

# Reactive Synthesis of Smart Contract Control Flows

Bernd Finkbeiner<sup>1</sup>[0000-0002-4280-8441], Jana Hofmann<sup>2</sup>[0000-0003-1660-2949],  
Florian Kohn<sup>1</sup>[0000-0001-9672-2398], and Noemi Passing<sup>1</sup>[0000-0001-7781-043X]

<sup>1</sup> CISPA Helmholtz Center for Information Security, Saarbrücken, Germany  
{finkbeiner,florian.kohn,noemi.passing}@cispa.de

<sup>2</sup> Microsoft Azure Research, Cambridge, UK  
t-jhofmann@microsoft.com

**Abstract.** Smart contracts are small but highly error-prone programs that implement agreements between multiple parties. We present a reactive synthesis approach for the automatic construction of smart contract state machines. Towards this end, we extend temporal stream logic (TSL) with universally quantified parameters over infinite domains. Parameterized TSL is a convenient logic to specify the temporal control flow, i.e., the correct order of transactions, as well as the data flow of the contract’s fields. We develop a two-step approach that 1) synthesizes a finite representation of the – in general – infinite-state system and 2) splits the system into a compact hierarchical architecture that enables the implementation of the state machine in Solidity. We implement the approach in our prototype tool SCSYNT, which – within seconds – automatically constructs Solidity code that realizes the specified control flow.

**Keywords:** Reactive Synthesis · Smart Contracts · Temporal Stream Logic.

**Acknowledgements.** This work was supported by the European Research Council (ERC) Grant HYPER (No. 101055412) and by DFG grant 389792660 as part of TRR 248.

## 1 Introduction

Smart contracts are small programs that implement digital contracts between multiple parties. They are deployed on the blockchain and thereby remove the need for a trusted third party that enforces a correct execution of the contract. Recent history, however, has witnessed numerous bugs in smart contracts, some of which led to substantial monetary losses. One critical aspect is the implicit state machine of a contract: to justify the removal of a trusted third party – a major selling point for smart contracts – all parties must trust that the contract indeed enforces the agreed order of transactions.

Formal methods play a significant role in the efforts to improve the trustworthiness of smart contracts. Indeed, the *code is law* paradigm is shifting towards

a *specification is law* paradigm [1]. Formal verification has been successfully applied to prove the correctness of the implicit state machine of smart contracts, for example, by verifying the contract against temporal logic specifications [39,44,36] or a given state machine [46]. Other approaches model the control flow with state machines and construct Solidity code from it [47,32,6,29]. Synthesis, i.e., the automatic construction of Solidity code *directly* from a temporal specification, has hardly been studied so far (except for a first step [45], see related work).

In this paper, we study the synthesis of smart contracts state machines from temporal stream logic (TSL), which we equip with universally quantified parameters. TSL extends linear-time temporal logic (LTL) with data cells and uninterpreted functions and predicates. These features enable us to reason about the order of transactions as well as the data flow of the contract’s fields. To distinguish method calls from different callers, we extend the logic with universally quantified parameters. For example, the following parameterized TSL formula expresses that every voter can only vote once and that a field `numVotes` is increased with every vote.

$$\forall m. \square(\text{vote}(m) \rightarrow [\text{numVotes} \leftarrow \text{numVotes} + 1]) \wedge \bigcirc \square \neg \text{vote}(m)$$

The above formula demonstrates the challenges associated with parameterized TSL synthesis. First of all, a part of the formula restricts the allowed method calls, which are inputs in the synthesis problem. To make specifications realizable, we restrict ourselves to safety properties, which we express in the past-time fragment of parameterized TSL. Second, as the contract might interact with arbitrarily many voters, the above formula ranges over an infinite domain. However, we need to find a finite representation of the system that can be translated into feasible Solidity code.

We tackle this challenge in two steps. First, we translate the parameterized pastTSL formula to pastTSL to synthesize a finite representation of the system. Unfortunately, we show that the realizability problem of pastTSL is undecidable, even without parameters. As a remedy, we employ a sound approximation in LTL [13] to make synthesis possible.

In a second step, we split the resulting state machine into a hierarchical structure of smaller, distributed state machines. This architecture can be interpreted as an infinite-state system realizing the original parameterized formula and also minimizing the number of transactions needed to keep the system up to date at runtime.

We implement the approach in our prototype `SCSYNT`<sup>3</sup>, which, due to the past-time fragment, leverages efficient symbolic algorithms. We specify ten different smart contract specifications and obtain an average synthesis time of two seconds. Our largest specification is based on Avolab’s NFT auction [2] and produces a state machine with 12 states in 12 seconds. To summarize, we

- show how to specify smart contract control flows in parameterized pastTSL,
- prove undecidability of the general realizability problem of pastTSL,

<sup>3</sup> The artifact is available at: <https://1drv.ms/u/s!AuQiCoSjTkXChcYzynYNw-NQ8pjkzw?e=Di3BCs>

- and present a sound (but necessarily incomplete) synthesis approach for parameterized pastTSL formulas that generates a hierarchy of state machines to enable a compact representation of the system in Solidity.

*Related Work.* Formal approaches for smart contracts range from the automatic construction of contracts from state machines [32,33], over the verification against temporal logics [39,44,36] and state machines [46,26], to deductive verification approaches [8,21]. Closest to our work is a synthesis approach based on LTL specifications [45]. The approach does not reason about the contract’s data: neither about the current value of the fields, nor about parameters like the method’s caller. To quote the authors of [45]: the main challenge in the synthesis of smart contracts is “how to strike a balance between simplicity and expressivity [...] to allow effective synthesis of practical smart contracts”. In this paper, we opt for a more expressive temporal logic and simultaneously aim to keep the specifications readable.

TSL has been successfully applied to synthesize FPGA controllers [16] and functional reactive programs [12]. To include domain-specific reasoning, TSL has been extended with theories [10] and SMT solvers [30]. A recent approach combines TSL reactive synthesis with SyGus to synthesize implementations for TSL’s uninterpreted functions [5]. Parameterized synthesis has so far focused on distributed architectures parameterized in the number of components [20,22,23,31]. Orthogonal to this work, these approaches rely on a reduction to bounded isomorphic synthesis [20,22,23] or apply a learning-based approach [31].

*Overview.* We first provide some brief preliminaries on state machines, reactive synthesis, and TSL. In Section 3, we introduce parameterized TSL and demonstrate how it can be used for specifying smart contract control flows. Subsequently, we discuss the high-level idea and associated challenges of our synthesis approach in Section 4 and discuss synthesis from plain pastTSL in Section 5. We demonstrate how to specify smart contracts using pure pastTSL and prove the undecidability of its realizability problem. We proceed with the main part of the approach, a splitting algorithm for state machines, in Section 6. Finally, we discuss the implementation of SCSYNT and its evaluation in Section 7.

## 2 Preliminaries

We assume familiarity with linear-time temporal logic (LTL). A definition with past-time temporal operators can be found in [17,7]. We only assume basic knowledge about smart contracts; for an introduction we refer to [9].

### 2.1 State Machines, Safety Properties and Reactive Synthesis

We give a brief introduction to Mealy machines, safety properties, and reactive synthesis. In this work, we represent smart contract control flows as *Mealy state machines* [35], which separate the alphabet into inputs  $I$  and outputs  $O$ . A

Mealy machine  $\mathcal{M}$  is a tuple  $(S, s_0, \delta)$  of states  $S$ , initial state  $s_0$ , and transition relation  $\delta \subseteq S \times I \cup O \times S$ . For a compact representation, we attach the outputs also to transitions, not to the states. We call  $\mathcal{M}$  *finite-state* if both  $\Sigma = I \cup O$  and  $S$  are finite, and *infinite-state* otherwise. An infinite sequence  $t \in \Sigma^\omega$  is a *trace* of  $\mathcal{M}$  if there is an infinite sequence of states  $r \in S^\omega$  such that  $r[0] = s_0$  and  $(r[i], t[i], r[i+1]) \in \delta$  for all points in time  $i \in \mathbb{N}$ . A finite sequence of states  $r \in S^+$  results in a finite trace  $t \in \Sigma^+$ .

In this paper, we work with specifications that are *safety properties* (see, e.g., [25,27]). A safety property can be equivalently expressed as a Mealy machine  $\mathcal{M}$  that describes the set of traces that satisfy the property ( $\mathcal{M}$  is called the *safety region* of the property). For a safety specification, the *reactive synthesis* problem is to determine the *winning region*, i.e., the maximal subset of its safety region such that for every combination of state and input, there is a transition into said subset. A *strategy* is a subset of the winning region such that in each state, there is exactly one outgoing transition for every input.

## 2.2 Past-time Temporal Stream Logic

PastTSL is the past-time variant of TSL [13], a logic that extends LTL with cells that can hold data from a possibly infinite domain. To abstract from concrete data, TSL includes uninterpreted functions and predicates. *Function terms*  $\tau_f \in \mathcal{T}_F$  are recursively defined by

$$\tau_f ::= \mathbf{s} \mid f \tau_f^1 \dots \tau_f^n$$

where  $\mathbf{s}$  is either a cell  $\mathbf{c} \in \mathbb{C}$  or an input  $i \in \mathbb{I}$ , and  $f \in \Sigma_F$  is a function symbol. *Constants*  $\Sigma_F^0 \subseteq \Sigma_F$  are 0-ary function symbols. *Predicate terms*  $\tau_p \in \mathcal{T}_P$  are obtained by applying a predicate symbol  $p \in \Sigma_P$  with  $\Sigma_P \subseteq \Sigma_F$  to a tuple of function terms. PastTSL formulas are built according to the following grammar:

$$\varphi, \psi ::= \neg\varphi \mid \varphi \wedge \psi \mid \ominus\varphi \mid \varphi \mathcal{S} \psi \mid \tau_p \mid \llbracket \mathbf{c} \leftarrow \tau_f \rrbracket$$

An *update term*  $\llbracket \mathbf{c} \leftarrow \tau_f \rrbracket \in \mathcal{T}_U$  denotes that cell  $\mathbf{c}$  is overwritten with  $\tau_f$ . The temporal operators are called “Yesterday”  $\ominus$  and “Since”  $\mathcal{S}$ . Inputs, function symbols, and predicate symbols are purely syntactic objects. To assign meaning to them, let  $\mathcal{V}$  be the set of values with  $\mathbb{B} \subseteq \mathcal{V}$ . We denote by  $\mathcal{I} : \mathbb{I} \rightarrow \mathcal{V}$  the evaluation of inputs. An *assignment function*  $\langle \cdot \rangle : \Sigma_F \rightarrow \mathcal{F}$  assigns function symbols to functions  $\mathcal{F} = \bigcup_{n \in \mathbb{N}} \mathcal{V}^n \rightarrow \mathcal{V}$ .

The type  $\mathcal{C} = \mathbb{C} \rightarrow \mathcal{T}_F$  describes an update of all cells. For every cell  $\mathbf{c} \in \mathbb{C}$ , let  $init_{\mathbf{c}}$  be its initial value. The evaluation function  $\eta_{\langle \cdot \rangle} : \mathcal{C}^\omega \times \mathcal{I}^\omega \times \mathbb{N} \times \mathcal{T}_F \rightarrow \mathcal{V}$  evaluates a function term at point in time  $i$  with respect to an *input stream*  $\iota \in \mathcal{I}^\omega$  and a *computation*  $\varsigma \in \mathcal{C}^\omega$ :

$$\eta_{\langle \cdot \rangle}(\varsigma, \iota, i, \mathbf{s}) := \begin{cases} \iota \ i \ \mathbf{s} & \text{if } \mathbf{s} \in \mathbb{I} \\ init_{\mathbf{s}} & \text{if } \mathbf{s} \in \mathbb{C} \wedge i = 0 \\ \eta_{\langle \cdot \rangle}(\varsigma, \iota, i-1, \varsigma(i-1) \ \mathbf{s}) & \text{if } \mathbf{s} \in \mathbb{C} \wedge i > 0 \end{cases}$$

$$\eta_{\langle \cdot \rangle}(\varsigma, \iota, i, f \ \tau_0 \dots \tau_{m-1}) := \langle f \rangle \eta_{\langle \cdot \rangle}(\varsigma, \iota, i, \tau_0) \dots \eta_{\langle \cdot \rangle}(\varsigma, \iota, i, \tau_{m-1})$$

Note that  $\iota i \mathbf{s}$  denotes the value of  $\mathbf{s}$  at position  $i$  according to  $\iota$ . Likewise,  $\varsigma i \mathbf{s}$  is the function term that  $\varsigma$  assigns to  $\mathbf{s}$  at position  $i$ . With the exception of update and predicate terms, the semantics of pastTSL is similar to that of LTL.

$$\begin{array}{ll}
 \varsigma, \iota, t \models_{\langle \rangle} \neg\varphi & \text{iff } \varsigma, \iota, t \not\models_{\langle \rangle} \varphi \\
 \varsigma, \iota, t \models_{\langle \rangle} \varphi \wedge \psi & \text{iff } \varsigma, \iota, t \models_{\langle \rangle} \varphi \text{ and } \varsigma, \iota, t \models_{\langle \rangle} \psi \\
 \varsigma, \iota, t \models_{\langle \rangle} \ominus\varphi & \text{iff } t > 0 \wedge \varsigma, \iota, t-1 \models_{\langle \rangle} \varphi \\
 \varsigma, \iota, t \models_{\langle \rangle} \varphi \mathcal{S} \psi & \text{iff } \exists 0 \leq t' \leq t. \varsigma, \iota, t' \models_{\langle \rangle} \psi \text{ and} \\
 & \forall t' < k \leq t. \varsigma, \iota, k \models_{\langle \rangle} \varphi \\
 \varsigma, \iota, t \models_{\langle \rangle} \llbracket \mathbf{v} \leftarrow \tau \rrbracket & \text{iff } \varsigma t \mathbf{v} \equiv \tau \\
 \varsigma, \iota, t \models_{\langle \rangle} p \tau_0 \dots \tau_m & \text{iff } \eta_{\langle \rangle}(\varsigma, \iota, t, p \tau_0 \dots \tau_{m-1})
 \end{array}$$

We use  $\equiv$  to syntactically compare two terms. We derive three additional operators:  $\ominus\varphi := \neg\ominus\neg\varphi$ ,  $\diamond\varphi := \text{true } \mathcal{S} \varphi$ , and  $\boxminus\varphi := \neg\diamond\neg\varphi$ . The difference between  $\ominus$  and “Weak Yesterday”  $\odot$  is that  $\ominus$  evaluates to *false* in the first step and  $\odot$  to *true*. We use pastTSL formulas to describe safety properties. Therefore, we define that computation  $\varsigma$  and an input stream  $\iota$  satisfy a pastTSL formula  $\varphi$ , written  $\varsigma, \iota \models_{\langle \rangle} \varphi$ , if  $\forall i \in \mathbb{N}. \varsigma, \iota, i \models_{\langle \rangle} \varphi$ .

The realizability problem of a pastTSL formula  $\psi$  asks whether there exists a strategy that reacts to predicate evaluations with cell updates according to  $\psi$ . Formally, a strategy is a function  $\sigma : (2^{\mathcal{T}_P})^+ \rightarrow \mathcal{C}$ . For  $\iota \in \mathcal{I}^\omega$ , we write  $\sigma(\iota)$  for the computation obtained from  $\sigma$ :

$$\sigma(\iota)(i) = \sigma(\{\tau_p \in \mathcal{T}_P \mid \eta_{\langle \rangle}(\sigma(\iota), \iota, 0, \tau_p)\} \dots \{\tau_p \in \mathcal{T}_P \mid \eta_{\langle \rangle}(\sigma(\iota), \iota, i, \tau_p)\})$$

Note that in order to define  $\sigma(\iota)(i)$ , the definition uses  $\sigma(\iota)$ . This is well-defined since the evaluation function  $\eta_{\langle \rangle}(\varsigma, \iota, i, \tau)$  only uses  $\varsigma 0 \dots \varsigma(i-1)$ .

**Definition 1 ([13]).** *A pastTSL formula  $\psi$  is realizable if, and only if, there exists a strategy  $\sigma : (2^{\mathcal{T}_P})^+ \rightarrow \mathcal{C}$  such that for every input stream  $\iota \in \mathcal{I}^\omega$  and every assignment function  $\langle \rangle : \Sigma_F \rightarrow \mathcal{F}$  it holds that  $\sigma(\iota), \iota \models_{\langle \rangle} \psi$ .*

### 3 Parameterized TSL for Smart Contract Specifications

In this section, we introduce parameterized pastTSL and show how the past-time fragment of the logic can be used for specifying smart contract state machines.

#### 3.1 Parameterized TSL

Parameterized TSL extends TSL with universally quantified parameters. Let  $P$  be a set of parameters and  $\mathbb{C}_P$  a set of parameterized cells, where each cell is of the form  $\mathbf{c}(p_1, \dots, p_m)$  with  $p_1, \dots, p_m \in P$ . A parameterized TSL formula is a formula  $\forall p_1, \dots, \forall p_n. \psi$ , where  $\psi$  is a TSL formula with cells from  $\mathbb{C}_P$  and which may use parameters as base terms in function and predicate terms. We require

that the formula is closed, i.e., every parameter occurring in  $\psi$  is bound in the quantifier prefix.

Parameterized TSL formulas are evaluated with respect to a domain  $\mathbb{P}$  for the parameters. We use a function  $\mu : P \rightarrow \mathbb{P}$  to instantiate parameters. Given a parameterized TSL formula  $\forall p_1, \dots, \forall p_n. \psi$ ,  $\psi[\mu]$  is the formula obtained by replacing all parameters according to  $\mu$ . To simplify our constructions, we want  $\psi[\mu]$  to be a TSL formula. Therefore, we assume that  $\mathbb{P}$  is a subset of the set of constants and that  $c(\mu(p_1), \dots, \mu(p_m)) \in \mathbb{C}$ , i.e., the instantiation of a parameterized cell refers to a normal, non-parameterized cell. Given a computation  $\varsigma$  and an input stream  $\iota$ , we define  $\varsigma, \iota \models \forall p_1, \dots, \forall p_n. \psi$  iff  $\forall \mu : P \rightarrow \mathbb{P}. \varsigma, \iota \models \psi[\mu]$ .

### 3.2 Example: ERC20 Contract

We illustrate how parameterized pastTSL can be used to specify the state machine logic of smart contract with an ERC20 token system. An ERC20 token system provides a platform to transfer tokens between different accounts. We follow the Open Zeppelin documentation [37]. The special feature of the contract is the possibility to transfer not only tokens from one's own account but, after approval, also from a different account. The core contract consists of methods `transfer`, `transferFrom`, and `approve`. We do not model getters like `totalSupply` or `balanceOf` as they are not relevant for the temporal behavior of the contract. The Open Zeppelin ERC20 contract describes various extensions to the core contract, one of which is the ability to pause transfers. We distinguish between pausing transfers globally (`pause`) and from one's own account (`pause(m)`).

Our specifications describe the temporal control flow of the contract's method calls and the data flow of its fields. We distinguish between *requirements*, *obligations*, and *assumptions*. Requirements enforce the right order of method calls with correct arguments. Obligations describe the data flow in the fields of the contract. Assumptions restrict the space of possible predicate evaluations. For this example, we do not need any assumptions. A typical assumption in other specifications would be that  $x > y$  and  $y > x$  cannot hold at the same time.

To emphasize that all past-time formulas are required to hold globally, we add a  $\Box$  operator to formulas. We use two parameters `m` and `n`, where `m` always refers to the address from which tokens are subtracted and parameter `n`, whenever different from `m`, to the address that initiates the transfer. We start with the requirements. First, any transfer from `m` must be backed by sufficient funds.

$$\Box(\text{transfer}(m) \vee \text{transferFrom}(m,n) \rightarrow \text{suffFunds}(m, \text{arg@amount}))$$

Second, no method call can happen after `pause` until `unpause` is called:

$$\Box(\text{transferFrom}(m,n) \vee \text{transfer}(m) \vee \text{approve}(m,n) \vee \text{localPause}(m) \vee \text{localUnpause}(m) \rightarrow (\neg \text{pause} \mathcal{S} \text{unpause}) \vee \Box \neg \text{pause})$$

In contrast, `localPause(m)` only stops method calls from `m`'s account:

$$\Box(\text{transferFrom}(m,n) \vee \text{transfer}(m) \vee \text{approve}(m,n) \rightarrow ((\neg \text{localPause}(m)) \mathcal{S} \text{localUnpause}(m)) \vee \Box \neg \text{localPause}(m))$$

Finally, `pause` and `unpause` can only be called by the owner of the contract. Additionally, they cannot be called twice without the respective other in between and `unpause` cannot be called if `pause` has not been called at least once.

$$\begin{aligned} & \Box(\text{unpause} \rightarrow \text{msg.sender} = \text{owner}() \wedge \ominus(\neg \text{unpause} \mathcal{S} \text{pause})) \\ & \Box(\text{pause} \rightarrow \text{msg.sender} = \text{owner}() \wedge \ominus(\neg \text{pause} \mathcal{S} \text{unpause}) \vee \ominus \Box \neg \text{pause}) \end{aligned}$$

`msg.sender` is an input, whereas `owner()` is a constant. For the obligations, we need to make sure that the `approved` field is updated correctly. We use TSL's cell mechanism to model fields and use parameterized cells for mappings.

$$\begin{aligned} & \Box(\text{approve}(m, n) \rightarrow \llbracket \text{approved}(m, n) \leftarrow \text{arg@amount} \rrbracket) \\ & \Box(\text{transferFrom}(m, n) \rightarrow \llbracket \text{approved}(m, n) \leftarrow \text{approved}(m, n) - \text{arg@amount} \rrbracket) \\ & \Box(\neg(\text{transferFrom}(m) \vee \text{approve}(m, n)) \rightarrow \llbracket \text{approved}(m, n) \leftarrow \text{approved}(m, n) \rrbracket) \end{aligned}$$

Transitions that do not change the content of a cell are indicated by self-updates like  $\llbracket \text{approved}(m, n) \leftarrow \text{approved}(m, n) \rrbracket$ .

## 4 Synthesis Approach

The synthesis goal of this paper is to construct a state machine that satisfies parameterized pastTSL specifications like the one given in the last section.

### 4.1 Problem Statement

Our specifications are split into assumptions  $\varphi_A$ , requirements  $\varphi_R$ , and obligations  $\varphi_O$ , all of which are parameterized pastTSL formulas. Each of them can be given as invariant  $\varphi^{inv}$  or as initial formula  $\varphi^{init}$ . For synthesis, we compose them to the following formula, which, according to the definition of (parameterized) pastTSL, is required to hold globally.

$$\begin{aligned} \varphi := & \forall p_1, \dots, p_m. \\ & (\ominus \text{false} \rightarrow \varphi_A^{init} \wedge \varphi_R^{init}) \wedge (\Box(\varphi_A^{inv} \wedge \varphi_R^{inv})) \rightarrow (\ominus \text{false} \rightarrow \varphi_O^{init}) \wedge \varphi_O^{inv} \end{aligned}$$

Here,  $p_1, \dots, p_m$  are the parameters occurring in the inner formulas  $\varphi_A^{init}$ ,  $\varphi_R^{init}$ ,  $\varphi_O^{init}$ ,  $\varphi_A^{inv}$ ,  $\varphi_R^{inv}$ , and  $\varphi_O^{inv}$ . We use  $\ominus \text{false}$  to refer to the first position of a trace. It might seem counter-intuitive that we include requirements on the left side of the implication. The reason is that requirements describe a monitor on the method calls, which, from a synthesis perspective, constitute system inputs. Thus, if we conjoined requirements with obligations, the specification would be unrealizable. Instead, we leverage the fact that all specifications describe safety properties. Thus, state machines satisfying  $\varphi$  have a shape as depicted in Figure 1. Whenever an assumption or a requirement is violated, the machine enters an accepting sink state. To obtain

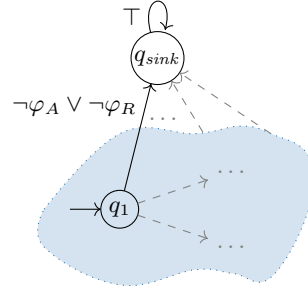


Fig. 1: Sketch of the system synthesized from  $\varphi$ . The dotted blue area implements the contract.

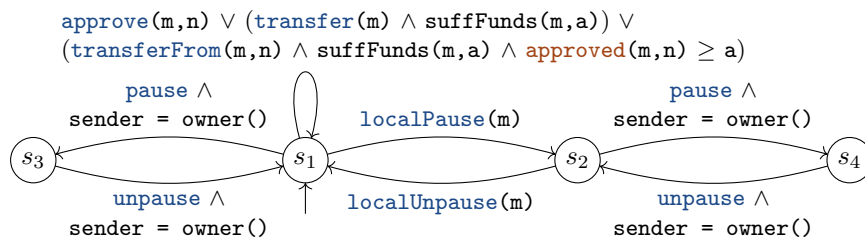


Fig. 2: System  $\mathcal{W}$  for the ERC20 contract. Irrelevant predicates and all cell updates are omitted for readability. We also write  $a$  instead of  $\text{arg@amount}$ .

the desired result, we reject any method call for which the system moves to the sink state. Like this, the remaining system enforces the requirements on method calls and also satisfies the obligations. For the rest of the paper, we depict state machines synthesized from  $\varphi$  without the sink state.

*On Safety Properties.* We restrict ourselves to safety properties for three reasons. First of all, as we consider the synthesis problem, our requirements can only describe a monitor on the method calls. Liveness properties are known not to be monitorable. For future work, one could consider model-checking the synthesized state machine with regard to liveness properties like “eventually, method  $X$  is *callable*”. Second, the restriction to safety automata enables the splitting algorithm described in Section 6, which is essential for our approach in order to efficiently implement the state machine in Solidity. Lastly, synthesis from safety properties is less complex than full LTL synthesis (c.f. Section 7), which enables us to synthesize non-trivial state machines within seconds.

## 4.2 High-Level Description of the Approach

*Challenges.* We need to address two major challenges. First, as parameters range over an infinite domain  $\mathbb{P}$ , parameterized pastTSL formulas describe (in general) *infinite-state systems*. Second, even if we managed to synthesize some representation of the infinite-state system, we still need to translate it to Solidity code. In Solidity, every computation costs gas. Therefore, we need to find a compact representation of the system that minimizes the number of computation steps needed to update the system after a method call.

*Approach in a Nutshell.* We address these challenges in two steps. First, we interpret the specification as being unquantified, i.e., we remove all quantifiers and treat the parameters as normal constants (e.g., in case of `suffFunds(m, arg@amount)`) or as part of the cells’ name (e.g., in case of `approved(m,n)`). Like that, we obtain a plain pastTSL formula that describes the finite-state system representing the correct control flow for every parameter instantiation.



We synthesize the winning region from that formula, which we call  $\mathcal{W}$ . For the running ERC20 example,  $\mathcal{W}$  can be found in Figure 2.

Of course, the contract can be in different states of  $\mathcal{W}$  depending on the parameter instantiation. In theory, we would therefore like to keep the necessary number of copies of  $\mathcal{W}$ . For example, if `approve(m=1,n=2)` is called, we would execute the corresponding transition in system  $\mathcal{W}_{(m=1,n=2)}$ . The problem with this naive approach is that calling a method parameterized with only a subset of the parameters would lead to updates of several systems. For example, if `localPause(m=1)` is called, this would have to be recorded in all  $\mathcal{W}_{(m=1,n=v)}$  for any value  $v$  of  $\mathbf{n}$  observed so far. Updating all these state machines after each method call would lead to a quick explosion of the gas consumption in Solidity. Instead, addressing the second challenge, we split  $\mathcal{W}$  into a hierarchical structure of state machines, one for each subset of parameters. As a result, we only have to update a single state machine per method call and still maintain the correct state of each instance (we describe this approach in more detail in Section 6.1). To summarize, we proceed as follows.

1. Interpret the parameterized pastTSL formula  $\varphi$  as a pastTSL formula  $\psi$  and synthesize the winning region  $\mathcal{W}$  from it.
2. Split  $\mathcal{W}$  into a hierarchical structure  $\mathcal{W}_1, \dots, \mathcal{W}_n$  and show how these systems can be interpreted as an infinite-state machine  $\mathcal{M}$  satisfying  $\varphi$ .
3. Generate Solidity code that implements transitions according to  $\mathcal{W}_1, \dots, \mathcal{W}_n$ .

In the following sections, we discuss each of these steps in detail.

## 5 PastTSL Synthesis

Let  $\varphi$  be a parameterized pastTSL formula as described in Section 4.1. We first translate  $\varphi$  to pastTSL. This is easy: just remove all quantifiers and interpret parameters as constants (i.e.,  $P \subseteq \Sigma_F^0$ ) and parameterized cells as normal cells (i.e.,  $\mathbb{C}_P \subseteq \mathbb{C}$ ).

Unfortunately, even though past-time fragments usually simplify logical problems, we establish that the realizability problem of pastTSL is undecidable. We obtain this result by a reduction from the universal halting problem of lossy counter machines [34].

An *n-counter machine* (*nCM*) consists of a finite set of instructions  $l_1, \dots, l_m$ , which modify  $n$  counters  $c_1, \dots, c_n$ . Each instruction  $l_i$  is of one of the following forms, where  $1 \leq x \leq n$  and  $1 \leq j, k \leq m$ .

- $l_i$ :  `$c_x := c_x + 1$ ; goto  $l_j$`
- $l_i$ : `if  $c_x = 0$  then goto  $l_j$  else  $c_x := c_x - 1$ ; goto  $l_k$`
- $l_i$ : `halt`

A configuration of a *nCM* is a tuple  $(l_i, v_1, \dots, v_n)$ , where  $l_i$  is the next instruction to be executed, and  $v_1, \dots, v_n$  denote the values of the counters. Compared to non-lossy *nCMs*, the counters of a lossy *nCM* may spontaneously decrease. We employ a version of lossiness where a counter can become zero if it is tested

for zero (see [34] for details). A lossy  $n$ CM halts from an initial configuration if it eventually reaches a state with the halting instruction.

**Theorem 1.** *The pastTSL realizability problem is undecidable.*

*Proof.* We reduce from the universal halting problem of lossy  $n$ CMs, which is undecidable [34]. We spell out the main ideas. Our formulas consist of one constant  $z()$ , one function  $f$ , and one predicate  $p$ . There are no inputs. Applying an idea from [28], we use two cells for every counter  $c_x$ :  $c_x^{inc}$  to count increments and  $c_x^{dec}$  to count decrements. Applying  $f$  to  $c_x^{inc}$  increments the counter, applying  $f$  to  $c_x^{dec}$  decrements it. If the number of increments and decrements is equal, the counter is zero. In TSL, we use the formula  $\psi_x^0 := p(c_x^{inc}) \leftrightarrow p(c_x^{dec})$  to test if a counter is zero. Note that if the counter really is zero, then the test for zero *must* evaluate to *true* by the TSL semantics. For all other cases, it *may* evaluate to *true*. If the equivalence evaluates to *true* even though the counter is non-zero, we interpret it as a spontaneous reset. Initially, the value of the counters need to be arbitrary. We reflect this by making no assumptions on the first step, thereby allowing the strategy to set the counter cells to any valid function term  $f^*(z())$ . We use  $n$  cells  $l_1, \dots, l_n$  for encoding the instructions. Globally, all instruction cells but the one indicating the next instruction, indicated by  $\llbracket l_i \leftarrow f(l_i) \rrbracket$ , need to self-update. We spell out the encoding of an instruction of the second type.

$$\begin{aligned} \Box(\Box \llbracket l_i \leftarrow f(l_i) \rrbracket \rightarrow (\psi_x^0 \rightarrow \llbracket l_j \leftarrow f(l_j) \rrbracket \wedge \llbracket c_x^{inc} \leftarrow z() \rrbracket \wedge \llbracket c_x^{dec} \leftarrow z() \rrbracket) \\ \wedge (\neg \psi_x^0 \rightarrow \llbracket l_k \leftarrow f(l_k) \rrbracket \wedge \llbracket c_x^{inc} \leftarrow c_x^{inc} \rrbracket \wedge \llbracket c_x^{dec} \leftarrow f(c_x^{dec}) \rrbracket)) \end{aligned}$$

The formula tests if the instruction to be executed is  $l_i$ . If so, we test the counter  $c_x$  for zero and set the corresponding cell to  $z()$  if that is the case. Furthermore, the correct next instruction is updated by applying  $f$ . Finally, we encode that we never reach a halting state:  $\Box \neg \llbracket l_{halt} \leftarrow f(l_{halt}) \rrbracket$ . The resulting pastTSL formula is realizable if, and only if, there is an initial state such that the machine never halts. Thus, undecidability of the pastTSL realizability problem follows.

## 5.1 PastTSL Synthesis via PastLTL Approximation

As pastTSL realizability is undecidable, we have to approximate the synthesis problem. To do so, we employ a reduction proposed in [13], which approximates TSL synthesis in LTL, for which realizability is decidable. The reduction replaces all predicate terms and update terms of a TSL formula  $\psi$  with unique atomic propositions, e.g.,  $a_{p,x}$  for  $p(\mathbf{x})$  and  $a_{x,to-f,x}$  for  $\llbracket \mathbf{x} \leftarrow f(\mathbf{x}) \rrbracket$ . Additionally, the reduction adds a formula that ensures that every cell is updated with exactly one function term in each step. Given a pastTSL formula  $\psi$ , the reduction produces an LTL approximation  $\psi_{LTL}$  that also falls into the past-time fragment. The reduction is sound but not complete [13], i.e.,  $\psi$  might be realizable even if  $\psi_{LTL}$  is not. For the smart contract specifications we produced for our evaluation, however, we never encountered spurious unrealizability.

Let  $AP$  be the set of atomic propositions of  $\psi_{LTL}$ . From every trace  $t$  over  $AP$ , we can directly generate a computation  $comp(t) \in \mathcal{C}^\omega$  as follows:

$$comp(t)(i)(c) = \tau_f \quad \text{if } a_{c\_to\_tau_f} \in t(i)$$

For the other direction, given a computation  $\varsigma$ , an input stream  $\iota$ , and an assignment function  $\langle \cdot \rangle$ , we write  $LTL(\iota, \varsigma, \langle \cdot \rangle)$  for the corresponding trace over  $AP$ .

$$LTL(\iota, \varsigma, \langle \cdot \rangle)(i) = \{\{a_{\tau_p} \mid \eta_{\langle \cdot \rangle}(\varsigma, \iota, i, \tau_p)\}\} \cup \{a_{c\_to\_tau_f} \mid \varsigma(i)(c) = \tau_f\}$$

The following proposition follows from the soundness of the approximation.

**Proposition 1.** *For every assignment function  $\langle \cdot \rangle$ , input stream  $\iota$ , and computation  $\varsigma$ ,  $LTL(\iota, \varsigma, \langle \cdot \rangle) \models \psi_{LTL}$  iff  $\varsigma, \iota \models_{\langle \cdot \rangle} \psi$ .*

*Parameterized Atomic Propositions.* In our case, the pastTSL formula  $\psi$  is obtained from a parameterized pastTSL formula  $\varphi$ . Thus, the atomic propositions of  $\psi_{LTL}$  contain parameters, e.g.,  $a_{transferFrom\_m\_n}$ . To enable correctness reasoning in the next section, we lift the instantiation of parameters to the level of atomic propositions and LTL formulas.

For  $a \in AP$ , we write  $a(p_1, \dots, p_m)$  if  $a$  contains parameters  $p_1, \dots, p_m$ . We usually denote the sequence  $p_1, \dots, p_m$  with some  $P_i$ , for which we also use set notation. We assume that every proposition occurs with only one sequence of parameters, i.e., there are no  $a(P_i), a(P_j) \in AP$  with  $P_i \neq P_j$ .

Given  $\mu : P \rightarrow \mathbb{P}$ ,  $P_i[\mu]$  denotes  $(\mu(p_1), \dots, \mu(p_m))$  and  $a[\mu]$  denotes  $a(P_i[\mu])$ . For example, for  $a_{transferFrom\_m\_n}[m \mapsto 1, n \mapsto 2]$ , we obtain  $a_{transferFrom\_1\_2}$ . We also write  $\psi_{LTL}[\mu]$  for an LTL formula where every atomic proposition is instantiated according to  $\mu$ . We define  $AP_{\mathbb{P}} = \{a[\mu] \mid a \in AP, \mu : P \rightarrow \mathbb{P}\}$ . As there are no two  $a(P_i), a(P_j) \in AP$  with  $P_i \neq P_j$ , for any  $\alpha \in AP_{\mathbb{P}}$ , there is exactly one  $a$  such that  $a[\mu] = \alpha$  for some  $\mu$ .

## 6 Splitting Algorithm

In the last section, we discussed that we need to approximate the parameterized pastTSL formula  $\varphi$  to an LTL formula  $\psi_{LTL}$  to synthesize  $\mathcal{W}$ . Note that  $\mathcal{W}$  alone does not implement a strategy for  $\varphi$  as each parameter instance might be in a different state of  $\mathcal{W}$  (c.f. Section 4.2). In this section, we discuss how to split up  $\mathcal{W}$  to enable an efficient implementation in Solidity while at the same time making sure that the generated traces realize the original formula  $\varphi$ .

### 6.1 Idea of the Algorithm

The idea of the algorithm is to split  $\mathcal{W}$  into multiple subsystems  $\mathcal{W}_1, \dots, \mathcal{W}_n$  such that each  $\mathcal{W}_i$  contains the transitions for method calls with parameters  $P_i$ . For the ERC20 example, we produce the three systems  $\mathcal{W}_\emptyset$ ,  $\mathcal{W}_{\{m\}}$ , and  $\mathcal{W}_{\{m, n\}}$  depicted in Figure 3. For each of these systems, at runtime, we create a copy for every instantiation of their parameters.

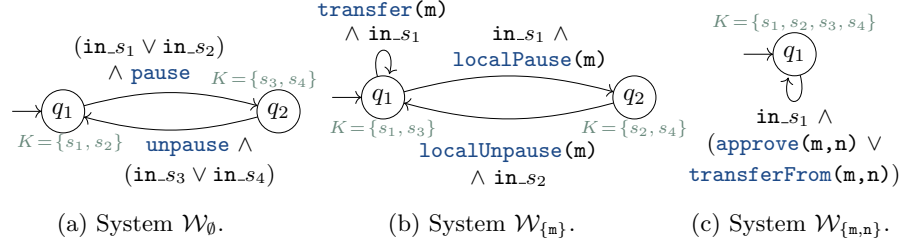


Fig. 3: State machines for all non-empty parameter sets. For readability, we omit cell updates and all predicates apart from method calls.

If a method with parameters  $P_i$  is called and  $\mathcal{W}_i$  is in state  $q$ , then the transition from  $q$  labeled with that method call is the candidate transition to be executed. This means that compared to the naive solution (c.f. Section 4.2) a call to `localPause(m=1)` only has to be recorded in a single transition system (namely  $\mathcal{W}_{\{m=1\}}$ ).

Crucially, however, we now need to ensure that we only produce traces of  $\mathcal{W}$ . For example, if `localPause(m=1)` is called, we move from state  $q_1$  to  $q_2$  in system  $\mathcal{W}_{\{m=1\}}$ . This corresponds to a transition from  $s_1$  to  $s_2$  in  $\mathcal{W}$ . Now, for instances with  $m=1$ , calls to all methods except `localUnPause(m=1)` and `pause` need to be rejected (according to  $\mathcal{W}$ ), even though these would technically be possible in systems  $\mathcal{W}_{\{m=1, n=v\}}$  (for any  $v$ ). To do so, we synchronize the systems with the help of transition guards `in_s_i` and additional state labels  $K$ .

A guard `in_s_i` indicates that the transition can only be taken in state  $s_i$  of  $\mathcal{W}$ . To check if this requirement is satisfied, the systems share their knowledge about the state  $\mathcal{W}$  would currently be in. A knowledge label  $K = \{s_1, s_2\}$  in state  $q$  of  $\mathcal{W}_i$  means that  $\mathcal{W}$  could be in state  $s_1$  or state  $s_2$  if  $\mathcal{W}_i$  is in state  $q$ . Each system is a projection to some transitions of  $\mathcal{W}$  and therefore has different knowledge labels.

The systems share their knowledge in order to determine which state  $\mathcal{W}$  would be in for a trace of one parameter instantiation. For each method call, the systems must come to a conclusion if that call would be allowed in the current state of  $\mathcal{W}$ . However,  $\mathcal{W}_i$  may only use the knowledge of systems  $\mathcal{W}_j$  with  $P_j \subseteq P_i$  as these are the parameters for which there is currently a value available. To guarantee that an unambiguous conclusion is always possible to achieve, we formulate two simple requirements and an independence check.

## 6.2 Construction

Let  $\psi_{\text{LTL}}$  be given. The formula is the approximation of a pastTSL formula and therefore ranges over  $AP = I \cup O$ , where  $I$  are the atomic propositions obtained from predicate terms and  $O$  are the ones obtained from update terms. For  $A \subseteq AP$ , we write  $A|_O$  instead of  $A \cap O$ . We denote the set of atomic

propositions that correspond to some method call  $\mathbf{f}(P_i)$  by  $I_{call} \subseteq I$  and the set of output propositions that denote self-updates by  $O_{self} \subseteq O$ .

Let  $\mathcal{W} = (S_{\mathcal{W}}, s_{\mathcal{W}}^0, \delta_{\mathcal{W}})$  be the finite-state machine over  $AP$  that constitutes the winning region of  $\psi_{LTL}$ .  $\delta_{\mathcal{W}}$  its transition relation. We state two requirements on  $\mathcal{W}$ , which are needed to enable a sound splitting of  $\mathcal{W}$  and can be checked easily by inspecting all its transitions. First, we require that calls to a method parameterized with parameter sequence  $P_i$  only result in cell updates parameterized with the same parameter sequence.

**Requirement 1** (Local Updates). *For every transition  $(s, A, s') \in \delta_{\mathcal{W}}$ , if  $\circ(P_i) \in A_{|O}$  and  $\circ(P_i) \notin O_{self}$ , then there is a method call proposition  $\mathbf{f}(P_i) \in A$ .*

Second, whether a method can be called at a given state must not depend on predicates with parameters that are not included in the current method call.

**Requirement 2** (Independence of Irrelevant Predicates). *For every  $(s, A, s') \in \delta_{\mathcal{W}}$ , if  $\mathbf{f}(P_i) \in A$ , then for every  $a(P_j) \in I$  with  $P_j \not\subseteq P_i$  and  $a(P_j) \notin I_{call}$ , there is a transition  $(s, A', s')$  with  $a(P_j) \in A$  iff  $a(P_j) \notin A'$  and  $A_{|O} = A'_{|O}$ .*

The above requirement is needed to unify software and state machine reasoning. In state machines, the value of all propositions needs to be known to determine the right transition. In software, however, if `localPause(m)` is called, the value of `n` is undefined and we cannot evaluate predicates depending on `n`.

If  $\mathcal{W}$  satisfies the above requirements, we construct  $\mathcal{W}_1, \dots, \mathcal{W}_n$  for each parameter subset  $P_i$ . Each  $\mathcal{W}_i$  projects  $\mathcal{W}$  to the method calls with parameters  $P_i$ . The algorithm to construct the projections combines several standard automata-theoretic concepts:

1. Introduce a new guard proposition `in_s` for every state  $s \in S_{\mathcal{W}}$  of  $\mathcal{W}$ . For every transition  $(s, A, s') \in \delta_{\mathcal{W}}$ , replace  $A$  with  $A \cup \{\mathbf{in}_s\}$ .
2. Label all transitions  $(s, A, s') \in \delta_{\mathcal{W}}$  for which there is no  $\mathbf{f}(P_i) \in A$  with  $\epsilon$ . The result is a nondeterministic safety automaton with  $\epsilon$ -edges.
3.  $\mathcal{W}_i$  is obtained by determinizing the safety automaton using the standard subset construction. This removes all  $\epsilon$  transitions. During the construction, we label each state with the subset of  $S_{\mathcal{W}}$  it represents, these are the knowledge labels  $K$ .

We use  $S_i$  for the states of  $\mathcal{W}_i$ ,  $\delta_i$  for its transition relation, and  $K_i : S_i \rightarrow 2^{S_{\mathcal{W}}}$  for the knowledge labels. Note that every transition in  $\mathcal{W}$  is labeled with exactly one method call proposition and is therefore present in exactly one  $\mathcal{W}_i$ . The following two propositions follow from the correctness of the subset construction for the determinization of finite automata. The first proposition states that the outgoing transitions of a state  $s_i \in \mathcal{W}_i$  are exactly the outgoing transitions of all states  $s \in K_i(s_i)$ .

**Proposition 2.** *For every state  $s_i \in S_i$ , if  $s \in K_i(s_i)$ , then for all  $s' \in S$  and  $A \subseteq AP$ ,  $(s, A, s') \in \delta_{\mathcal{W}}$  iff  $(s_i, A \cup \{\mathbf{in}_s\}, s'_i) \in \delta_i$  for some  $s'_i \in S_i$ .*

The second one states that the knowledge labels in  $\mathcal{W}_i$  are consistent with the transitions of  $\mathcal{W}$ .

**Proposition 3.** *Let  $(s, A, s') \in \delta_{\mathcal{W}}$  with  $f(P_i) \in A$ . Then, for every state  $s_i \in S_i$  with  $s \in K_i(s_i)$ , and every transition  $(s_i, A \cup \{\text{in}_s\}, s'_i) \in \delta_i$ , it holds that  $s' \in K_i(s'_i)$ . Furthermore, for every  $s_j$  of  $\mathcal{W}_j$  with  $i \neq j$ , if  $s \in K_j(s_j)$ , then  $s' \in K_j(s_j)$ .*

### 6.3 Check for Independence

We now define the check if transitions in  $\mathcal{W}_i$  can be taken independently of the current state of all  $\mathcal{W}_j$  with  $P_j \not\subseteq P_i$ . If the check is positive, we can implement the system efficiently in Solidity: when a method  $f(P_i)$  is called, we only need to update the single system  $\mathcal{W}_i$  and whether the transition can be taken only depends on the available parameters.

Let  $s_i$  and  $s'_i$  be states in  $\mathcal{W}_i$  and  $A \subseteq AP$ . Let  $G_{(s_i, A, s'_i)} = \{s \mid (s_i, A \cup \{\text{in}_s\}, s'_i) \in \delta_i\}$  be the set of all guard propositions that occur on transitions from  $s_i$  to  $s'_i$  with  $A$ . Let  $P_{j_1}, \dots, P_{j_l}$  be the maximum set of parameter subsets such that  $P_{j_k} \subseteq P_i$  for  $1 \leq k \leq l$ . A transition  $(s_i, A, s'_i)$  is *independent* if for all states  $s_{j_1}, \dots, s_{j_l}$  with  $s_{j_k} \in S_{j_k}$  either

- (i)  $K_i(s_i) \cap \bigcap_{1 \leq k \leq l} K_{j_k}(s_{j_k}) \subseteq G_{(s_i, A, s'_i)}$  or
- (ii)  $(K_i(s_i) \cap \bigcap_{1 \leq k \leq l} K_{j_k}(s_{j_k})) \cap G_{(s_i, A, s'_i)} = \emptyset$ .

The check combines the knowledge of  $\mathcal{W}_i$  in state  $s_i$  with the knowledge of each combination of states from  $\mathcal{W}_{j_1}, \dots, \mathcal{W}_{j_l}$ . For each potential combination, it must be possible to determine whether transition  $(s_i, A, s'_i)$  can be taken. If the first condition is satisfied, then the combined knowledge leads to the definite conclusion that  $\mathcal{W}$  is currently in a state where an  $A$ -transition can be taken. If the second condition is satisfied, it definitely cannot be taken. If none of the two is satisfied, then the combined knowledge of  $P_i$  and all  $P_{j_k}$  is insufficient to reach a definite answer.

Note that some state combinations  $s_i, s_{j_1}, \dots, s_{j_l}$  might be impossible to reach. But then, we have that  $K_i(s_i) \cap \bigcap_{1 \leq k \leq l} K_{j_k}(s_{j_k}) = \emptyset$  and the second condition is satisfied. The check is successful if all transitions  $(s_i, A, s'_i)$  in all  $\delta_i$  are independent.

### 6.4 Interpretation as Infinite-State Machine

The goal of this section is to construct a state machine  $\mathcal{M}$  from  $\mathcal{W}_1, \dots, \mathcal{W}_n$  such that the original parameterized pastTSL formula  $\varphi$  is satisfied. To simplify the presentation, we define  $\mathcal{M}$  as a state machine over  $AP_{\mathbb{P}}$ . Due to the direct correspondence of atomic propositions in  $AP_{\mathbb{P}}$  to predicate and update terms  $\mathcal{T}_P \cup \mathcal{T}_U$ , a state machine for  $\varphi$  can easily be obtained from that. In the following, we assume that  $\mathcal{W}$  satisfies Requirements 1 and 2 and that  $\mathcal{W}_1, \dots, \mathcal{W}_n$  pass the check for independence. We construct  $\mathcal{M}$  as follows.

A state in  $\mathcal{M}$  is a collection of  $n = |2^P|$  functions  $f_1, \dots, f_n$ , where  $f_i : \mathbb{P}^m \rightarrow S_i$  if  $P_i = (p_{i_1}, \dots, p_{i_m})$ . Each  $f_i$  indicates in which state of  $\mathcal{W}_i$  instance  $\mu$  currently is. The initial state is the collection of functions that all map to

the initial states of their respective  $\mathcal{W}_i$ . For every state  $s = (f_1, \dots, f_n)$  of  $\mathcal{M}$ , every  $P_i \subseteq P$ , and every instance  $\mu$ , we add a transition where  $P_i[\mu]$  takes a step and all other instances stay idle. Let  $f_i(P_i[\mu]) = s_i$ ,  $s'_i \in S_i$ ,  $A \subseteq AP$ , and  $G_{(s_i, A, s'_i)} = \{s \mid (s_i, A \cup \{\text{in}_s\}, s'_i) \in \delta_i\}$ . Let  $P_{j_1}, \dots, P_{j_l}$  be all subsets of  $P_i$ . If  $K_i(s_i) \cap \bigcap_{1 \leq k \leq l} K_{j_k}(f_{j_k}(P_{j_k}[\mu])) \subseteq G_{(s_i, A, s'_i)}$ , we add the transition  $(s, A', s')$  to  $\mathcal{M}$ , where  $A'$  and  $s'$  are defined as follows.

$$\begin{aligned} A' &= \{a[\mu] \mid a \in A\} \cup \{o[\mu'] \mid o \in O_{self}, o[\mu'] \neq o[\mu]\} \\ s' &= (f_1, \dots, f_i[P_i[\mu] \mapsto s'_i], \dots, f_n) \end{aligned}$$

The label  $A'$  sets all propositions of instance  $\mu$  as in  $A$  and sets all other input propositions to *false*. Of all other outputs propositions, it only sets those denoting self-updates to *true*.

## 6.5 Correctness

Finally, we argue that  $\mathcal{M}$  as defined above satisfies the original specification  $\varphi$  for all instantiations of its parameters.

*Trace Projection.* To obtain a compact state machine, our specifications require that in each step, exactly one method is called. Like that, the resulting specification describes the control flow projected on each instance. To argue that  $\mathcal{M}$  satisfies  $\varphi$ , we therefore need to project its traces to the steps relevant for an instance  $\mu$ . These are the steps that either include a method call to  $\mu$  or a non-self-update of one of  $\mu$ 's cells.

For  $A \subseteq AP_{\mathbb{P}}$ , we define  $A_{\mu}$  as  $\{\alpha \in A \mid \exists a \in AP. \alpha = a[\mu]\}$ . Let  $traces(\mathcal{M})$  be the set of infinite traces produced by  $\mathcal{M}$ . Given  $t \in traces(\mathcal{M})$ , let  $t' = (t[0])_{\mu}(t[1])_{\mu} \dots$ . Now, we define  $t_{\mu}$  to be the trace obtained from  $t'$  by deleting all positions  $i$  such that  $(t[i]_{\mu})|_O \subseteq O_{self}$  and  $\neg \exists \mathbf{f}(P_i) \in I_{call}. \mathbf{f}(P_i)[\mu] \in t'[i]$ . Note that  $t_{\mu}$  might be a finite trace even if  $t$  is infinite. Since  $t_{\mu}$  only deletes steps from  $t$  that do not change the value of the cells,  $t_{\mu}$  still constitutes a sound computation regarding the TSL semantics. We define  $traces_{\mu}(\mathcal{M}) = \{t_{\mu} \mid t \in traces(\mathcal{M})\}$ .

*Correctness Proof.* Most of the work is done in the following lemma. We define  $\mathcal{W}_{\mu}$  as the state machine that replaces the transition labels of  $\mathcal{W}$  with their instantiations according to  $\mu$ , i.e., if  $(s, A, s') \in \mathcal{W}$ , then  $(s, A[\mu], s') \in \mathcal{W}_{\mu}$ . Not every infinite run of  $\mathcal{M}$  corresponds to an infinite run in  $\mathcal{W}_{\mu}$  for every  $\mu$ . However, we show that if the run has infinitely many  $\mu$ -transitions, then it can be mapped to an infinite trace in  $\mathcal{W}_{\mu}$ . The proof of the lemma can be found in the full version of this paper [11].

**Lemma 1.** *For every instance  $\mu$ ,  $traces_{\mu}(\mathcal{M}) = traces(\mathcal{W}_{\mu})$ .*

From the above lemma we directly obtain the desired correctness result.

**Theorem 2.** *Let  $\varphi = \forall \mathbf{p}_1, \dots, \mathbf{p}_m. \psi$  be a parameterized pastTSL formula and  $\psi_{LTL}$  its LTL approximation. If  $\mathcal{W}$  is the winning region of  $\psi_{LTL}$ ,  $\mathcal{W}$  satisfies Requirements 1 and 2, and can be split into  $\mathcal{W}_1, \dots, \mathcal{W}_n$  such that the check for independence is successful, then for every  $\mu$ ,  $\mathcal{M}$  defines a strategy for  $\varphi[\mu]$ .*

*Proof.* Let  $\mu : P \rightarrow \mathbb{P}$  be an instantiation of the parameters  $p_1, \dots, p_m$ ,  $\iota \in \mathcal{I}^\omega$  be an input stream, and  $\langle \cdot \rangle$  be an assignment function. First, for any trace  $t \in \text{traces}_\mu(\mathcal{M})$  with  $LTL(\iota, \text{comp}(t), \langle \cdot \rangle) = t$  (see Section 5.1 for the definition), we have that  $t \in \text{traces}(\mathcal{W}_\mu)$  because of Lemma 1. As all traces of  $\mathcal{W}$  satisfy  $\psi_{LTL}$ ,  $t \models \psi_{LTL}[\mu]$  (since  $\mu$  is only a renaming of atomic propositions on the LTL-level). By Proposition 1, we obtain  $\iota, \text{comp}(t) \models \psi[\mu]$ . Second, as  $\mathcal{W}$  implements the set of all strategies satisfying  $\psi_{LTL}$ , with the same reasoning, there is at least one  $t$  in  $\mathcal{M}$  with  $LTL(\iota, \text{comp}(t), \langle \cdot \rangle) = t$ .

## 6.6 Extension to Existential Quantifiers

Currently, our approach cannot handle existential quantifiers. In the example of the ERC20 contract, this forbids us to use a field `funds(m)` to store the balance of all users of the contract. If we were to try, we could use an additional parameter `r` for the recipient of the tokens and state the following.

$$\begin{aligned} & \forall m, n, r. \Box(\text{transferFrom}(m, n, r) \vee \text{transfer}(m, r) \\ & \quad \rightarrow \llbracket \text{funds}(m) \leftarrow \text{funds}(m) - \text{arg@amount} \rrbracket \\ & \quad \wedge \llbracket \text{funds}(r) \leftarrow \text{funds}(r) + \text{arg@amount} \rrbracket) \end{aligned}$$

However, for completeness, we would have to specify that the `funds` field does not spuriously increase, which would require existential quantifiers.

$$\begin{aligned} & \forall r. \Box(\llbracket \text{funds}(r) \leftarrow \text{funds}(r) + \text{arg@amount} \rrbracket \\ & \quad \rightarrow \exists m. \exists n. \text{transferFrom}(m, n, r) \vee \text{transfer}(m, r)) \end{aligned}$$

A similar limitation stems from Requirement 1, which requires that a field parameterized with set  $P_i$  can only be updated by a method that is also parameterized with  $P_i$ . As for existential quantifiers, we would otherwise not be able to distinguish spurious updates from intended updates of cells. While it might be challenging to extend the approach with arbitrary existential quantification, it should be possible for future work to include existential quantification that prevents spurious updates. One could, for example, define some sort of “lazy synthesis”, which only does a non-self-update when necessary.

## 7 Implementation and Evaluation

### 7.1 Implementation

We implemented our approach in a toolchain consisting of several steps. First, we translate the pastTSL specification into a pastLTL formula using TSLtools [24], which we adapted to handle past-time operators. We then synthesize a state machine using BDD-based symbolic synthesis. To make our lives easier, we implemented a simple analysis to detect free choices and deadlocks, which both indicate potential specification errors. If the specification contains parameters,



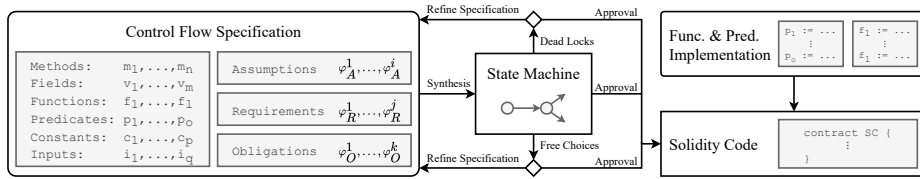


Fig. 4: Workflow of our smart contract control flow synthesis.

we split the resulting state machine as described in Section 6. Lastly, the state machines are translated to Solidity code. The toolchain is implemented in our tool SCSYNT consisting of approximately 3000 lines of Python code (excluding TSLtools). From a user perspective, we obtain the workflow depicted in Figure 4.

*Synthesis from PastLTL.* The first part of our toolchain implements a symbolic synthesis algorithm for pastLTL. As such, it can also be employed outside the context of smart contract synthesis. We are not aware of any other tool that implements pastLTL synthesis. We first build the safety automaton of the specification using a representation as BDDs. For pastLTL, a symbolic approach is especially efficient due to the long-known fact that for evaluating a pastLTL formula at time point  $i$ , it is sufficient to know the value of all subformulas at point  $i - 1$  [18]. Afterwards, we symbolically extract the winning region from the safety region with a classic fixpoint attractor construction. Finally, we minimize the resulting state machine using an explicit implementation of Hopcroft’s minimization algorithm [19].

*State Machine Analysis.* We analyze the winning region for free choices and potential deadlocks, which both usually indicate specification errors. A free choice is a state which, for the same input, has multiple outgoing transitions into the winning region. If there are free choices and the developer has no preference which one is chosen, SCSYNT nondeterministically commits to one option. For the deadlock detection, we require the user to label *determined predicate terms*. We call a predicate determined if either 1) it becomes a constant at some time or 2) only method calls can change its value. An example of class 1 are predicates over the time, e.g., `time > cTime()`: if it is *true* at some point, it will never be *false* again. A class 2 example would be a predicate that counts whether the number stored in a field has passed a fixed threshold. A predicate like `msg.sender = owner()`, on the other hand, is not determined as the evaluation changes with the input `msg.sender`. SCSYNT automatically detects if, at some point, there is an evaluation of the determined predicate terms that is allowed by the assumptions but for which there is no valid transition. It then warns of a potential deadlock.

*Translation to Solidity.* For the translation, the developer needs to provide the implementation of all predicates and functions, as they are uninterpreted (which makes the synthesis feasible after all). Some of the most common functions and

Table 1: Sizes of the specifications and state machines as well as the average running time of SCSYNT. #Forms. is the number of individual past-time formulas, #Nodes is the number of nodes of the AST. The state machine size is the sum of the states/transitions of the split state machines. The synthesis and translation times are the respective average on 10 runs of the same benchmark.

Contract	Specification		State Machine		Avg. Time (s)	
	#Forms.	#Nodes	#States	#Trans.	Synth.	Transl.
Asset Transfer	36	216	8	14	5.9996	0.0053
Blinded Auction	19	218	5	8	1.5446	0.0026
Coin Toss	27	154	5	7	1.6180	0.0029
Crowd Funding	17	100	4	8	0.2178	0.0026
ERC20	15	140	9	5	0.4812	0.0033
ERC20 Extended	19	244	10	7	1.9608	0.0040
NFT Auction	30	325	12	15	12.1853	0.0080
Simple Auction	15	83	4	7	0.1362	0.0026
Ticket System	13	97	4	6	0.1812	0.0028
Voting	17	98	6	5	0.1478	0.0023

predicates (e.g., equality and addition) are automatically replaced by SCSYNT. The `owner` and `msg.sender` keywords are translated automatically; the owner is set in the constructor. Conceptually, the translation to Solidity is straightforward. For each method of the contract, we create a function that contains the state machine logic for that particular method. For parameterized specifications, the contract is augmented with a mapping recording the knowledge labels (c.f. Section 6). The parameters other than the sender are included as arguments. Following [32], we also add automatic protection against reentrancy attacks by setting a Boolean flag if a method is currently executing.

## 7.2 Evaluation

The goal of our evaluation is to show that 1) parameterized pastTSL is indeed a suitable logic for specifying smart contract state machines and 2) that our implemented toolchain is efficient. To do so, we specified and synthesized ten different smart contracts with a non-trivial temporal control flow using pastTSL specifications with and without parameters. A detailed description of all benchmarks is provided in the full version of this paper [11]. The most challenging benchmark to specify was the NFT auction, a parameterized specification for a contract actively maintained by Avolabs. Its reference implementation has over 1400 lines of code. We manually extracted 30 past-time formulas from the README of the contract provided on the GitHub of Avolabs [2].

All experiments were run on a 2020 Macbook with an Apple M1 chip, 16GB RAM, running MacOS. The results are shown in Table 1. We report the size of the specification and of the resulting state machine as well as the running time of the synthesis procedure itself and the translation to Solidity code. Most importantly, the evaluation shows that specifying and automatically generating the non-trivial state machine logic of a smart contract is possible. We successfully

synthesized Solidity code for state machines of up to 12 states. The evaluation also shows that our toolchain is efficient: synthesis itself took up to 12 seconds; in most cases, SCSYNT synthesizes a state machine in less than two seconds. The translation of the state machine into Solidity code is nearly instantaneous.

## 8 Conclusion

We have described the synthesis of Solidity code from specifications given in pastTSL equipped with universally quantified parameters. Our approach is the first that facilitates a comprehensive specification of the implicit state machine of a smart contract, including the data flow of the contract’s fields and guards on the methods’ arguments. The algorithm proceeds in two steps: first, we translate the specification to pastTSL. While we have shown that pastTSL realizability without parameters is undecidable in general, solutions can be obtained via a sound reduction to LTL. In a second step, we split the resulting system into a hierarchical structure of multiple systems, which constitutes a finite representation of a system implementing the original formula and also enables a feasible handling when translated to Solidity. Our prototype tool SCSYNT implements the synthesis toolchain, including an analysis of the state machine regarding potential specification errors.

For future work, we aim to extend our approach to specifications given in pastTSL with alternating parameter quantifiers. There are also several exciting possibilities to combine our work with other synthesis and verification techniques. One avenue is to automatically prove necessary assumptions in deductive verification tools [8], especially for assumptions that state invariants maintained by method calls. Another possibility is to synthesize function and predicate implementations in the spirit of [5]. Finally, now that we have developed the algorithmic foundations and implemented a first prototype, we aim to conduct a thorough evaluation of our approach in comparison to hand-written (non-formal) approaches.

## References

1. Antonino, P., Ferreira, J., Sampaio, A., Roscoe, A.W.: Specification is law: Safe creation and upgrade of ethereum smart contracts. CoRR **abs/2205.07529** (2022). <https://doi.org/10.48550/arXiv.2205.07529>, <https://doi.org/10.48550/arXiv.2205.07529>
2. Avolabs: NFT auction reference contract. <https://github.com/avolabs-io/nft-auction> (2022), accessed: 2022-07-05
3. Azure: Asset transfer sample from the azure blockchain workbench. <https://github.com/Azure-Samples/blockchain/tree/master/blockchain-workbench/application-and-smart-contract-samples/asset-transfer> (2021), accessed: 2022-07-05
4. Azure: Asset transfer reference contract. <https://github.com/Azure-Samples/blockchain/blob/master/blockchain-workbench/application-and-smart-contract-samples/asset-transfer/ethereum/AssetTransfer.sol> (2022), accessed: 2022-07-05

5. Choi, W., Finkbeiner, B., Piskac, R., Santolucito, M.: Can reactive synthesis and syntax-guided synthesis be friends? In: 43rd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI) (June 2022), <https://publications.cispa.saarland/3674/>
6. Ciccio, C.D., Cecconi, A., Dumas, M., García-Bañuelos, L., López-Pintado, O., Lu, Q., Mendling, J., Ponomarev, A., Tran, A.B., Weber, I.: Blockchain support for collaborative business processes. *Inform. Spektrum* **42**(3), 182–190 (2019). <https://doi.org/10.1007/s00287-019-01178-x>, <https://doi.org/10.1007/s00287-019-01178-x>
7. Cimatti, A., Roveri, M., Sheridan, D.: Bounded verification of past LTL. In: Hu, A.J., Martin, A.K. (eds.) *Formal Methods in Computer-Aided Design*, 5th International Conference, FMCAD 2004, Austin, Texas, USA, November 15–17, 2004, Proceedings. *Lecture Notes in Computer Science*, vol. 3312, pp. 245–259. Springer (2004). [https://doi.org/10.1007/978-3-540-30494-4\\_18](https://doi.org/10.1007/978-3-540-30494-4_18), [https://doi.org/10.1007/978-3-540-30494-4\\_18](https://doi.org/10.1007/978-3-540-30494-4_18)
8. Dharanikota, S., Mukherjee, S., Bhardwaj, C., Rastogi, A., Lal, A.: Celestial: A smart contracts verification framework. In: *Proceedings of the 21st Conference on Formal Methods in Computer-Aided Design (FMCAD 2021)*. pp. 133–142 (2021)
9. Ethereum: Introduction to ethereum. <https://ethereum.org/en/developers/docs/intro-to-ethereum/> (2021), accessed: 2022-07-05
10. Finkbeiner, B., Heim, P., Passing, N.: Temporal stream logic modulo theories. In: Bouyer, P., Schröder, L. (eds.) *Foundations of Software Science and Computation Structures - 25th International Conference, FOSSACS 2022, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2022, Munich, Germany, April 2-7, 2022, Proceedings*. *Lecture Notes in Computer Science*, vol. 13242, pp. 325–346. Springer (2022). [https://doi.org/10.1007/978-3-030-99253-8\\_17](https://doi.org/10.1007/978-3-030-99253-8_17), [https://doi.org/10.1007/978-3-030-99253-8\\_17](https://doi.org/10.1007/978-3-030-99253-8_17)
11. Finkbeiner, B., Hofmann, J., Kohn, F., Passing, N.: *Reactive synthesis of smart contract control flows* (2023)
12. Finkbeiner, B., Klein, F., Piskac, R., Santolucito, M.: Synthesizing functional reactive programs. In: Eisenberg, R.A. (ed.) *Proceedings of the 12th ACM SIGPLAN International Symposium on Haskell, Haskell@ICFP 2019, Berlin, Germany, August 18-23, 2019*. pp. 162–175. ACM (2019). <https://doi.org/10.1145/3331545.3342601>, <https://doi.org/10.1145/3331545.3342601>
13. Finkbeiner, B., Klein, F., Piskac, R., Santolucito, M.: Temporal Stream Logic: Synthesis Beyond the Booleans. In: Dillig, I., Tasiran, S. (eds.) *Computer Aided Verification - 31st International Conference, CAV 2019, New York City, NY, USA, July 15-18, 2019, Proceedings, Part I*. *Lecture Notes in Computer Science*, vol. 11561, pp. 609–629. Springer (2019). [https://doi.org/10.1007/978-3-030-25540-4\\_35](https://doi.org/10.1007/978-3-030-25540-4_35)
14. Fourswords: Simple auction reference contract. <https://fourswords.io/docs/smartcontracts/solidity-guide/solidity-examples/> (2022), accessed: 2022-07-05
15. Fourswords: Voting reference contract. <https://fourswords.io/docs/smartcontracts/solidity-guide/solidity-examples/> (2022), accessed: 2022-07-05
16. Geier, G., Heim, P., Klein, F., Finkbeiner, B.: Syntroids: Synthesizing a game for fpgas using temporal logic specifications. *CoRR* **abs/2101.07232** (2021), <https://arxiv.org/abs/2101.07232>
17. Havelund, K., Rosu, G.: Synthesizing monitors for safety properties. In: Katoen, J., Stevens, P. (eds.) *Tools and Algorithms for the Construction and Analysis of Systems*, 8th International Conference, TACAS 2002, Held as Part of the Joint

- European Conference on Theory and Practice of Software, ETAPS 2002, Grenoble, France, April 8-12, 2002, Proceedings. Lecture Notes in Computer Science, vol. 2280, pp. 342–356. Springer (2002). [https://doi.org/10.1007/3-540-46002-0\\_24](https://doi.org/10.1007/3-540-46002-0_24), [https://doi.org/10.1007/3-540-46002-0\\_24](https://doi.org/10.1007/3-540-46002-0_24)
18. Havelund, K., Rosu, G.: Synthesizing monitors for safety properties. In: Katoen, J., Stevens, P. (eds.) Tools and Algorithms for the Construction and Analysis of Systems, 8th International Conference, TACAS 2002, Held as Part of the Joint European Conference on Theory and Practice of Software, ETAPS 2002, Grenoble, France, April 8-12, 2002, Proceedings. Lecture Notes in Computer Science, vol. 2280, pp. 342–356. Springer (2002). [https://doi.org/10.1007/3-540-46002-0\\_24](https://doi.org/10.1007/3-540-46002-0_24), [https://doi.org/10.1007/3-540-46002-0\\_24](https://doi.org/10.1007/3-540-46002-0_24)
  19. Hopcroft, J.: An  $n \log n$  algorithm for minimizing states in a finite automaton. In: Theory of machines and computations, pp. 189–196. Elsevier (1971)
  20. Jacobs, S., Bloem, R.: Parameterized synthesis. *Log. Methods Comput. Sci.* **10**(1) (2014). [https://doi.org/10.2168/LMCS-10\(1:12\)2014](https://doi.org/10.2168/LMCS-10(1:12)2014), [https://doi.org/10.2168/LMCS-10\(1:12\)2014](https://doi.org/10.2168/LMCS-10(1:12)2014)
  21. Kalra, S., Goel, S., Dhawan, M., Sharma, S.: ZEUS: analyzing safety of smart contracts. In: 25th Annual Network and Distributed System Security Symposium, NDSS 2018, San Diego, California, USA, February 18-21, 2018. The Internet Society (2018), [http://wp.internet-society.org/ndss/wp-content/uploads/sites/25/2018/02/ndss2018\OT1\ss\\_09-1.Kalra\\_paper.pdf](http://wp.internet-society.org/ndss/wp-content/uploads/sites/25/2018/02/ndss2018\OT1\ss_09-1.Kalra_paper.pdf)
  22. Khalimov, A., Jacobs, S., Bloem, R.: PARTY parameterized synthesis of token rings. In: Sharygina, N., Veith, H. (eds.) Computer Aided Verification - 25th International Conference, CAV 2013, Saint Petersburg, Russia, July 13-19, 2013. Proceedings. Lecture Notes in Computer Science, vol. 8044, pp. 928–933. Springer (2013). [https://doi.org/10.1007/978-3-642-39799-8\\_66](https://doi.org/10.1007/978-3-642-39799-8_66), [https://doi.org/10.1007/978-3-642-39799-8\\_66](https://doi.org/10.1007/978-3-642-39799-8_66)
  23. Khalimov, A., Jacobs, S., Bloem, R.: Towards efficient parameterized synthesis. In: Giacobazzi, R., Berdine, J., Mastroeni, I. (eds.) Verification, Model Checking, and Abstract Interpretation, 14th International Conference, VMCAI 2013, Rome, Italy, January 20-22, 2013. Proceedings. Lecture Notes in Computer Science, vol. 7737, pp. 108–127. Springer (2013). [https://doi.org/10.1007/978-3-642-35873-9\\_9](https://doi.org/10.1007/978-3-642-35873-9_9), [https://doi.org/10.1007/978-3-642-35873-9\\_9](https://doi.org/10.1007/978-3-642-35873-9_9)
  24. Klein, F., Santolucito, M.: Tsl tools. <https://github.com/kleinreact/tsltools> (2019)
  25. Kupferman, O., Vardi, M.Y.: Model checking of safety properties. In: Halbwachs, N., Peled, D.A. (eds.) Computer Aided Verification, 11th International Conference, CAV '99, Trento, Italy, July 6-10, 1999, Proceedings. Lecture Notes in Computer Science, vol. 1633, pp. 172–183. Springer (1999). [https://doi.org/10.1007/3-540-48683-6\\_17](https://doi.org/10.1007/3-540-48683-6_17), [https://doi.org/10.1007/3-540-48683-6\\_17](https://doi.org/10.1007/3-540-48683-6_17)
  26. Lahiri, S.K., Chen, S., Wang, Y., Dillig, I.: Formal specification and verification of smart contracts for azure blockchain. *CoRR* **abs/1812.08829** (2018), <http://arxiv.org/abs/1812.08829>
  27. Lamport, L.: Proving the correctness of multiprocess programs. *IEEE Trans. Software Eng.* **3**(2), 125–143 (1977). <https://doi.org/10.1109/TSE.1977.229904>, <https://doi.org/10.1109/TSE.1977.229904>
  28. Lisitsa, A., Potapov, I.: Temporal logic with predicate lambda-abstraction. In: 12th International Symposium on Temporal Representation and Reasoning (TIME 2005), 23-25 June 2005, Burlington, Vermont, USA. pp. 147–155. IEEE Computer Society (2005). <https://doi.org/10.1109/TIME.2005.34>, <https://doi.org/10.1109/TIME.2005.34>

29. López-Pintado, O., García-Bañuelos, L., Dumas, M., Weber, I., Ponomarev, A.: Caterpillar: A business process execution engine on the ethereum blockchain. *Softw. Pract. Exp.* **49**(7), 1162–1193 (2019). <https://doi.org/10.1002/spe.2702>, <https://doi.org/10.1002/spe.2702>
30. Maderbacher, B., Bloem, R.: Reactive synthesis modulo theories using abstraction refinement. *CoRR* **abs/2108.00090** (2021), <https://arxiv.org/abs/2108.00090>
31. Markgraf, O., Hong, C., Lin, A.W., Najib, M., Neider, D.: Parameterized synthesis with safety properties. In: d. S. Oliveira, B.C. (ed.) *Programming Languages and Systems - 18th Asian Symposium, APLAS 2020, Fukuoka, Japan, November 30 - December 2, 2020, Proceedings. Lecture Notes in Computer Science*, vol. 12470, pp. 273–292. Springer (2020). [https://doi.org/10.1007/978-3-030-64437-6\\_14](https://doi.org/10.1007/978-3-030-64437-6_14), [https://doi.org/10.1007/978-3-030-64437-6\\_14](https://doi.org/10.1007/978-3-030-64437-6_14)
32. Mavridou, A., Laszka, A.: Designing secure ethereum smart contracts: A finite state machine based approach. In: Meiklejohn, S., Sako, K. (eds.) *Financial Cryptography and Data Security - 22nd International Conference, FC 2018, Nieuwpoort, Curaçao, February 26 - March 2, 2018, Revised Selected Papers. Lecture Notes in Computer Science*, vol. 10957, pp. 523–540. Springer (2018). [https://doi.org/10.1007/978-3-662-58387-6\\_28](https://doi.org/10.1007/978-3-662-58387-6_28), [https://doi.org/10.1007/978-3-662-58387-6\\_28](https://doi.org/10.1007/978-3-662-58387-6_28)
33. Mavridou, A., Laszka, A., Stachtari, E., Dubey, A.: Verisolid: Correct-by-design smart contracts for ethereum. In: Goldberg, I., Moore, T. (eds.) *Financial Cryptography and Data Security - 23rd International Conference, FC 2019, Frigate Bay, St. Kitts and Nevis, February 18-22, 2019, Revised Selected Papers. Lecture Notes in Computer Science*, vol. 11598, pp. 446–465. Springer (2019). [https://doi.org/10.1007/978-3-030-32101-7\\_27](https://doi.org/10.1007/978-3-030-32101-7_27), [https://doi.org/10.1007/978-3-030-32101-7\\_27](https://doi.org/10.1007/978-3-030-32101-7_27)
34. Mayr, R.: Undecidable problems in unreliable computations. *Theor. Comput. Sci.* **297**(1-3), 337–354 (2003). [https://doi.org/10.1016/S0304-3975\(02\)00646-1](https://doi.org/10.1016/S0304-3975(02)00646-1), [https://doi.org/10.1016/S0304-3975\(02\)00646-1](https://doi.org/10.1016/S0304-3975(02)00646-1)
35. Mealy, G.H.: A method for synthesizing sequential circuits. *The Bell System Technical Journal* **34**(5), 1045–1079 (1955). <https://doi.org/10.1002/j.1538-7305.1955.tb03788.x>
36. Nehai, Z., Piriou, P.Y., Dumas, F.: Model-checking of smart contracts. In: 2018 IEEE International Conference on Internet of Things (iThings) and IEEE Green Computing and Communications (GreenCom) and IEEE Cyber, Physical and Social Computing (CPSCom) and IEEE Smart Data (SmartData). pp. 980–987. IEEE (2018)
37. Open Zeppelin: Erc20 token system documentation from open zeppelin. <https://docs.openzeppelin.com/contracts/2.x/api/token/erc20> (2022), accessed: 2022-07-05
38. Open Zeppelin: Erc20 token system reference contract. <https://github.com/OpenZeppelin/openzeppelin-contracts/blob/9b3710465583284b8c4c5d2245749246bb2e0094/contracts/token/ERC20/ERC20.sol> (2022), accessed: 2022-07-05
39. Permenev, A., Dimitrov, D.K., Tsankov, P., Drachsler-Cohen, D., Vechev, M.T.: Verx: Safety verification of smart contracts. In: 2020 IEEE Symposium on Security and Privacy, SP 2020, San Francisco, CA, USA, May 18-21, 2020. pp. 1661–1677. IEEE (2020). <https://doi.org/10.1109/SP40000.2020.00024>, <https://doi.org/10.1109/SP40000.2020.00024>

40. Program the Blockchain: Coin funding reference contract. <https://programtheblockchain.com/posts/2018/01/19/writing-a-crowdfunding-contract-a-la-kickstarter/> (2022), accessed: 2022-07-05
41. Program the Blockchain: Coin toss reference contract. <https://programtheblockchain.com/posts/2018/03/16/flipping-a-coin-in-ethereum/> (2022), accessed: 2022-07-05
42. Solidity: Simple auction protocol from the solidity documentation. <https://docs.soliditylang.org/en/v0.6.8/solidity-by-example.html#simple-open-auction> (2021), accessed: 2022-07-05
43. Solidity: Blinded auction reference contract. <https://docs.soliditylang.org/en/v0.5.11/solidity-by-example.html#id2> (2022), accessed: 2022-07-05
44. Stephens, J., Ferles, K., Mariano, B., Lahiri, S.K., Dillig, I.: Smartpulse: Automated checking of temporal properties in smart contracts. In: 42nd IEEE Symposium on Security and Privacy, SP 2021, San Francisco, CA, USA, 24-27 May 2021. pp. 555–571. IEEE (2021). <https://doi.org/10.1109/SP40001.2021.00085>, <https://doi.org/10.1109/SP40001.2021.00085>
45. Suvorov, D., Ulyantsev, V.: Smart contract design meets state machine synthesis: Case studies. CoRR **abs/1906.02906** (2019), <http://arxiv.org/abs/1906.02906>
46. Wang, Y., Lahiri, S.K., Chen, S., Pan, R., Dillig, I., Born, C., Naseer, I., Ferles, K.: Formal verification of workflow policies for smart contracts in azure blockchain. In: Chakraborty, S., Navas, J.A. (eds.) Verified Software. Theories, Tools, and Experiments - 11th International Conference, VSTTE 2019, New York City, NY, USA, July 13-14, 2019, Revised Selected Papers. Lecture Notes in Computer Science, vol. 12031, pp. 87–106. Springer (2019). [https://doi.org/10.1007/978-3-030-41600-3\\_7](https://doi.org/10.1007/978-3-030-41600-3_7), [https://doi.org/10.1007/978-3-030-41600-3\\_7](https://doi.org/10.1007/978-3-030-41600-3_7)
47. Zupan, N., Kasinathan, P., Cuellar, J., Sauer, M.: Secure smart contract generation based on petri nets. In: Blockchain Technology for Industry 4.0, pp. 73–98. Springer (2020)

## A Proof of Lemma 1

**Lemma 1.** *For every instance  $\mu$ ,  $\text{traces}_\mu(\mathcal{M}) = \text{traces}(\mathcal{W}_\mu)$ .*

In the following, we show that  $\text{traces}_\mu(\mathcal{M}) \cup \text{finTraces}_\mu(\mathcal{M}) = \text{traces}(\mathcal{W}_\mu) \cup \text{finTraces}(\mathcal{W}_\mu)$ , where  $\text{finTraces}(\cdot)$  and  $\text{finTraces}_\mu(\cdot)$  are the sets of finite prefixes of the traces in  $\text{traces}(\cdot)$  and  $\text{traces}_\mu(\cdot)$ , respectively. From this, the original statement of the lemma follows. We therefore have to show that every (finite or infinite) run  $r_{\mathcal{M}}$  of  $\mathcal{M}$  can be matched with a (finite or infinite) run  $r_{\mathcal{W}_\mu}$  of  $\mathcal{W}_\mu$  (and vice versa) such that  $\text{trace}_\mu(r_{\mathcal{M}}) = \text{trace}(r_{\mathcal{W}_\mu})$ . For the first direction of the equality, we show that every transition of  $r_{\mathcal{M}}$  can be either matched with a transition in  $\mathcal{W}_\mu$  or constitutes a step which is removed in the trace  $t_\mu = \text{trace}(r_{\mathcal{M}})_\mu$ . For the other direction, we show that every transition of  $r_{\mathcal{W}_\mu}$  can be matched with a transition in  $\mathcal{M}$ . Assume  $\mathcal{M}$  is currently in state  $s = (f_1, \dots, f_n)$  of  $r_{\mathcal{M}}$  and  $\mathcal{W}_\mu$  is in state  $s_{\mathcal{W}}$  of  $r_{\mathcal{W}_\mu}$ . We keep the invariant that  $s_{\mathcal{W}} \in K_\mu = \bigcap_{1 \leq j \leq n} K_j(f_j(P_j[\mu]))$ .

- For the first direction, assume that the next transition of  $\mathcal{M}$  is  $(s, A, s')$  with  $s' = (f'_1, \dots, f'_n)$ . Let  $B \subseteq AP$  be such that  $B[\mu] = A_\mu$ . By construction of  $\mathcal{M}$ , there is an instance  $\mu'$  and parameter subset  $P_i$  such that there is exactly one  $\mathbf{f}(P_i[\mu']) \in A$ . Let  $B' \subseteq AP$  be such that  $B'[\mu'] = A_{\mu'}$ . We distinguish two cases.

- Assume  $P_i[\mu] = P_i[\mu']$ . Let  $s_i = f_i(P_i[\mu'])$  and  $s'_i = f'_i(P_i[\mu'])$ . Let  $G_{(s_i, B', s'_i)} = \{s \mid (s_i, B' \cup \{\text{in-}s\}, s'_i) \in \delta_i\}$ . Let  $P_{j_1}, \dots, P_{j_l}$  be all subsets of  $P_i$  and  $K = K_i(s_i) \cap \bigcap_{1 \leq k \leq l} K_{j_k}(f_{j_k}(P_{j_k}[\mu']))$ . By definition of the independence check employed to construct  $\mathcal{M}$ ,  $K \subseteq G_{(s_i, B', s'_i)}$ . As  $P_i[\mu] = P_i[\mu']$ , we have that  $K_\mu \subseteq K$  and thus also  $K_\mu \subseteq G_{(s_i, B', s'_i)}$ . Therefore, by the invariant,  $s_{\mathcal{W}} \in G_{(s_i, B', s'_i)}$  and there is a transition  $(s_{\mathcal{W}}, B', s'_{\mathcal{W}})$  in  $\mathcal{W}$ .

Now, if  $\mu' = \mu$ , there is a transition  $(s_{\mathcal{W}}, B[\mu], s'_{\mathcal{W}})$  in  $\mathcal{W}_\mu$ , what shows that the step in  $\mathcal{M}$  can be mirrored in  $\mathcal{W}_\mu$ . Otherwise, if  $\mu' \neq \mu$ ,  $B$  and  $B'$  agree on the propositions parameterized with  $P_i$ . By construction of  $\mathcal{M}$ ,  $B$  has all inputs not parameterized with  $P_i$  set to *false* and only self-updates of cells not parameterized with  $P_i$  set to *true*. By Requirements 1 and 2, since there is a transition  $(s_{\mathcal{W}}, B', s'_{\mathcal{W}})$  in  $\mathcal{W}$ , there is also a transition  $(s_{\mathcal{W}}, B, s'_{\mathcal{W}})$  in  $\mathcal{W}$  and therefore a transition  $(s_{\mathcal{W}}, B[\mu], s'_{\mathcal{W}})$  in  $\mathcal{W}_\mu$ .

All that remains to show is that the invariant is preserved by both transitions  $(s_{\mathcal{W}}, B[\mu], s'_{\mathcal{W}})$  and  $(s_{\mathcal{W}}, B'[\mu'], s'_{\mathcal{W}})$ . That means that we need to show that  $s'_{\mathcal{W}} \in K'_\mu$  for  $K'_\mu = \bigcap_{1 \leq j \leq n} K_j(f'_j(P_j[\mu]))$  in either case. Since  $s_{\mathcal{W}} \in K_\mu$ , we have that for every  $j$ ,  $s_{\mathcal{W}} \in K_j(f_j(P_j[\mu]))$ . Furthermore,  $\mathbf{f}(P_i) \in B$  and  $\mathbf{f}(P_i) \in B'$ . For  $j \neq i$  we have that  $f'_j = f_j$  and, therefore, by Proposition 3,  $s'_{\mathcal{W}} \in K_j(f'_j(P_j[\mu]))$ . Furthermore,  $K_\mu \subseteq G_{(s_i, B', s'_i)}$  and  $(s_i, C \cup \{\text{in-}s_{\mathcal{W}}\}, s'_i) \in \delta_i$ , for both  $C = B$  and  $C = B'$ . Therefore, again by Proposition 3,  $s'_{\mathcal{W}} \in f_i[P_i[\mu] \mapsto s'_i]$ . Thus,  $s'_{\mathcal{W}} \in K_j(f'_j(P_j[\mu]))$  for all  $j$  and the invariant holds for  $s'_{\mathcal{W}}$ .



- Assume  $P_i[\mu] \neq P_i[\mu']$ . By construction of  $\mathcal{M}$ , there is no  $\mathbf{g}(P_j) \in B$ . As  $\mathcal{W}$  satisfies Requirement 1, so does every  $\mathcal{W}_i$ . Together with how  $\mathcal{M}$  is constructed, it follows that  $B|_O \subseteq O_{self}$ . Thus, by definition of  $t_\mu$ , the transition will not appear in  $t_\mu$  and  $\mathcal{W}_\mu$  thus stays in its current state. It remains to show that the invariant is maintained by the transition in  $\mathcal{M}$ . Since  $P_i[\mu] \neq P_i[\mu']$ , we have by definition of  $\mathcal{M}$  that  $f_j(P_j[\mu]) = f'_j(P_j[\mu])$  for all  $j$ , and therefore the invariant is maintained.
- For the second direction, assume that the next transition of  $\mathcal{W}_\mu$  is  $(s_{\mathcal{W}}, B[\mu], s'_{\mathcal{W}})$  for some  $B \subseteq AP$ . By construction of  $\mathcal{W}$ , there is a parameter subset  $P_i$  such that some  $\mathbf{f}(P_i) \in B$ . By our invariant,  $s_{\mathcal{W}} \in K_i(s_i)$ , therefore, by Proposition 2, there is a transition  $(s_i, B \cup \{\mathbf{in}_{s_{\mathcal{W}}}\}, s'_i) \in \delta_i$ . By construction of  $\mathcal{M}$ , there is a transition from  $(f_1, \dots, f_n)$  to  $(f'_1, \dots, f'_n)$  such that  $f'_i(P_i[\mu]) = s'_i$ . Furthermore, the transition is labeled with  $A$  such that  $A_\mu = B[\mu]$ . It remains to show that the invariant is maintained by the transition. For  $j \neq i$ ,  $f'_j = f_j$  and therefore, by Proposition 3,  $s'_{\mathcal{W}} \in K_j(f_j(P_j[\mu]))$ . Furthermore, since  $(s_i, B \cup \{\mathbf{in}_{s_{\mathcal{W}}}\}, s'_i) \in \delta_i$ , again by Proposition 3, it holds that  $s'_{\mathcal{W}} \in K_i(s'_i)$  and therefore, by definition of  $f'_i = f_i[P_i[\mu] \mapsto s'_i]$ ,  $s'_{\mathcal{W}} \in K_i(f'_i(P_i[\mu]))$ . Therefore, the invariant is maintained for the transition.

## B Description of Benchmarks

- *Asset Transfer* (not parameterized, reference contract: [4]). This contract originates from Microsoft’s Azure Blockchain Workbench [3]. It has been formulated similarly in [46,8].
- *Blinded Auction* (not parameterized, reference contract: [43]). The blinded auction protocol, following [32], describes how to sell an item to the highest bidder. The bids of all bidders are hashed and are only revealed after the auction closed.
- *Coin Toss* (not parameterized, reference contract: [41]). In this protocol, following [3], two users bet, together with a wager, on whether a coin toss results in heads or tails. After reaching a time limit, the winner receives both wagers.
- *Crowd Funding* (not parameterized, reference contract: [40]). In this contract, following [45], users can donate coins until a time limit is reached. If the funding goal is reached by the end, the owner retrieves the donations. Otherwise, the users can reclaim their donations.
- *ERC20 Token System* (parameterized, reference contract: [38]). We consider a classical ERC20 token system following the Open Zeppelin documentation [37] but without pausing feature.
- *ERC20 Token System Extended* (parameterized). This is the contract described in this paper which extends the first ERC20 token system with different methods for pausing.
- *NFT Auction* (parameterized, reference contract: [2]). This is a comparatively large contract from Avolabs that implements an NFT auction combined with a buy-now feature. The original implementation has over 1400

lines of code. We specified the main requirements on the control flow as described in the README of Avolabs' GitHub.

- *Simple Auction* (not parameterized, reference contract: [14]). The simple auction protocol is a common example for smart contract control flows, see, e.g., [42]. It is similar to the blinded auction without the bids being hashed.
- *Ticket System* (not parameterized, reference contract written by the authors of this paper). The ticket system contract describes the selling process of tickets. As long as tickets are available and the sale has not been closed, users can buy tickets. Users can return their tickets while the ticket sale is still open.
- *Voting* (parameterized, reference contract: [15]). This is a simple voting contract which enforces that every voter vote only votes once. Additionally, we let the owner close the contract only after a fixed number of votes is reached.