Model-checking Ethereum smart contracts written in Vyper

Francisco Moreira da Silva Rogado Domingues

Instituto Superior Técnico, Lisboa, Portugal

December 2019

Abstract

Ethereum was the first blockchain platform to introduce the concept of smart contract, a computer program stored and executed in the blockchain. Smart contracts are capable of holding digital assets and communicating with each other, making them ideal for implementing decentralised automated financial applications. Since smart contracts are immutable once deployed and perform transactions of assets with monetary value, it is of most importance that they behave as expected and are resilient to attacks. The best way of assuring both is through formal verification. The objective of this work was to develop a tool to automatically translate smart contracts written in Vyper to the nuXmv language. Vyper is a new programming language for Ethereum smart contracts, designed to be simple, safe and easy to audit, while nuXmv is a model checker used to verify invariant and temporal properties. This translation allows one to use nuXmv's model checking algorithms to attempt the verification of smart-contracts' properties specified in that format. Considering the limitations of the nuXmv language, we defined a sublanguage of Vyper, named MiniVyper, and defined its concrete syntax and operational semantics, the latter serving as basis for the translation. This work represents not only one of the first attempts at defining a formal verification technique specific for Vyper smart contracts, but also one of the first formal descriptions of a subset of this language. We tested this verification method by translating and verifying properties on real smart-contracts and were able to detect some design flaws.

Keywords: Ethereum, blockchain, Vyper, model checking, formal verification, operational semantics

1. Introduction

Blockchains are decentralised digital ledgers managed by networks of mutually distrusting nodes. Transactions between parties are recorded in structures called blocks, which are linked to each other by cryptographic hashes and together form the blockchain. After a block's validation and integration in the blockchain, its data cannot be changed without altering the posterior blocks, which requires a consensus of the network majority. Blockchains are, therefore, considered append-only data structures.

The Ethereum blockchain platform was first described by Vitalik Buterin in his white paper ([3]), and released in 2015. Even though it can be used just as a cryptocurrency, Ethereum also introduced the concept of smart contract. An Ethereum smart contract is a computer program that is stored and executed in the Ethereum blockchain. The code of Ethereum smart contracts is written in a Turing-complete bytecode language and executed in the Ethereum Virtual Machine (EVM)([8]). Naturally, developers usually write smart contracts in high-level languages, the most popular being Solidity and Vyper ([4],[5]).

Just like any other computer program, smart contracts are vulnerable to errors. However, since these contracts deal directly with funds, these errors may have extremely expensive consequences. Therefore, making sure smart contracts work as expected is an essential task. This fact is even more clear considering the immutable nature of the blockchain, as errors in a smart contract are impossible to correct after its deployment to the blockchain.

Besides assuring the lack of execution errors, smart contracts should also be protected against potential attacks. Vulnerabilities in smart contracts have been exploited in order to deviate funds, the most notable example being the 'DAO attack', in which a set of smart contracts implementing a decentralised autonomous organisation (DAO) were drained of the equivalent to $60M in cryptocurrency.

One possible solution to this problem is formal verification. In this work, we consider the model checking approach, in which specifications are proved or disproved by analysing all of the model's possible states.

The main goal of this work is provide a way to formally verify specifications concerning the execution

of Ethereum smart contracts written in the high-level language Vyper. These specifications can either be invariant properties or temporal properties specified in the Linear Time Logic (LTL) or the Computer Tree Logic (CTL). We start by defining a special class of Kripke structures, named *quasi-deterministic*, where the expressiveness of both logics coincides. Then, we define a sublanguage of Vyper, called MiniVyper, along with its operational semantics, based on Kripke structures. Using this semantics, we built a tool which automatically translates the source code of a MiniVyper smart contract to the nuXmv model checker's language ([2]). Two different models are considered: one represents the execution of a single call to the smart contract, while the other represents the execution of an infinite series of calls. Using nuXmv, we were able to test several properties in both models of real smart contracts and detect some design flaws.

This article is structured as follows. Section 2 provides basic concepts concerning the Ethereum blockchain and temporal logic. In Section 3, we define the Vyper sublanguage, MiniVyper and its operational semantics. In Section 4 we explain how to translate MiniVyper smart contracts to the nuXmv language and show an example of a detected vulnerability. Finally, in Section 5 we summarise our achievements and present ideas for future work.

2. Background

2.1 The Ethereum blockchain

One may think of the Ethereum blockchain as a state transition machine, where states are made up of objects called *accounts* and transitions are *transactions* between accounts.

Each Ethereum account is uniquely identified in the blockchain by a 160-bit *address* and contains four fields. The first is the current Ether *balance*. Ether is Ethereum's cryptocurrency and is needed to pay transaction fees. The balance is specified in Ether's smallest unit, Wei[1]. The second field is the account's *storage*, initially empty. The third field is the *contract code*, which contains the EVM bytecode executed when the account receives a transaction. If an account's code is empty, we say it is *externally owned*, otherwise we call it a *contract*. This is the only immutable field of an account. Finally, the last field is the *nonce*, a counter of outgoing transactions. We say that the *state of an account* is its current values for the balance, storage and nonce, and that the *global state* of the Ethereum blockchain is the combination of all account states.

Changes in the global state happen through *transactions*. A transaction can only be generated by an externally owned account and can either be a *message call*, used to transfer funds and, if they target a contract, to execute its code, or a *contract creation*. The execution of every transaction in Ethereum incurs a fee, which is paid in *gas*. The actual cost of a transaction in Ether is the amount of gas spent times the gas price, which is set by the sender. When a transaction is broadcast to the Ethereum network, it is eventually picked up by special nodes called *miners*, which bundle it with other transactions in structures called *blocks*. When constructing a block, a miner must execute and validate all transactions, thus defining the next state of the blockchain if the block is approved, and perform a proof of computational work.

The execution model of a transaction is based on the already mentioned EVM, which is essentially a stack-based machine operating on 256-bit integers. The elements of its language are called opcodes and every opcode has an associated cost in gas. If the gas required for the transaction's executions is greater than the gas limit set by the sender, then the transaction fails and all state changes are discarded. Besides the stack, the EVM includes a volatile memory, can read and write the executing contract's storage, and access its balance and environmental information like the message and block components.

2.2 Basic concepts of temporal logic

We start this section by presenting the definition of a Kripke structure, one of the most used types of models in model checking. In the following, given a non-empty and finite set Σ (alphabet), we call an infinite sequence $\rho = v_1 v_2 ...$ an *infinite word* over Σ if $v_i \in \Sigma$, for every $i \in \mathbb{N}$.

Definition 1. Given a set AP of atomic propositions, we define a *Kripke structure* over AP as a tuple $K = (S, S_0, \longrightarrow, L)$, where

- S is a non-empty finite set of states;
- $S_0 \subset S$ is the set of initial states;
- $\longrightarrow \subset S \times S$ is a transition relation between states such that $\forall s \in S \; \exists s' \in S : s \longrightarrow s'$;
- $L : S \to 2^{AP}$ is the labelling function that indicates which atomic propositions are true in each state.

Given a Kripke structure $K = (S, S_0, \longrightarrow, L)$, we define a *path* of K as an infinite sequence $s_0 s_1 s_2 ...$ of states of K, such that $s_i \longrightarrow s_{i+1}$, for all $i \in \mathbb{N}$. We normally denote a path by the letter π and write $\pi[i]$ to refer the i-th state of π. A *complete path* is any path π_0 such that $\pi_0[0] \in S_0$. Finally, we define the trace of a path π as the sequence $trace(\pi) = L(\pi[0])L(\pi[1])L(\pi[2])...$, which is an infinite word over 2^{AP}.

Now, we define two special kinds of Kripke structures.

[1] 1 Ether = 10^{18} Wei

Definition 2. We say that a Kripke structure $K = (S, S_0, \longrightarrow, L)$ is *deterministic* if the following conditions hold:

1. For all $s \in S$, $|\{s' \in S : s \longrightarrow s'\}| = 1$;
2. $S_0 = \{s_0\}$, for some state $s_0 \in S$.

If only condition 1 is true, we call K *quasi-deterministic*.

Deterministic and *quasi*-deterministic Kripke structures are characterised by having exactly one outgoing transition per state. Hence, they have one and only one path starting at each state.

For the rest of this section, we discuss the temporal logics LTL and CTL, both commonly used for specifying properties of Kripke structures. We define their syntax and semantics in terms of these structures.

The LTL logic is called linear because its notion of time assumes that, for each instant, there is only one possible future. For a set of atomic propositions AP and $p \in AP$, LTL formulas over AP are inductively defined by the following grammar.

$$\varphi ::= True \mid p \mid \neg\varphi \mid \varphi_1 \wedge \varphi_2 \mid \mathsf{X}\varphi \mid (\varphi_1 \,\mathsf{U}\, \varphi_2)$$

Besides the usual operators of propositional logic, LTL has the additional *next* ('X') and *until* ('U') temporal operators. Intuitively, the formula '$\mathsf{X}\varphi$' means that 'φ will hold in the next instant', while the formula '$\varphi_1 \,\mathsf{U}\, \varphi_2$' means that 'either φ_2 is true now, or there is an instance in the future where φ_2 will hold and, until then, φ_1 holds'. From these, we define two other temporal operators: *eventually* (F), defined by $\mathsf{F}\varphi \equiv_{\text{def}} True \,\mathsf{U}\, \varphi$, and *always* ($\mathsf{G}$), defined by $\mathsf{G}\varphi \equiv_{\text{def}} \neg\mathsf{F}\neg\varphi$. Formulas of the kind '$\mathsf{F}\varphi$' mean that '$\varphi$ will hold in some future instant', while formulas of the kind '$\mathsf{G}\varphi$' mean that 'φ holds now and in every future instance'.

Next we present the semantics of LTL.

Definition 3. Let ρ be an infinite word over 2^{AP}, φ a LTL formula over AP and $i \in \mathbb{N}$. We call ρ an *interpretation* for LTL and define its satisfaction relation as follows:

(*a*) $\rho, i \Vdash True$;
(*b*) $\rho, i \Vdash p$ iff $p \in \rho[i]$, for $p \in AP$;
(*c*) $\rho, i \Vdash \neg\varphi$ iff $\rho, i \nVdash \varphi$;
(*d*) $\rho, i \Vdash \varphi_1 \wedge \varphi_2$ iff $\rho, i \Vdash \varphi_1$ and $\rho, i \Vdash \varphi_2$;
(*e*) $\rho, i \Vdash \mathsf{X}\varphi$ iff $\rho, i+1 \Vdash \varphi$;
(*f*) $\rho, i \Vdash \varphi_1 \mathsf{U} \varphi_2$ iff there is a $j \geq i$ such that $\rho, j \Vdash \varphi_2$ and $\rho, k \Vdash \varphi_1$, for every $i \leq k < j$.

We say that the interpretation ρ *satisfies* or *models* φ if $\rho, 0 \vDash \varphi$. Furthermore, given a Kripke structure $K = (S, S_0, \longrightarrow, L)$, π a path of K and $s \in S$ a state of K, the satisfiability of φ by K, π and s is defined as follows:

- $K, \pi \Vdash \varphi$ iff $trace(\pi) \Vdash \varphi$.
- $K, s \Vdash \varphi$ iff $K, \pi \Vdash \varphi$ for all paths π such that $\pi[0] = s$.
- $K \vDash \varphi$ iff $K, s_0 \Vdash \varphi$ for every $s_0 \in S_0$.

Note that the last condition is equivalent to $K, \pi_0 \Vdash \varphi$ for all complete paths π_0 of K.

We now introduce the CTL, which is based on a branching notion of time. This means that, for every instance of time, there may be various possible futures. For a set of atomic propositions AP and $p \in AP$, CTL formulas over AP are inductively defined by the following grammar.

$$\Phi ::= True \mid p \mid \neg\Phi \mid \Phi_1 \wedge \Phi_2 \mid \mathsf{AX}\Phi \mid \mathsf{EX}\Phi \mid \mathsf{A}(\Phi_1 \,\mathsf{U}\, \Phi_2) \mid \mathsf{E}(\Phi_1 \,\mathsf{U}\, \Phi_2)$$

Non propositional operators in CTL are always formed by a path quantifier, A or E, followed by a temporal operator, X or U. A is called the universal path quantifier, and, intuitively, formulas of the kind $\mathsf{AT}\Phi$ hold in a state s if $\mathsf{T}\Phi$ holds in all paths starting at s, where T is a temporal operator. On the other hand, E is the existential path quantifier and formulas of the kind $\mathsf{ET}\Phi$ hold in a state s if there is a path starting at s where $\mathsf{T}\Phi$ holds. Note that we can also add the F and G operators to this logic, combined with A and E, by defining them as before.

Unlike LTL, the semantics of CTL is defined in terms of states of Kripke structures.

Definition 4. Let $K = (S, S_0, \longrightarrow, L)$ be a Kripke structure, $s \in S$ a state of K and Φ a CTL formula over AP. The satisfiability of Φ by state s is inductively defined as follows:

a) $K, s \Vdash True$;
b) $K, s \Vdash p$ iff $p \in L(s)$, for $p \in AP$;
c) $K, s \Vdash \neg\Phi$ iff $K, s \nVdash \Phi$;
d) $K, s \Vdash \Phi_1 \wedge \Phi_2$ iff $K, s \Vdash \Phi_1$ and $K, s \Vdash \Phi_2$;
e) $K, s \Vdash \mathsf{AX}\Phi$ iff, for every path π such that $\pi[0] = s$, $K, \pi[1] \Vdash \Phi$;
f) $K, s \Vdash \mathsf{EX}\Phi$ iff there is a path π such that $\pi[0] = s$ and $K, \pi[1] \Vdash \Phi$;
g) $K, s \Vdash \mathsf{A}(\Phi_1 \,\mathsf{U}\, \Phi_2)$ iff, for every path π such that $\pi[0] = s$, there is a $j \geq 0$ such that $K, \pi[j] \Vdash \Phi_2$ and, for every $0 \leq k < j$, $K, \pi[k] \Vdash \Phi_1$;
h) $K, s \Vdash \mathsf{E}(\Phi_1 \,\mathsf{U}\, \Phi_2)$ iff there is path π such that $\pi[0] = s$, there is a $j \geq 0$ such that $K, \pi[j] \Vdash \Phi_2$ and, for every $0 \leq k < j$, $K, \pi[k] \Vdash \Phi_1$.

Once again, we say that K satisfies Φ, written $K \vDash \Phi$, iff $K, s_0 \Vdash \Phi$ for every $s_0 \in S_0$.

With both logics defined, we are interested in comparing their expressiveness. We say that a LTL formula φ and a CTL formula Φ are *equivalent* if, for

all Kripke structures K, we have that $K \vDash \varphi$ if and only if $K \vDash \Phi$. It is a well known fact that, in general, the expressiveness of these two logics is incomparable, meaning one can always find a LTL formula with no CTL equivalent and vice versa. See, for example, [1] for such proof. Nevertheless, if we only consider Kripke structures where no branching occurs, such has deterministic and *quasi*-deterministic structures, the difference between these two logics fades. We end this section by stating two results regarding this matter. First, in Lemma 1, we state that, for these kind of structures, there is no difference between the existential and the universal path operators in CTL. This result is a direct consequence of definition 4 lines *e)* to *h)*, and of the fact that there is one and only one path originating in each state of a (*quasi-*)deterministic Kripke structure. Then, we end this section by proving that LTL and CTL have the same expressiveness for these kind of structures.

Lemma 1. *The path operators* A *and* E *are semantically indistinguishable to (quasi-)deterministic Kripke structures. In other words, if* Φ_1 *and* Φ_2 *are CTL formulas,* K *is a (quasi-)deterministic Kripke structure and* s *is one of its states,* $K, s \Vdash \mathsf{AX}\,\Phi_1$ *if and only if* $K, s \Vdash \mathsf{EX}\,\Phi_1$ *and* $K, s \Vdash \mathsf{A}(\Phi_1 \,\mathsf{U}\, \Phi_2)$ *if and only if* $K, s \Vdash \mathsf{E}(\Phi_1 \,\mathsf{U}\, \Phi_2)$.

Theorem 2. *If* K *is a deterministic or quasi-deterministic Kripke structure and* s *is an arbitrary state of* K*,*

i) *for every CTL formula* Φ*,* $K, s \Vdash \Phi$ *if and only if* $K, s \Vdash \varphi_\Phi$*, where* φ_Φ *is the LTL formula obtained from* Φ *by omitting all of its path quantifiers. Particularly,* $K \vDash \Phi$ *if and only if* $K \vDash \varphi_\Phi$*;*

ii) *for every LTL formula* φ*,* $K, s \Vdash \varphi$ *if and only if* $K, s \Vdash \Phi_\varphi$*, where* Φ_φ *is the CTL formula obtained from* φ *by adding the* A *path quantifier before every* X *or* U *operator. Particularly,* $K \vDash \varphi$ *if and only if* $K \vDash \Phi_\varphi$.

Proof. Both proofs follow by induction on the structure of the formulas. Note that, in particular, the equivalences hold in all of the initial states. This justifies the second part of the claims.

i) Base case: $\Phi = p$, for $p \in AP$. Then $\varphi_\Phi = p$, the two formulas coincide and the equivalence is trivial.
Induction hypothesis (†): $K, s \Vdash \Phi_i$ iff $K, s \Vdash \varphi_{\Phi_i}$, for $i = 1, 2$.
(a) $\Phi = \neg\, \Phi_1$. Then $\varphi_\Phi = \neg\, \varphi_{\Phi_1}$ and $K, s \Vdash \Phi$ iff $K, s \nVdash \Phi_1$ iff $^{(\dagger)}$ $K, s \nVdash \varphi_{\Phi_1}$ iff $K, s \Vdash \varphi_\Phi$.
(b) $\Phi = \Phi_1 \wedge \Phi_2$. Then $\varphi_\Phi = \varphi_{\Phi_1} \wedge \varphi_{\Phi_2}$ and $K, s \Vdash \Phi$ iff $K, s \Vdash \Phi_1$ and $K, s \Vdash \Phi_2$ iff $^{(\dagger)}$ $K, s \Vdash \varphi_{\Phi_1}$ and $K, s \Vdash \varphi_{\Phi_2}$ iff $K, s \Vdash \varphi_\Phi$.
(c) $\Phi = \mathsf{AX}\Phi_1$. Then $\varphi_\Phi = \mathsf{X}\varphi_{\Phi_1}$. Let π be the unique path starting at s. Therefore, $K, s \Vdash \Phi$ iff $^{(*)}$ $K, \pi[1] \Vdash \Phi_1$ iff $^{(\dagger)}$ $K, \pi[1] \Vdash \varphi_{\Phi_1}$ iff $^{(*)}$ $trace(\pi), 1 \Vdash \varphi_{\Phi_1}$ iff $trace(\pi), 0 \Vdash \mathsf{X}\varphi_{\Phi_1}$ iff $^{(*)}$ $K, s \Vdash \varphi_\Phi$, where the (*) equivalences are consequence of K being (*quasi-*)deterministic and π being the unique path starting at s.
(d) $\Phi = \mathsf{A}(\Phi_1 \,\mathsf{U}\, \Phi_2)$. Then $\varphi_\Phi = \varphi_{\Phi_1} \mathsf{U} \varphi_{\Phi_2}$. Once again, let π be the unique path starting at s.
($\Rightarrow$) Suppose that $K, s \Vdash \Phi$. This means that there is a $j \geq 0$ such that $K, \pi[j] \Vdash \Phi_2$ and $K, \pi[k] \Vdash \Phi_1$ for every $0 \leq k < j$. By (†), we also have that $K, \pi[j] \Vdash \varphi_{\Phi_2}$ and $K, \pi[k] \Vdash \varphi_{\Phi_1}$ for every $0 \leq k < j$. As π is a path crossing this states, we have that $trace(\pi), j \Vdash \varphi_{\Phi_2}$ and $trace(\pi), k \Vdash \varphi_{\Phi_1}$ for every $0 \leq k < j$, which implies that $trace(\pi), 0 \Vdash \varphi_{\Phi_1} \,\mathsf{U}\, \varphi_{\Phi_2}$. We finally conclude, by uniqueness of π, that $K, s \Vdash \varphi_{\Phi_1} \,\mathsf{U}\, \varphi_{\Phi_2}$.
($\Leftarrow$) Suppose now that $K, s \Vdash \varphi_\Phi$. Then, $trace(\pi), 0 \Vdash \varphi_{\Phi_1} \,\mathsf{U}\, \varphi_{\Phi_2}$ and so, there must be a $j \geq 0$ such that $trace(\pi), j \Vdash \varphi_{\Phi_2}$ and $trace(\pi), k \Vdash \varphi_{\Phi_1}$, for every $0 \leq k < j$. Because of the lack of bifurcations, we are certain that any path crossing a state $\pi[i]$, for $0 \leq i \leq j$, will also coincide with π from that point on. Consequently, we can affirm that $K, \pi[j] \Vdash \varphi_{\Phi_2}$ and $K, \pi[k] \Vdash \varphi_{\Phi_1}$, for every $0 \leq k < j$. By (†), these states also satisfy Φ_2 and Φ_1, respectively, and we conclude, by uniqueness of π, that $K, s \Vdash \Phi$.
(e) $\Phi = \mathsf{EX}\,\Phi_1$. Then $\varphi_\Phi = \mathsf{X}\,\varphi_{\Phi_1}$ and the result follows from case (c) and the previous lemma.
(f) $\Phi = \mathsf{E}(\Phi_1 \,\mathsf{U}\, \Phi_2)$. Then $\varphi_\Phi = \varphi_{\Phi_1} \,\mathsf{U}\, \varphi_{\Phi_2}$ and the result follows from case (d) and the previous lemma.

ii) The base case and the Boolean cases are analogous to the previous ones. Furthermore if $\varphi = \mathsf{X}\varphi_1$ then $\Phi_\varphi = \mathsf{AX}\,\Phi_{\varphi_1}$ and the result's proof is analogous to **(c)**. Similarly, if $\varphi = \varphi_1 \,\mathsf{U}\, \varphi_2$, then $\Phi_\varphi = \mathsf{A}(\Phi_{\varphi_1} \,\mathsf{U}\, \Phi_{\varphi_2})$ and the equivalence is proved in **(d)**.

□

3. MiniVyper

Vyper is a new Python-based programming language for Ethereum smart contracts. The goal of this still under development language is to offer a simpler and safer alternative to Solidity, the most common Ethereum smart contract language. Flaws in the Solidity language have been among the main causes of attacks on smart contracts in the Ethereum blockchain. Therefore, it was essential to create a new language that was less error prone and easier to audit. Some of Vyper's features aiming at

these objectives include built-in overflow checking on arithmetic operations and disallowing recursion and infinite loops. Only Python-style for-loops over elements of fixed sized lists are permitted, allowing the computation of precise upper bound for gas consumption.

Note that the restriction to limited for-loops aids our goal of model checking Vyper contracts, as it guarantees that their execution always halts in finite time without the need to take in account gas-consumption. Nevertheless, we still need to restrict the Vyper language to accommodate for nuXmv's limited syntax. Therefore, we define a new sub-language, named MiniVyper, which, in comparison with Vyper, lacks internal and external calls (except the `send` function), creating new contracts, byte arrays, lists and mappings, structs, custom units and event logging. Despite being considerably limited, MiniVyper is still complex enough to write interesting smart contracts and may be extended in the future.

3.1 Concrete syntax

In Vyper, as in Solidity, a contract's structure resembles the one of a class of an object-oriented language. First, the user defines all of the contract's state variables, whose values are kept in the contract's storage, and then all of its methods.

```
1  owner: address
2  max_pay: constant(wei_value) = as_wei_value(10,"ether")
3
4  @public
5  @payable
6  def __init__():
7      self.owner = msg.sender
8
9  @public
10 def pay(_to: address, amount: wei_value) -> bool:
11     _from: address = msg.sender
12     assert _from == self.owner
13     assert amount <= self.balance
14     assert amout <= max_pay
15     send(_to, amount)
16     return True
17
18 @public
19 @payable
20 def __default__():
21     pass
22
```

Figure 1: A simple wallet smart contract.

Figure 1 shows an example of a Vyper contract complying with the MiniVyper syntax. This smart contract implements a simple wallet with a constant cap on payments. In this example, there is one state variable declared (`owner`, the wallet's owner), and a constant (`max_payment`, the maximum amount of Ether that can be transferred to another account in a single transaction). There are also three methods defined: `__initial__`, which always represents a contract's constructor; `__default__`, which always represents the fallback function, executed whenever the call received had no or incorrect input data and `pay`, which allows the owner of the wallet to transfer Ether to other accounts.

Since Vyper, unlike Python, is statically typed, a state variable's declaration always includes its type, besides an optional visibility indicator (`public()`). If present, the variable's value can be read by every account. Otherwise, the variable is private and can only be read internally. On the other hand, if a `constant()` indicator is used, than this variable is a constant, functioning as a macro for the given expression.

There are two kinds of variables types in the concrete syntax. The first are the *basic types*, namely:

- `bool`, with domain $\{True, False\}$;
- `int128`, 128-bit signed integer, with domain $\mathbb{S}_{128} = \mathbb{Z} \cap [-2^{127}, 2^{127} - 1]$;
- `uint256`, 256-bits unsigned integers, with domain $\mathbb{N}_{256} = \mathbb{N} \cap [0, 2^{256} - 1]$;
- `address`, 160-bit sequences, with domain $\mathbb{A}$;
- `bytes32`, 32-bytes sequences, with domain $\mathbb{B}^{32}$, where $\mathbb{B}$ is the set of bytes.

The *unit types* are based on the `uint256` base type and represent either an amount of Wei (`wei_value`), a point in time (`timestamp`) or an interval of time (`timedelta`).

Every method declaration must be preceded by a visibility decorator, either `@public` or `@private`. The latter asserts that only internal calls to the method are allowed, while the former indicates that the method is externally callable. Since MiniVyper does not support internal calls, only the `@public` decorator is permitted. Furthermore, one may also add the `@payable` decorator, which allows passing Ether along with messages to the method. Note that any call to a method without this decorator and with a positive message value fails immediately.

The declaration of methods in Vyper is practically identical to Python's, except that the argument's types and the output's type must be declared. Also, in MiniVyper, we do not allow default arguments, for simplicity sake.

Local variables are defined inside methods and their values are stored in EVM's volatile memory, being erased after each execution. In MiniVyper, we impose that all local variables must be declared before the first statement in a method. This ensures that their scope is the whole method, avoiding problems in the translation.

3.2 Abstract syntax of expressions

In this section we define the abstract syntax of MiniVyper expressions, including the language's typing rules.

We start by defining the sets $\mathbb{T}$, the set of MiniVyper's base types, and $\mathbb{U}$, the set of

MiniVyper's units:

$$\mathbb{T} = \{\texttt{bool}, \texttt{int128}, \texttt{uint256}, \texttt{address}, \texttt{bytes32}\}$$

$$\mathbb{U} = \{\varepsilon, \texttt{wei}, \texttt{s}, \texttt{s\_pos}\}$$

In the following, we will also refer to $\mathbb{I}$, the set of MiniVyper's identifiers. Now, we define an *atom*, the simplest expression that can occur in a MiniVyper program.

Definition 5 (Atom)**.** An *atom* is either a *literal*, or a *variable*.

- A *literal* is a tuple $\lambda = (t, u, val)$, such that $t \in \mathbb{T}$ is the literal's base type, $u \in \mathbb{U}$ is the literal's unit and *val* is the literal's value and its domain depends on the literal's base type.
- A *variable* is a tuple $\nu = (id, t, u, l)$, such that $id \in \mathbb{I}$ is the variable's identifier, t and u are as before and $l \in \{\texttt{storage}, \texttt{memory}, \texttt{input}, \texttt{environment}, \texttt{contract}\}$ is the location of the variable's value.

In both cases, if $u \neq \varepsilon$ then $t = \texttt{uint256}$. We denote a generic atom by α and the set of all atoms by $\mathbb{AT}$, the set of variables by $\mathbb{V}$ and the set of literals by $\mathbb{L}$. The dot notation $\alpha.c$ will be used to denote the component c of α. For example, for any atom α, $\alpha.t$ denotes its basic type. This notation will also be extended to other structures defined later on.

We distinguish between two kinds of atoms: literals and variables. Literals are simple representations of fixed values in the source code. Variables represent references to values that are not known at compile-time and are differentiated from each other by their identifier.

A smart contract can have six different kinds of variables. The *contract variables* have $l = \texttt{contract}$ and are the contract's address, with $id = \texttt{self}$, and its balance, with $id = \texttt{self.balance}$. The *environmental variables* reference message and block parameters. For example, among the most used are the `msg.sender` (address of message sender) and the `msg.value` (amount of Wei passed with the call). Both contract and environmental variables are common to all contracts. *State variables* have already been defined and must satisfy $t = \texttt{storage}$. Note that, similarly to attributes in a Python class, state variables are called using the `self` keyword. *Input variables* represent values given by the data field in the transaction and satisfy $l = \texttt{input}$. *Local variables* have also been described and satisfy $l = \texttt{memory}$. Finally, *for-loop variables* are the ones used as indexes in for-loop declarations and are only defined inside their bodies. In MiniVyper, we only allow for-loops of the kind '`for` *id* `in range(`n_1`,`n_2`)`', where $n_1, n_2 \in \mathbb{S}_{128}$ and $n_1 < n_2$. These variables always have type `int128` and are also stored in memory ($l = \texttt{memory}$).

An atom's unit is defined, i.e., is different than ε, whenever the atom represents a certain type of quantity. In MiniVyper, these can only be an amount of Ether in Wei (`wei`), a position in time in seconds (`s_pos`) or a number of seconds (`s`). For example, the `msg.value` and the `self.balance` variables have unit `wei`, while `block.timestamp` has unit `s_pos`. Note that variables in the abstract syntax with units `wei`, `s_pos` and `s` correspond, respectively, to variables declared with the `wei_value`, `timestamp` and `timedelta` types in the concrete syntax.

Now that we have introduced the atomic parts of expressions, we can discuss how to form expressions in general. Since the grammar we defined for the abstract syntax of MiniVyper expressions is too large to be presented here, we simply give a brief description of the available operators and functions. In general, we denote an expression by E and write E_x, for $x \in \mathbb{T}$ or $x \in \mathbb{U}$, to denote an expression with type/unit x.

In general, binary operators and functions in Vyper must be used in expressions with the same base type and unit. MiniVyper supports the operators `==` and `!=` for all expressions; the comparison operators `<`, `>`, `<=` and `>=`, the arithmetic operators `+`, binary `-` and `*`, and the `max` and `min` functions for `int128` and `uint256` expressions; the unary minus `-` just for the former; `/`, `%` and the bitwise operations just for the latter and `not`, `and` and `or` for `bool` expressions. Conversions of types and units are also allowed using the `convert`, `as_unitless_number` and `as_wei_value` functions.

3.3 Operational semantics

In order to develop MiniVyper's operational semantics, we will define proper models for smart contracts, smart contracts' states, machine states and execution environments. These concepts are loosely based on the ones in the execution models of [7], although many of the components are different. One key difference is the absence of gas in our model. Note that the amount of gas spent in an execution is calculated at the opcode level, which makes it impossible to include in our model. Furthermore, in a local perspective, the only case where gas influences a smart contract's execution is if it runs out, which causes it to halt and the smart contract to revert to its previous state. Hence, it would be as if the execution never happened.

We begin the description of MiniVyper's semantics by defining a model for a smart contract and one of its states. Henceforth, let $\mathbb{V}_x = \{\nu \in \mathbb{V} : \nu.l = x\}$. Note that the sets $\mathbb{V}_{environment}$ and $\mathbb{V}_{contract}$ are fixed and equal for all contracts. Fur-

thermore, let $dom(\mathbb{AT})$ be the domain of all atoms, that is, the set $\{True, False\} \cup \mathbb{S}_{128} \cup \mathbb{N}_{256} \cup \mathbb{A} \cup \mathbb{B}^{32}$.

Definition 6. A MiniVyper *smart contract* is a tuple $C = (addr, V_{storage}, code)$ where $addr \in \mathbb{A}$ is the contract's address, $V_{storage} \subset \mathbb{V}_{storage}$ is the contract's finite set of storage variables and $code \subset \mathcal{M}$ is the finite set of the contract's methods.

A *state* of C is either $\perp$, if the contract is destroyed, or a pair $\sigma_C = (bal, stor)$, such that $bal \in \mathbb{N}_{256}$ is the contract's current balance and $stor : V_{storage} \to dom(\mathbb{AT})$ is the contract's storage. We denote the set of all of C's states by $\mathcal{CS}_C$ and the set of all MiniVyper contracts by $\mathcal{C}$.

A smart contract C is immutable, while its state may suffer changes caused by transactions. The code of a contract is defined in terms of a set $\mathcal{M}$. We call this the *set of MiniVyper methods*, and its elements are tuples $M = (id, payable, V_{input}, V_{local}, V_{for}, rt, size, body, indent, next)$, where id is the method's identifier; $payable$ is a boolean that indicates if the method can receive funds; $V_{input} \subset \mathbb{V}_{input}$, $V_{local} \subset \mathbb{V}_{memory}$ and $V_{for} \subset \mathbb{V}_{memory}$ are the method's finite sets of input, local and for-loop variables, respectively; rt is a pair $(t, u) \in \mathbb{T} \times \mathbb{U}$ indicating the type and unit of the output, if any, or ε otherwise; $size \in \mathbb{N}^+$ is the number of lines in the method, such that each line contains a statement; $body : \{1, ..., size\} \to \mathcal{S}_M$ is a mapping assigning to each line its corresponding statement, $indent : \{1, ..., size\} \to \mathbb{N}$ is a mapping assigning to each line its corresponding indentation level and finally, $next : \{1, ..., size\} \to \{-2, ..., size\}$ is a mapping assigning to each line its jump destination.

Our idea is to represent the execution of a method the same way a person looking at its source code imagines it takes place. A program counter variable is used to keep track of the line of source code currently being executed, which, after the line's statement and expressions are evaluated, jumps to some destination determined by such evaluation. In general, it is fairly easy for a person to determine such jump destinations just by looking at the code. Nevertheless, a way to automatically find the jump destination of each line is necessary. The *next* mapping serves that purpose and can be built from the mappings *body* and *indent*.

Let C be a MiniVyper smart contract and M one of its methods. $\mathcal{E}_M$ is the set of finite expressions producible from literals and variables in M. The set of M's possible statements $\mathcal{S}_M$ is formed by:

- $\nu := E$ and $clear(\nu)$, such that $\nu \in C.V_{storage} \cup M.V_{local}$ and the types/units of ν and E are the same;
- $return(E)$ and $selfdestruct(E_{address})$, such that the type/unit of E matches $M.rt$.;
- $pass$, $break$ and $continue$;
- $send(E_{address}, E_{wei})$;
- $assert(E_{bool})$ and $raise$;
- $if(E_{bool})$, $elif(E_{bool})$ and $else$;
- $for(\nu, \lambda_{lower}, \lambda_{upper})$, such that $\nu \in M.V_{for}$, $\lambda_{lower}, \lambda_{upper} \in \mathbb{L}$, $\lambda_{lower}.t = \lambda_{upper}.t =$ `int128` and $\lambda_{lower}.val < \lambda_{upper}.val$.

where all the expressions belong to $\mathcal{E}_M$. Most of these statements are common to Python, but there are also some different ones: $clear(\nu)$ is simply an assignment to the variable's default value, $send(E_{address}, E_{wei})$ attempts to send the amount of Wei E_{wei} to the address $E_{address}$ and $selfdestruct(E_{address})$ permanently destroys the contract, sending its whole balance do the address specified.

The next step is to define the *machine state* and the *execution environment*.

Definition 7. A *machine state* is a tuple $\mu = (pc, m, out)$, such that $pc \in \mathbb{N} \cup \{-2, -1\}$ is the current program counter; $m : \mathbb{V}_{memory} \to dom(\mathbb{AT}) \cup \{\varepsilon\}$ is the local memory and $output \in dom(\mathbb{AT}) \cup \{\varepsilon\}$ is the output value.

An *execution environment* is a tuple $\iota = (me, input, env, initial)$, where $me \in \mathbb{I}$ is the method selector, the identifier of the method being executed; $input : \mathbb{V}_{input} \to dom(\mathbb{AT}) \cup \{\varepsilon\}$ is the given input; $env : \mathbb{V}_{env} \to dom(\mathbb{AT})$ is the function which assigns a value to each environmental variable and $initial = (bal_{init}, stor_{init}) \in \mathbb{N}_{256} \times (\mathbb{V}_{storage} \to dom(\mathbb{AT}))$ is the contract's initial state. We denote the set of machine states by $\mathcal{MS}$ and the set of execution environments by $\mathcal{I}$.

In our model, the execution of a MiniVyper smart contract is represented by a Kripke structure such that its states are 3-tuples formed by the previously defined smart contract's states, machine states and execution environments. Its transitions are defined according to evaluation and transition rules, based on [6] and [7], which dictate how the execution state evolves according to the statement currently selected by the program counter and the evaluation of the statement's expressions.

Definition 8. Let $C \in \mathcal{C}$ be a MiniVyper smart contract. We define the *Kripke structure for* C as a tuple $K_C = (S, S_0, \longrightarrow, L)$, where

- $S \subset \mathcal{CS}_C \times \mathcal{MS} \times \mathcal{I}$, such that, for all $s = (\sigma_C, \mu, \iota) \in S$:
  - There is a $M \in C.code$ such that $\iota.me = M.id$;
  - If $\iota.env(\nu_{msg.value}) > 0$ then $M.payable = True$;
  - If $M.rt \neq \varepsilon$ then either $\mu.output = \varepsilon$ or $\mu.output$ is a value in the domain determined by $M.rt$.

- $S_0 = \{(\sigma_C, \mu_0, \iota) \in S : \mu_0 = (1, m_0, \varepsilon)\}$, where $\sigma_C.bal \geq msg.value$, $\iota.initial = \sigma_C$ and m_0 maps all local variables to their default values and all for-loop variables to the lower bound of the for statement defining them.
- $\longrightarrow$ is a transition relation defined by a series of inductive rules.
- For every $s \in S$, $L(s)$ is the set of conditions on the contract's variables that hold in s.

Before defining the transition relation $\longrightarrow$, we need to define an evaluation relation for expressions, inspired by [6]:

Definition 9 (Evaluation relation)**.** Let C be a MiniVyper smart contract, K_C its corresponding Kripke structure, $s = (\sigma_C, \mu, \iota)$ a state of K_C and M a method such that $\iota.me = M.id$. The *evaluation relation* $\Downarrow_s \subset \mathcal{E}_M \times (\mathbb{L} \times \{True, False\})$ associates an expression E in M to a pair $\langle \lambda, cond \rangle$, such that λ has the same type and unit as E and holds the value of E in the state s, while $cond$ is $False$ if and only if E is impossible to evaluate in state s due to some error, such as an overflow, and the execution should halt and revert.

The binary relation $\Downarrow_s$ is inductively defined by a set of evaluation axiom schemes and evaluation rules. The axioms determine how atoms are evaluated, while the rules specify how to evaluate expressions using the already defined evaluations of its subexpressions. We assume that an expression E can always be evaluated all the way through, even if some of its subexpressions triggered an error. In these cases, the error propagates to E, which means that, although there is a $\lambda \in \mathbb{L}$ such that $E \Downarrow_s \langle \lambda, False \rangle$, it does not represent the value of E. This behaviour is unique to this model and does not reflect the real behaviour of a Vyper smart contract's execution, since an evaluation error immediately halts it. However, in both cases, the smart contract ends up reverting to the state prior to the execution. Therefore, we consider these two behaviours equivalent.

In Figure 3.3 we present, as an example, the axiom for environmental variables and the rule for the sum of two `uint256` expressions. Note the overflow checking condition $cond_{of}$ in the latter.

The transition relation $\longrightarrow$ for a structure K_C is also defined by a set of inference rules, whose structure is inspired by the small-step rules defined in [7]. We call these *transition rules*. Normally each rule depends on the statement being executed, given by $M.body(\mu.pc)$, and the next value of the program counter is given by the $M.next$ map, where M is the executing method. One exception is when s is a terminating state, in which case we always set a self-loop in s. In our model, the terminating states

$$\frac{\nu.l = \texttt{environment} \qquad \lambda = (\nu.t, \nu.u, \iota.env(\nu))}{\nu \Downarrow_s \langle \lambda, True \rangle}$$

$$\frac{\begin{array}{c} E_{uint256} \Downarrow_s \langle \lambda, cond \rangle \qquad E'_{uint256} \Downarrow_s \langle \lambda', cond' \rangle \\ \lambda = (\texttt{uint256}, \varepsilon, n) \qquad \lambda' = (\texttt{uint256}, \varepsilon, n') \\ cond_{of} = (n +_{256} n' \geq n) \\ \lambda'' = (\texttt{uint256}, \varepsilon, n +_{256} n') \end{array}}{E_{uint256} + E'_{uint256} \Downarrow_s \langle \lambda'', cond \wedge cond' \wedge cond_{of} \rangle}$$

Figure 2: Axiom scheme for environmental variables and evaluation rule for the addition of `uint256` expressions. $+_{256}$ is the addition modulus 2^{256}.

$$\frac{\begin{array}{c} s = (\sigma_C, \mu, \iota) \\ \iota.me = M.id \qquad M.body(\mu.pc) = \nu := E \\ \nu \in C.V_{storage} \qquad E \Downarrow_s \langle \lambda, True \rangle \\ \sigma'_C = \sigma_C[stor \rightarrow stor[\nu \mapsto \lambda.val]] \\ \mu' = \mu[pc \rightarrow M.next(\mu.pc)] \qquad s' = (\sigma'_C, \mu', \iota) \end{array}}{s \longrightarrow s'}$$

Figure 3: Transition rules for state variable assignments.

are marked by the pc taking the value 0 (successful), -1 (exception) and -2 (contract destroyed). For example, we present in Figure 3 the transition rule for assignments to state variables.

Since the EVM language is deterministic, it is expected that, for all $s \in S$, there is only one transition rule and one state s' such that s and s' appear in that rule's premises. Hence, s' is the only successor of s. However, when a *send* statement is executed, its success depends on parameters exterior to the contract, such as the target's state. When the whole blockchain is represented, these parameters are known and so the success of the execution is determinable. Because our model follows a local approach, we simply set that a *send* statement may fail non-deterministically. Nevertheless, we may still state the following result:

Theorem 3. *Let $C \in \mathcal{C}$ be a MiniVyper smart contract such that none of its methods contains a send statement. Then, K_C, the Kripke structure for C, is quasi-deterministic and, by Theorem 2, CTL and LTL are equivalent in K_C.*

4. Model checking smart contracts

In this section we explain how our tool performs the translation from MiniVyper contracts to the NUXMV language and show a simple example where a denial of service attack is detected.

4.1 Translating MiniVyper to nuXmv

The input of this tool is a compilable[2] Vyper smart contract complying with the MiniVyper syntax, while its output are two nuXmv files, which model the contract's execution in different ways. The model of the first file (*simple model*) represents a single call to a method of the smart contract, while the model of the second file (*extended model*) represents an infinite series of calls to the contract's methods starting with the constructor, if defined. The simple model is based on the Kripke structures K_C defined earlier, while the extended model is based on an adaptation of K_C to handle several calls. The following explanation concerns the simple mode, the extended case being analogous.

The semantics of nuXmv is based on transition systems, where each state is an assignment to variables. To represent K_C in the nuXmv input language, we have to include every component of the σ_C, μ and ι tuples as a variable and properly define their changes in value according to the K_C's transition rules.

Here, the only kinds of nuXmv variables we consider are *state variables* and *frozen variables*. The main difference between them is that a frozen variable's value cannot change from state to state. Hence, the contract's address and all environmental and input variables are declared as frozen variables, while state, local, for-loop variables, as well as the balance, are declared as nuXmv state variables. We also declare a copy of the balance and each state variable as frozen variables, to hold their initial value ($\iota.initial$). Finally, the program counter ($\mu.pc$) is declared as a bounded integer variable (`pc`), the method selector ($\iota.me$) as a symbolic variable (`method`), whose domain is the set of method's identifiers, and an `output` variable is created for each appropriate method. The translation of types between MiniVyper and nuXmv is performed according to Table 1. Besides the foregoing variables, we declare an extra boolean variable, `send_success`, whose value varies nondeterministically and dictates if a send statement is successful.

MiniVyper	nuXmv
`bool`	`boolean`
`uint256`	`unsigned word[256]`
`int128`	`signed word[128]`
`address`	`unsigned word[160]`
`bytes32`	`unsigned word[256]`

Table 1: Type translation from MiniVyper to nuXmv.

In each transition, variables in nuXmv can take any value in their domain, unless constraints are defined. In this work, we mostly use *assignments*, which allow one to specify the initial and next values of an individual variable.

For the initial values, the program counter variable (`pc`) is set to 1, each local variable to its default value and each for-loop variable to the loop's lower bound. The values for rest of the variables is chosen from their domain non-deterministically.

The next value of `pc` depends on its current value. If it is -2, -1 or 0, it remains the same, as the execution has ended. Otherwise, the next value depends on the current values of `method` and `pc`, which indicate a MiniVyper statement in the source code, and is determined by the only applicable transition rule. Note that, if the statement contains an expression, all the overflow and similar errors checks concerning it must be true for the next `pc` value to be different than -1, which, as mentioned earlier, marks an execution error. Additionally, if the statement is a `send`, then `send_success` must also be true.

The next values of state and local variables is normally the same as the current one. This only changes when `method` and `pc` indicate an assignment to such variable, in which case the next value is its right-hand side expression. Likewise, for-loop variables, `output` variables and the balance change accordingly to `for`, `return` and successful `send` statements, respectively. Furthermore, if `pc=-1`, the balance and state variables are assigned their initial values, as the contract is supposed to revert to its pre-execution state.

4.2 Testing the tool

Consider the King of the Ether Throne smart contract, depicted in Figure 4, which is a simplification of a real smart contract[3]. This contract implements a game where the objective is to acquire the title of king. To do so, a player must overthrow the current king by sending the contract an amount of Ether greater than the current prize, through the `overthrow()` method.

Although seemingly secure, this smart contract is vulnerable to a denial of service attack. This can be detected by trying to prove this fairness property in the extended model: *If there are infinite calls to* `overthrow()` *with a message value greater than the prize and a caller different than the current king, then, eventually, the king will change during the execution of this method.* In Figure 5 we show the LTL specification expressing that property and a representation of the counter-example trace found, where the execution of the `overthrow` method constantly fails because `send_success` is set to `FALSE`.

[2] At the time of writing, the compiler version was v0.1.0-beta.13.

[3] Website dedicated to this contract: https://www.kingoftheether.com/

```
king : public(address)
prize : public(wei_value)
owner : public(address)

@public
@payable
def __init__():
    self.owner = msg.sender
    self.king = msg.sender
    self.prize = msg.value

@public
@payable
def overthrow():
    assert msg.sender != self.king
    assert msg.value > self.prize
    send(self.king, msg.value)
    self.king = msg.sender
    self.prize = msg.value

```

Figure 4: A simple version of the King of the Ether Throne contract.

This situation can occur, for example, if the target contract's default function (or fallback function, in Solidity) automatically reverts every call received. In reality, what happened was that the amount of gas passed with the send function (2300) was not enough to fully execute the fallback function of the king at the time, leaving this smart contract permanently stuck.

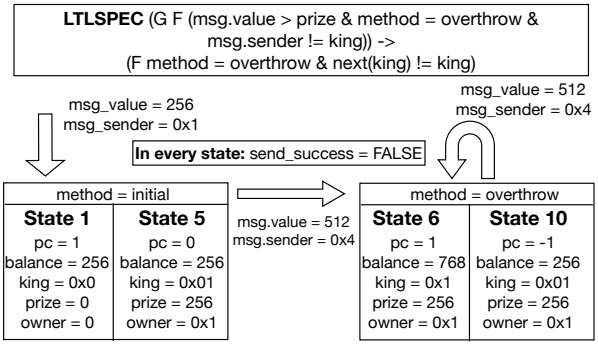

Figure 5: Counter-example trace for the given LTL specification in the King of the Ether Throne contract.

Overall, when testing our tool in various smart contracts, we found that the simple model is very useful to test invariant properties. In fact, even with more complex contracts, the invariant checking algorithms in nuXmv were powerful enough to prove most specifications, specially if we introduce some harmless restrictions to the state space

On the other hand, the extended model did not meet our expectations, mainly because it is too detailed to consider more than a few calls. In a method's execution, each different value of the program counter is a different state, and thus a single call can represent dozens of states, making it is extremely hard to find counter-example traces. Nevertheless, it may still be of use in very simple contracts as the King of the Ether Throne example.

5. Conclusions

In this work we studied the formal verification of Ethereum smart-contracts using model checking. First, we studied the temporal logics LTL and CTL and defined a class of Kripke structures where their expressiveness is the same. Next, we described Vyper, the high-level language for Ethereum smart contracts, and defined its sublanguage MiniVyper, providing a formal definition of its operational semantics based on Kripke structures. Finally, we described how to translate a MiniVyper contract to the nuXmv language, using models that represent a contract's execution at different scales, and illustrated a simple example where a denial of service attack was detected.

Possible future works include expanding the MiniVyper syntax, from the easier task of including arrays and mappings to finding a way to represent external calls, developing an appropriate abstraction of the extended model, in order to reduce its state space, and completing the formalisation of Vyper's operational semantics.

Acknowledgements

This work was partially supported by Programa Operacional Competitividade e Internacionalização, Programa Operacional Regional de Lisboa, Project Bloch, POCI-01-0247-FEDER-033823.

References

[1] C. Baier and J. Katoen. *Principles of model checking*. MIT Press, 2008.

[2] M. Bozzano, R. Cavada, A. Cimatti, M. Dorigatti, A. Griggio, A. Mariotti, A. Micheli, S. Mover, M. Roveri, and S. Tonetta. *nuXmv 1.1.1 User Manual*. Fondazione Bruno Kessler, Via Sommarive 18, 38055 Povo (Trento) – Italy, 2016.

[3] V. Buterin. A next-generation smart contract and decentralized application platform. `http://blockchainlab.com/pdf/Ethereum_white_paper-a_next_generation_smart_contract_and_decentralized_application_platform-vitalik-buterin.pdf`, 2014.

[4] Ethereum. Solidity documentation - release 0.5.12. `https://solidity.readthedocs.io/en/v0.5.12/`.

[5] Ethereum. Vyper documentation - release 0.1.0-beta.13. `https://vyper.readthedocs.io/en/v0.1.0-beta.13/`.

[6] M. Fernández. *Programming Languages and Operational Semantics - A Concise Overview*. Undergraduate Topics in Computer Science. Springer, 2014.

[7] I. Grishchenko, M. Maffei, and C. Schneidewind. A semantic framework for the security analysis of ethereum smart contracts. In *International Conference on Principles of Security and Trust*, pages 243–269, 2018.

[8] G. Wood. Ethereum: A secure decentralized generalized transaction ledger. `https://gavwood.com/paper.pdf`, 2014.