 5) are generalized to this setting. Fig. A.1 is used as a running example. In Fig. A.1, the `SimpleToken` contract implements a simplified version of the ERC-20 protocol [71].

The ERC-20 protocol is used to implement a token-based currency. In a token-based currency, users have tokens (i.e., digital money) that can be transferred from one user to another. A user may also approve another user to move tokens on their behalf. In Fig. A.1, the one-dimensional mapping, `balances`, is used to record tokens and the two dimensional mapping, `allowances`, is used to record approvals. For simplicity, only the approval (`approve`) and delegate transfer (`transferFrom`) methods are implemented.

Impact on Semantics In a one-dimensional mapping, each user maintains a single value from $\mathbb{D}$. For instance, each user of `SimpleToken` maintains a single token count in `balances`. Therefore, the size of each user's data in a one-dimensional mapping is independent of the network size. This changes in the case of a multidimensional mapping. As an example, each user of `SimpleToken` maintains an allowance for every other user of the smart contract. Therefore, the size of each user's data in a two-dimensional mapping is $O(N)$, where N is the network size. Consequently, $\mathsf{user}(\mathcal{C}, \mathcal{A}) := \left(\mathcal{A} \times \mathbb{D} \times \mathbb{D}^{|\mathcal{A}|}\right)$. In general, each user stores $O(N^{k-1})$ values in a k-dimensional mapping.

Impact on Concrete Regions Recall a concrete region $\gamma \subseteq \mathsf{control}(\mathcal{C}, [N]) \times [N]$ from Section 5.4. If $(a, c) \notin \gamma$, then the user with address a is abstract from control state c. Alternatively, the address a can represent arbitrarily many users from control state c. Therefore,

```
/// @title A simplified ERC20 protocol.
/// @dev Ignores overflow for simplicity
contract SimpleToken {
  mapping(address => uint) balances;
  mapping(address => mapping(address => uint))
    allowances;

  /// Allows spender to transfer v tokens from sender.
  function approve(address spender, uint v)
    public returns (bool) {
    allowances[msg.sender][spender] = v;
  }

  /// If approved, moves v tokens from s(rc) to d(st).
  function transferFrom(address s, address d, uint v)
    public returns (bool) {
    require(allowances[s][msg.sender] >= v);
    require(balances[s] > v);

    allowances[s][msg.sender] -= v;
    balances[s] -= v;
    balances[d] += v;
  }
}
```

Figure A.1: A simplified implementation of the ERC-20 protocol [71].

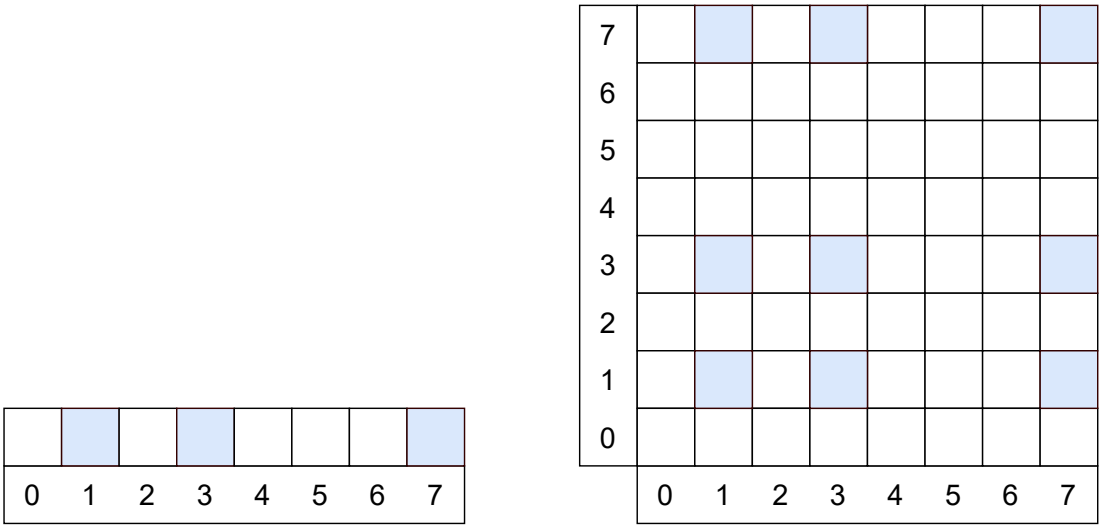


Figure A.2: An illustration of two-dimensional concretization. The network size is 8, control state is c, and $\gamma = \{(c, 1), (c, 3), (c, 7)\}$. The mappings are `balances` and `allowances`, respectively. A cell is shaded if the corresponding mapping entry is concrete.

an entry in a k-dimensional mapping is concrete from control state c, if and only if all k indices appear in γ with c. As a concrete example, assume that `transferFrom` is called with `msg.sender` set to a_0 and `s` set to a_1 from control state c. Then `allowance[s][msg.sender]` is concrete if and only if $(c, a_0) \in \gamma$ and $(c, a_1) \in \gamma$. An illustration is given in Fig. A.2.

Appendix B

Auction LTS Example

This section provides the full LTS definition for `Auction` smart contract in Fig. 3.1. This means that $\mathcal{C} = (\texttt{Auction})$ and $\mathbb{F} = \{\texttt{constructor}, \texttt{bid}, \texttt{withdraw}, \texttt{stop}\}$. To ensure that the constructor is only called once, a Boolean `constructed` flag is added to $\mathcal{C}$. As in Section 3.1.2, let $\mathcal{A}$ be an address space.

The control states are $\mathsf{control}(\mathcal{C}, \mathcal{A}) \subseteq (\mathcal{A} \times \mathbb{D}^6)$ such that if $c \in \mathsf{control}(\mathcal{C}, \mathcal{A})$, then:

1. $\mathsf{role}(c, 0)$ is the address of `manager`,
2. $\mathsf{data}(c, 0)$ is `block.number`,
3. $\mathsf{data}(c, 1)$ is `block.timestamp`,
4. $\mathsf{data}(c, 2)$ is `constructed`.
5. $\mathsf{data}(c, 3)$ is the balance of the `Auction` contract,
6. $\mathsf{data}(c, 4)$ is `leadingBid`,
7. $\mathsf{data}(c, 5)$ is `stopped`.

The user states are $\mathsf{user}(\mathcal{C}, \mathcal{A}) \subseteq (\mathcal{A} \times \mathbb{D})$ such that if $u \in \mathsf{user}(\mathcal{C}, \mathcal{A})$, then $\mathsf{map}(u)_0$ is the bid of user u.

The inputs are $\mathsf{inputs}(\mathcal{C}, \mathcal{A}) \subseteq (\mathbb{F} \times \mathcal{A} \times \mathbb{D}^3)$ such that if $p = (fn, \mathbf{x}, \mathbf{y}) \in \mathsf{inputs}(\mathcal{C}, \mathcal{A})$:

1. $\mathsf{client}(p, 0)$ is `msg.sender`;

2. If $fn =$ `constructor`, then $\mathsf{client}(p, 1)$ is `manager`, else $\mathsf{client}(p, 1)$ is unused;

3. $\mathsf{arg}(p, 0)$ is `msg.value`;

4. $\mathsf{arg}(p, 1)$ is the next value of `block.number`;

5. $\mathsf{arg}(p, 2)$ is the next value of `block.timestamp`.

Fix $N \in \mathbb{N}$. For the rest of this section, the discussion is restrict to $\mathsf{global}(\mathcal{C}, N)$. This means that $\mathcal{A} = [N]$, and that $\mathcal{M}$ is the identify function for $[N]$. The rest of this section presents transaction semantics for $f = [\![\mathcal{C}]\!]_{\mathcal{M}}$. It is assumed that $N \geq 2$, so that the zero-account and `Auction` account are defined and unique.

To simplify presentation, predicates are used to factor out common guards and expressions. The predicate $\mathsf{ValidSender}(p, \mathbf{u})$ guards against invalid senders. It is true whenever the sender is not the zero-account (i.e., $\mathsf{client}(p, 0) \neq \mathsf{id}(\mathbf{u}_0)$) nor the `Auction` account (i.e., $\mathsf{client}(p, 0) \neq \mathsf{id}(\mathbf{u}_1, 0)$. An important observation is that the sender, $\mathsf{client}(p, 0)$, is compared to $\mathsf{id}(\mathbf{u}_i)$, rather than i. This is because $\mathcal{M}$ is used to map all literal addresses to users. For the first case, $\mathcal{M}$ first maps 0 to $\mathbf{u}_0$, and then $\mathsf{id}(\mathbf{u}_0)$ maps $\mathbf{u}_0$ back to 0[1].

The predicate $\mathsf{Constructed}(s)$ is used to guard multiple constructor calls. It is true once `Auction` has been constructed. That is, $\mathsf{data}(s, 2) = 1$.

The predicate $\mathsf{TimeCheck}(s, s', p)$ is used to enforce monotonic time and lockstep clocks. The deterministic progress of time is enforced by $\mathsf{data}(s', 0) = \mathsf{arg}(p, 1)$ and $\mathsf{data}(s', 1) = \mathsf{arg}(p, 2)$. The monotonicity of `block.number` is enforced by $\mathsf{data}(s', 0) \geq \mathsf{data}(s, 0)$. Similarly, the monotonicity of `block.timestamp` is enforced by $\mathsf{data}(s', 1) \geq \mathsf{data}(s, 1)$. Lockstep clocks are then enforced by $\mathsf{data}(s', 0) = \mathsf{data}(s, 0) \iff \mathsf{data}(s', 1) = \mathsf{data}(s, 1)$.

Finally, $\mathsf{KeepRole}(s, s', i)$ $\mathsf{KeepData}(s, s', j)$ and are used to enforce that the i-th role and j-th datum are unchanged. Formally, $\mathsf{KeepRole}(s, i) := \mathsf{role}(s', i) = \mathsf{role}(s, i)$ and $\mathsf{KeepData}(s, s', j) := \mathsf{data}(s', j) = \mathsf{data}(s, j)$.

$$f((s, \mathbf{u}), p) = \begin{cases} (s, \mathbf{u}) & \text{if } \neg\mathsf{ValidSender}(p, \mathbf{u}) \\ g_0((s, \mathbf{u}), p) & \text{if } fn = \texttt{constructor} \\ g_1((s, \mathbf{u}), p) & \text{if } fn = \texttt{stop} \\ g_2((s, \mathbf{u}), p) & \text{if } fn = \texttt{bid} \\ g_3((s, \mathbf{u}), p) & \text{if } fn = \texttt{withdraw} \end{cases}$$

[1] This indirection allows for users to be readdressed in Definition 4.1.3

$$g_0((s,\mathbf{u}),p) = \begin{cases} (s,\mathbf{u}) & \text{if } \neg\mathsf{Constructed}(s) \\ (s,\mathbf{u}) & \text{if } \mathsf{client}(p,1) = \mathsf{id}(\mathbf{u}_0) \\ (s',\mathbf{u}) & \text{else, s.t. } \mathsf{TimeCheck}(s,s',p),\ \mathsf{KeepData}(s,s',3),\ \mathsf{KeepData}(s,s',4), \\ & \mathsf{KeepData}(s,s',5),\ \mathsf{role}(s',0) = \mathsf{client}(p,1),\ \text{and } \mathsf{data}(s',2) = 1 \end{cases}$$

$$g_1((s,\mathbf{u}),p) = \begin{cases} (s,\mathbf{u}) & \text{if } \mathsf{Constructed}(s) \\ (s,\mathbf{u}) & \text{if } \mathsf{role}(s,0) \neq \mathsf{client}(p,0) \\ (s',\mathbf{u}) & \text{else, s.t. } \mathsf{TimeCheck}(s,s',p),\ \mathsf{KeepRole}(s,s',0),\ \mathsf{KeepData}(s,s',2), \\ & \mathsf{KeepData}(s,s',3),\ \mathsf{KeepData}(s,s',4),\ \text{and } \mathsf{data}(s',5) = 1, \end{cases}$$

$$g_2((s,\mathbf{u}),p) = \begin{cases} (s,\mathbf{u}) & \text{if } \mathsf{Constructed}(s) \\ (s,\mathbf{u}) & \text{if } \mathsf{role}(s,0) = \mathsf{client}(p,0) \\ (s,\mathbf{u}) & \text{if for } i = \mathsf{client}(p,0), \mathsf{data}(s,4) \leq \mathsf{map}(\mathbf{u}_i)_0 \wedge \mathsf{map}(\mathbf{u}_i)_0 \neq 0 \\ (s,\mathbf{u}) & \text{if for } i = \mathsf{client}(p,0), \mathsf{data}(s,4) \geq \mathsf{arg}(p,0) + \mathsf{map}(\mathbf{u}_i)_0 \\ (s,\mathbf{u}) & \text{if } \mathsf{data}(s,5) = 1 \\ (s',\mathbf{u}') & \text{else, s.t. } \mathsf{TimeCheck}(s,s',p),\ \mathsf{KeepRole}(s,s',0),\ \mathsf{KeepData}(s,s',2), \\ & \mathsf{KeepData}(s,s',4),\ \mathsf{KeepData}(s,s',5), \\ & \mathsf{data}(s',3) = \mathsf{data}(s,3) + \mathsf{arg}(p,0),\ \text{and } \forall i \in [N]\cdot \\ & \quad \mathsf{id}(\mathbf{u}'_i) = \mathsf{id}(\mathbf{u}_i) \wedge \\ & \quad i \neq \mathsf{client}(p,0) \Rightarrow \mathsf{map}(\mathbf{u}'_i) = \mathsf{map}(\mathbf{u}_i) \wedge \\ & \quad i = \mathsf{client}(p,0) \Rightarrow \mathsf{map}(\mathbf{u}'_i)_0 = \mathsf{arg}(p,0) + \mathsf{map}(\mathbf{u}_i)_0 \end{cases}$$

$$g_3((s,\mathbf{u}),p) = \begin{cases} (s,\mathbf{u}) & \text{if } \mathsf{Constructed}(s) \\ (s,\mathbf{u}) & \text{if } \mathsf{role}(s,0) = \mathsf{client}(p,0) \\ (s,\mathbf{u}) & \text{if } \mathsf{data}(s,3) \neq \mathsf{map}(\mathbf{u}_{x_1})_0 \\ (s',\mathbf{u}') & \text{else, s.t. } \mathsf{TimeCheck}(s,s',p),\ \mathsf{KeepRole}(s,s',0),\ \mathsf{KeepData}(s,s',2), \\ & \mathsf{KeepData}(s,s',4),\ \mathsf{KeepData}(s,s',5), \\ & \mathsf{data}(s',3) = \mathsf{data}(s,3) - \mathsf{data}(\mathbf{u}'_i),\ \text{and } \forall i \in [N]\cdot \\ & \quad \mathsf{id}(\mathbf{u}'_i) = \mathsf{id}(\mathbf{u}_i) \wedge \\ & \quad i \neq x_1 \Rightarrow \mathsf{data}(\mathbf{u}'_i) = \mathsf{data}(\mathbf{u}_i) \wedge \\ & \quad i = x_1 \Rightarrow \mathsf{data}(\mathbf{u}'_i) = \mathbf{0} \end{cases}$$

Recall from Section 3.1.2 that a reverted transaction (such as a failed require statement)

is treated as a no-op. In f, these no-ops correspond to the cases that send $(s, \mathbf{u})$ to $(s, \mathbf{u})$. Most of these cases, such as the second case of g_2, are derived directly from the source code (Line 27 for this case). Other cases correspond to more general guards, such as $\mathsf{ValidSender}(p, \mathbf{u})$ and $\mathsf{Constructed}(s)$.

Note that $\mathsf{Constructed}(s)$ (or its negation) appears in every method. The negation in g_0 guards against calls to `constructor` after the constructor has already the called. Other occurrences of $\mathsf{Constructed}(s)$ guard against calls to non-constructor functions before the constructor has been called. Collective, these guards ensure that `Auction` is constructed once and only once, during the first transaction.

Appendix C

Address Order Reductions

This appendix outlines preliminary results on refining local bundle abstractions. Let $(S, P, f, s_0) = \mathsf{local}(\mathcal{C}, N, \gamma_\perp, \theta)$, where θ is an interference invariant for $\mathcal{C}$.

C.1 Fixing the Order of Transient Participants

This section briefly shows that (S, P, f, s_0) is bisimulated by a local bundle that fixes the address of each non-implicit, transient participant. The proof sketch is given for one role. First, it is shown that for any $(c, \mathbf{u}) \in S$, a permutation can be constructed on-the-fly that ensures a non-implicit, transient participant has a predefined address. Second, it is shown that due to the interference relation, on-the-fly relabelling permits a bisimulation relation. This section does not provide an explicit construction of the permutation, and does not consider impacts on the implementation of SmartACE.

Let $(c, \mathbf{u}) \in S$ and $i \in [N] \backslash \mathsf{lits}(\mathcal{C})$. Assume that $\mathsf{data}(c, 0) \notin \mathsf{lits}(\mathcal{C})$. Let $\tau : [N] \to [N]$ be the permutation that swaps i with $\mathsf{data}(c, 0)$. Then $\forall a \in [N] \backslash \{i, \mathsf{data}(c, 0)\} \cdot \tau(a) = a$. Since $i \notin \mathsf{lits}(\mathcal{C})$ and $\mathsf{data}(c, 0) \notin \mathsf{lits}(\mathcal{C})$, then $\forall a \in \mathsf{lits}(\mathcal{C}) \cdot \tau(a) = a$. Furthermore, $\mathsf{data}(\mathsf{rename}(\mathcal{C}, \tau, c), 0) = i$. Then τ is a permutation that preserves implicit participation, and fixes the address of the first non-implicit, transient participant.

The existence of a bisimulation under input relabelling is briefly justified. If τ is an on-the-fly permutation, then $\mathsf{rename}(\mathcal{C}, \tau, s)$ is the bisimulation relation, and $\mathsf{rename}(\mathcal{C}, \tau, p)$ is the input relabelling. The initial state satisfies $s_0 = \mathsf{rename}(\mathcal{C}, \tau, s_0)$ for any choice of τ, since s_0 is zero-initialized and τ preserves address 0. From the results of Section 5.2, each transition satisfies the bisimulation under input relabelling, provided that the pre-states

are data-equivalent. In this setting, there is always a data-equivalence state, due to the interference of $(\gamma_\perp, \theta)$. This concludes the proof sketch.

C.2 Fixing the Order of Explicit Participants

It is hypothesized by the authors that analysis similar to Appendix C.1 can be applied to non-implicit, non-transient, explicit participants. In this case, the permutations are applied to the actions rather than the control states. Intuitively, the permutation would restrict the addresses for each client. It is unclear to the authors how coarse these restrictions must be to ensure that equivalence classes are not lost.

Appendix D

Extending to Batch Transfers

Batch transfers are used in many smart contracts. A batch transfer allows an action, such as transferring tokens, to be executed against a list of users. In general, batch transfers are not necessary. However, they are commonly used, as grouping together multiple transactions allows users to save on gas. An example of the batch transfer pattern is given in Fig. D.1.

The batch transfer pattern is not supported in MiniSol, as it requires passing an array of users to a smart contract function. This restriction exists, as it ensures that the number of explicit users is bounded. However, notice that in Fig. D.1, the array is only accessed for within the batch transfer loop. On each iteration of the loop, a single client from the array temporarily interacts with the transaction. This suggests a more general notion of transaction locality than the ones studied within this thesis. It seems to the author that batch transfers are a special case, and that this locality should generalize to k temporary clients during a loop iteration.

```
1 /// @title An example of the batch transfer pattern.
2 contract Example {
3   mapping(address => uint) balances;
4
5   // Single transfer.
6   function transferTo(address to, uint v) public {
7     require(balances[msg.sender] > v);
8
9     balances[msg.sender] -= v;
10    balances[to] += v;
11  }
12
13  // Batching pattern.
14  function batchTransfer(address[] as, uint[] vs) public {
15    // (1) Checks that input arrays are of the same length.
16    require(as.length == vs.length);
17
18    // (2) Iterates over input arrays, visiting a single element during each iteration.
19    for (uint i = 0; i < as.length; ++i) {
20      // (3) Invokes an action on the array element.
21      transferTo(as[i], vs[i]);
22    }
23  }
24 }
```

Figure D.1: An example of the batch transfer pattern.

Appendix E

Extending to ERC-721

This section briefly outlines the ERC-721 protocol, and the extensions required to support it in MiniSol. The ERC-721 protocol was first proposed in [19], and has gained recent attention for decentralized property management. A simplified implementation of the ERC-721 protocol is given in Fig. E.1.

In the ERC-721 protocol, each assert is represented by a token with a unique 256-bit identifier. To management the ownership of tokens, and ERC-721 implementation, such as in Fig. E.1, must map integer-type identifiers to address-typed owners.

This violates the syntax of MiniSol is two ways. First, the index-type of a mapping must be of an address-type. Second, the value-type of a mapping must be of an integer-type. The first restriction ensures that mapping indices are not modified. The second restriction prevents a topology in which each user has a successor (an address-to-address mapping is a successor relation). However, neither issue is true of Fig. E.1.

First, observe that if an integer-typed variable is used to index a mapping, then the variable is never subjected arithmetic operations. This is because token identifiers and user addresses are both nominal types. The MiniSol language could overcome this limitation by introducing a full type-system with inference for nominal types. Then mappings could be generalized to allow indices of any nominal type. Note that each nominal type would require a parameter in the network semantics of MiniSol, and representation in participation topologies.

Second, observe that each mapping with an address-typed value has an integer-typed index. Clearly, these mappings cannot represent successor relations. However, they do represent injections from assets with integer-typed identifiers to smart contract users. The

```
/// @title A simplified ERC721 protocol.
/// @dev Ignores overflow for simplicity
contract SimpleERC721 {
  mapping(address => uint) balances;
  mapping(uint => address) owners;

  // Returns the owner of the asset with ID uid.
  function ownerOf(uint uid) public returns (address) {
    return owners[uid];
  }

  // Transfers assert with ID uid to d.
  function transfer(address d, uint uid) public {
    require(ownerOf(uid) == msg.sender);

    balances[msg.sender] -= 1;
    balances[d] += 1;

    owners[uid] = d;
  }
}
```

Figure E.1: A simplified implementation of the ERC-20 protocol [19].

MiniSol language could overcome this limitation by allowing mappings between distinct nominal identifiers[1]. The injections would be reflected in the participation topology.

[1]It is not clear to the author what the right notion of "distinct" should be here. Distinguishing nominal identifiers from base types seems too coarse-grained in general.