Programming Methodology Group
Department of Computer Science
ETH Zürich

Verification of Ethereum Smart Contracts Written in Vyper

Master's Thesis

Robin Sierra

supervised by

Marco Eilers, Prof. Dr. Peter Müller

29 September 2019

Abstract

Ethereum is a blockchain-based distributed computing platform with support for smart contracts, programs for specifying financial transactions without the need for a trusted third party. These contracts may handle large amounts of funds, therefore it is important that they do not contain bugs that allow attackers to steal money, especially since the contracts cannot be changed once they are deployed on the blockchain. In this Thesis, we present 2VYPER, a verifier for smart contracts written in Vyper. Given a specification, 2VYPER can prove that a contract adheres to it. We introduce new specification constructs to give guarantees about the execution of a contract even in the presence of reentrancy, its correct handling of funds, and the absence of security problems like missing access control and denial-of-service. We demonstrate the usability of the verifier by proving non-trivial properties of real-world contracts.

Acknowledgements

I would like to thank Marco Eilers for all the time and effort spent supervising me, and all the advice he gave me. Additionally, I want to thank Prof. Müller for giving me the opportunity to work on a project in his group.

Contents

Chapter 1

Introduction

When dealing with financial transactions, one often requires a trusted third party to ensure that all participants adhere to the rules of the transaction. If such a party does not exist, a possible alternative is to use a *smart contract* running on a blockchain. A smart contract is a program that defines the rules of a transaction. When running on a blockchain, users and other contracts can interact with it, e.g. by transferring money to it. The execution on the blockchain enforces that all interactions happen according to the rules defined in the contract. The most widely used platform for smart contract is Ethereum [1]. On Ethereum, smart contracts are usually written in a high-level language and compiled into bytecode which then executes in the Ethereum Virtual Machine (EVM). The most common such language is *Solidity* [2]. More recently, *Vyper* [3], a Python-like language that is designed to be a simpler alternative to Solidity, has been developed. While this removes some pitfalls present in Solidity, writing correct smart contracts remains challenging [4], which is especially problematic as smart contracts cannot be altered once deployed, not even by their creator. It is therefore of paramount importance that they do not contain bugs that allow malicious actors to steal or tamper with the funds.

In recent years, a large effort has been made to develop tools for finding bugs in smart contracts. While testing may offer some confidence about the correctness of a contract, it cannot give guarantees about all possible executions. An alternative is to look for *vulnerable patterns*, that could potentially be exploited. This can usually be done without manual work by users, but therefore cannot be used to check custom properties and behavior. Tools that check for vulnerable patterns include Mythril [5], Oyente [6], Securify [7], and Slither [8].

Checking custom properties requires manual work in terms of *specifications*, but it can give much stronger guarantees about the behavior of the contract. Additionally, some potentially vulnerable patterns are not a concern for every type of contract, specifying which properties should hold can reduce the amount of false positives. Multiple tools that enable users to check custom properties exists. For example, KEVM uses the K framework to formalize the semantics of the EVM and includes it a deductive verifier. Since it works on bytecode, it can be used for both Solidity and Vyper, however, it requires learning a DSL to specify properties. In contrast, SOLC-VERIFY, a Solidity verifier that encodes contracts to Boogie [9, 10], and VerX [11], also a verifier for Solidity, enable specifications in a Solidity-like syntax.

This thesis introduces 2VYPER, a deductive smart contract verifier for Vyper. The tool allows users to specify the desired behavior of a contract in a Vyper-like language, and then verifies that the code adheres to it. The verifier is built on the following principles:

- It allows users to check a contract against a specification. It reports where a contract violates the specification and why.
- It supports a rich, but intuitive specification language, that allows users to specify correct behavior and the absence of security vulnerabilities.
- Instead of checking predefined properties that may or may not be vulnerabilities for a specific contract, it allows users to specify and check any property they want their contract to satisfy.

2VYPER works similarly to SOLC-VERIFY for Solidity, but offers a more extended set of verification constructs designed to rule out various security properties. In addition, instead of encoding to Boogie, it translates Vyper code with specifications to *Viper* [12], an imperative intermediate verification language developed at ETH Zürich. Viper includes backends based on verification condition generation and symbolic execution, both of which can be used with 2VYPER, and which ultimately verify programs using an SMT solver.

The thesis is structured as follows: Chapter 2 provides the necessary background for this thesis, Chapter 3 gives an overview of security problems in Vyper and introduces the specification constructs that 2VYPER offers for verifying the behavior of the contract. The encoding into Viper is explained in Chapter 4, while Chapter 5 gives a broad overview about the implementation. Finally, the verifier is evaluated in Chapter 6, with a conclusion in Chapter 7.

Chapter 2

Background

2.1 Ethereum

Ethereum is a decentralized computing platform with support for smart contracts. It has its own cryptocurrency, *ether*, which smart contracts can use for financial transactions. Ethereum is based on a *blockchain* which records transactions in a distributed and verifiable way. To execute a transaction, a user has to submit it to the network where multiple transactions are then grouped into a block and executed by *miners*. To compensate for the execution costs a transaction fee is payed to each miner who successfully mined a block. The fee is based on how expensive it was to execute the transaction: Each operation in a transaction has a *gas cost* attached to it. When submitting a transaction to the network the user sets an upper limit for how much gas the transaction may use, and a gas price. Miners can then choose to include the transaction in the next block if they deem the gas price high enough.

Each user account and each contract is associated with an address. Each address can store certain state. While account addresses only store their ether balance, contract addresses additionally have memory to store arbitrary values. The state of each address is maintained on the blockchain and is altered through transactions. To make a payment, one sends ether from one's own address to some other address, thereby changing their balance. In addition, if there is a contract at some address, one can also invoke it by sending a call message to its address. The contract may then execute code and send a result. In particular, it may update its state and may itself call contracts (and potentially transfer ether); it might even call back into the calling contract. This is called *reentrancy* and can lead to subtle bugs (see Section 3.2). If a call fails (e.g., because of a division by 0) or it runs out of gas, the effects of the call are *reverted*, i.e., the state of all addresses is set back to how it used to be before the call occurred. Reverts can also be triggered intentionally by contracts, e.g., if an invalid input was given.

2.2 Vyper

Vyper is a statically-typed, Python-like smart contract language that targets the Ethereum Virtual Machine. It is designed to make it easy for programmers to write secure, maximally auditable code. In the following sections the most relevant features of Vyper that are needed for this thesis are introduced. All sections are based on the Vyper documentation [13] where a more complete overview of the language can be found.

2.2.1 Structure of a Contract

Each file in Vyper represents a contract. Contracts can have persistent storage on the blockchain and functions that can be called by users or other contracts. Listing 2.1 shows an example of a simple wallet contract that allows other contracts to deposit and withdraw their ether. The contract has two explicit storage variables, `owner`, which is an address, and `balance_of`, which is a map from addresses to amounts of wei[1] (an unsigned integer). Every contract also has an implicit storage variable `balance` that stores the amount of ether currently on the contract. Storage variables are like fields in object-oriented programming and can only be altered by the contract itself. There are also three functions: `__init__` is a special function that initializes

```
owner: address
balance_of: map(address, wei_value)

@public
def __init__():
    self.owner = msg.sender

@public
@payable
def deposit():
    self.balance_of[msg.sender] += msg.value

@public
def withdraw(amount: wei_value):
    self.balance_of[msg.sender] -= amount
    send(msg.sender, amount)
```

Listing 2.1: Simple wallet contract that stores the balance of each address.

---

[1] wei is the subunit of ether; 1 ether = 10^{18} wei

the contract when it is deployed. The function `deposit` stores some amount of wei in the contract. Inside the body of the function the code has access to the `msg` variable to get information about the call itself – `msg.sender` is the caller, and `msg.value` is the amount of wei that was sent with the call. The function increases the balance of the caller in `balance_of` by the amount sent. In order for a function to be called with a non-zero amount of wei it must be annotated with the `@payable` decorator. Calling a payable function implicitly increases `self.balance`. Finally, `withdraw` is used to withdraw ether from the contract. In the body, the balance of the sender is decreased by the specified amount and sent to the caller by using the built-in `send` function. Note that `wei_value` is an unsigned value, therefore passing an amount greater than the balance will cause an underflow which in Vyper results in the transaction being reverted to the state before the call was executed. The same could be achieved by using an assertion

```
assert amount <= self.balance_of[msg.sender]
```

at the beginning of the function. Generally, assertions are used to reject invalid inputs, as failing an assertion reverts the current call. Furthermore, if the receiver is a contract, `send` could fail and revert the current call as well.

2.2.2 Statements and Expressions

Statements and expressions in Vyper are mostly the same as in Python, with some advanced Python features like list comprehensions missing. Since Vyper is statically typed, all variables (both storage and local) have to be annotated with the appropriate types. An important feature of Vyper is that no recursion is allowed and all loops must have a statically known amount of iterations. This allows one to calculate an upper bound of the amount of gas a function can use, therefore providing users with an estimate of how expensive a call would be.

2.2.3 Sends and Calls

As stated before, contracts may send money to each other. This is done by using the built-in `send` function. Internally, this is translated to a call of a special function, the *fallback function*. In Vyper (and also Solidity), its default implementation simply reverts the transaction to prevent accidental sending of ether, but it can be defined in the contract to override this behavior (see Listing 2.2). The function will also be called if no other function matches the signature provided in the call, hence the name fallback function. The fact that sending ether may result in execution of code can sometimes lead to surprising bugs.

```
...

@public
@payable
def __default__():
    # Do something interesting
    pass

...
```

Listing 2.2: The fallback function.

Functions of other contracts can be invoked by first declaring the interface of the contract to be called and then casting an address to that type. In Listing 2.3 the wallet contract has been extended to include an observer that gets notified whenever someone deposits ether. The observer interface `WalletObserver` is declared with the `contract` keyword, and the function `on_deposit` is defined as *modifying*, i.e., it is allowed to modify the state of the blockchain. In the initializer, `self.owner` is then set to the sender cast to a `WalletObserver` (line 9). Note that there is no actual check that the

```
 1  contract WalletObserver:
 2      def on_deposit(amount: wei_value): modifying
 3
 4  owner: WalletObserver
 5  balance_of: map(address, wei_value)
 6
 7  @public
 8  def __init__():
 9      self.owner = WalletObserver(msg.sender)
10
11  @public
12  @payable
13  def deposit():
14      self.balance_of[msg.sender] += msg.value
15      self.owner.on_deposit(msg.value)
16
17  @public
18  def withdraw(amount: wei_value):
19      self.balance_of[msg.sender] -= amount
20      send(msg.sender, amount)
```

Listing 2.3: Wallet contract with an observer.

contract at that address adheres to this interface. On line 15 in the `deposit` function the contract is then called after increasing the balance.

In general, calling another contract leads to execution of unknown code that could change arbitrary state.

2.3 Structs

Vyper does not have classes or objects, but, similar to C, includes *structs*, in order to group multiple variables. Listing 2.4 shows an example contract that uses a `Person` struct to record the vote of a user. Structs are declared like Python classes, but with the `struct` keyword. They may contain an arbitrary amount of variables with types. Structs can subsequently be used in maps and functions. In line 11 the syntax to initialize a struct is shown. To create a new struct one calls the initializer with the same name as the struct, and passes a dictionary from variables to their initial values.

```
 1  struct Person:
 2      id: int128
 3      vote: int128
 4
 5  people: map(address, Person)
 6
 7  ...
 8
 9  @public
10  def vote(i: int128, v: int128):
11      self.people[msg.sender] = Person({id: i, vote: v})
```

Listing 2.4: Struct syntax.

2.4 Events

Applications that are based on smart contracts often need to react to actions happening in the contract, e.g., a crypto wallet application needs to update its UI every time the user receives a payment. To do that, a contract can fire an *event*. On the blockchain-level, firing an events simply means writing to a log that is stored on-chain. Applications can then use these logs as triggers for UI updates. In Vyper, the syntax for firing an event is shown in Listing 2.5. In line 1, an event type `Deposit` is declared. Events may take parameters, in this case the address which deposited the ether and the amount. Later on, in line 9, the event is fired after ether has been deposited. Firing an event syntactically looks like calling a method on the `log` object.

```
1 Deposit: event({_by: address, _amount: wei_value})
2
3 ...
4
5 @public
6 @payable
7 def deposit():
8     self.balance_of[msg.sender] += msg.value
9     log.Deposit(msg.sender, msg.value)
```

Listing 2.5: Event syntax.

Chapter 3

Security Problems and Specifications

This chapter gives an overview of various security vulnerabilities present in Ethereum smart contracts. It also describes the specifications in 2VYPER that can be used to prove the absence of such vulnerabilities. The list of vulnerabilities is partially based on [4].

3.1 Functional Correctness

Probably the simplest and most straightforward security problem with smart contracts (and any other program) is incorrect behavior, i.e., the code is not doing what it is supposed to do. Verifiers generally provide annotations to specify the expected behavior of the code, and this one is no different. Usually, these specifications are given as *preconditions*, which are conditions under which functions are allowed to be called, and *postconditions*, conditions that the function ensures are true after a call. In the setting of smart contracts, however, functions can be called by anyone under any circumstance. It is, in general, not possible to enforce preconditions outside of the code that is being executed. Calling functions with unexpected values is one of the easiest ways to exploit a vulnerable contract. Therefore, the verifier does not allow writing preconditions. Instead, specifying the behavior of a function is done solely through postconditions. Specifications are written in special comments starting with `#@` followed by a keyword (e.g., `ensures` for postconditions) and a colon. They are normal Vyper expressions without side-effects, but may use various special functions. Listing 3.1 shows a slightly different version of Listing 2.1 in the previous chapter, where the `deposit` function has been annotated with a trivial postcondition `True` (line 6). To specify that the function increases the balance of the sender by the amount of wei sent, we need a suitable postcondition. It could look as follows:

```
contract Client:
    def transfer(): modifying

balance_of: map(address, wei_value)

#@ ensures: True
@public
@payable
def deposit():
    assert msg.value >= as_wei_value(1, "ether")
    self.balance_of[msg.sender] += msg.value

@public
def withdraw(amount: wei_value):
    self.balance_of[msg.sender] −= amount
    Client(msg.sender).transfer(value=amount)
```

Listing 3.1: Postcondition syntax.

```
#@ ensures: self.balance_of[msg.sender] ==
#@    old(self.balance_of[msg.sender]) + msg.value
```

The postconditions uses the `old` function, with which one can refer to the state of the contract before the execution of the function in order to express that the value of `self.balance_of[msg.sender]` has increased by `msg.value` compared to the state before. However, strictly speaking, this postcondition does not hold: If the transaction reverts, all the state will be the same as it was before the call. The verifier therefore requires users to explicitly state whether postconditions are required to hold only on success. Hence, a working postcondition[1] can be

```
#@ ensures: success() ==> self.balance_of[msg.sender] ==
#@    old(self.balance_of[msg.sender]) + msg.value
```

which indeed verifies. Requiring to explicitly write `success()` allows us the specify conditions in which `deposit` is supposed to fail, namely when being given illegal inputs:

```
#@ ensures: msg.value < as_wei_value(1, "ether") ==>
#@    not success()
```

---

[1] Currently, implications have to be written with the function `implies(a, b)`, however for ease of presentation we will use the implication operator `==>`.

states that the function always fails if `msg.value` is less than 1 ether and

```
#@ ensures: msg.value >= as_wei_value(1, "ether") ==>
#@    success(if_not=out_of_gas)
```

asserts that the function succeeds for legal inputs, given that it does not run out of gas. Note that using plain `success()` instead of the special expression `success(if_not=out_of_gas)` would not work as a transaction may fail at any point because it has used up all its available gas.

To verify `withdraw` we want to specify that it sends `amount` of wei to the caller (the optional `value` keyword argument is used to send ether with the call). We can use the function `sent(a)`, which denotes the total amount of wei sent from this contract to an address `a`. However, due to reentrancy (see Section 3.2), we cannot prove this precisely, as the caller might reenter our contract and call `deposit` and `withdraw` again, which increases `sent(a)`. We therefore prove that we send at least `amount` of wei:

```
#@ ensures: success() ==>
#@    sent(msg.sender) >= old(sent(msg.sender)) + amount
```

A corresponding function `received(a)`, that denotes all ether received through calls, also exists.

Postconditions that should hold for every function in the contract can more conveniently be written as

```
#@ always ensures: True
```

on the top-level. We call them *general postconditions*.

Listing 3.2 shows the entire example discussed in this section with the specifications for `deposit` and `withdraw`.

3.2 Reentrancy

Reentrancy might be the most famous security problem, not least because of the infamous DAO incident [14], where an attacker managed to steal ether worth 60M$, and which led to a subsequent hard fork of the Ethereum blockchain to undo the attack. Reentrancy is a consequence of the fact that calling another contract may execute arbitrary code. Listing 3.3 shows a version of Listing 3.2 (lines 22 and 23 have been swapped, and `withdraw` does not take an argument anymore) that is vulnerable to a reentrancy attack:

```
contract Client:
    def transfer(): modifying

balance_of: map(address, wei_value)

#@ ensures: success() ==> self.balance_of[msg.sender] ==
#@    old(self.balance_of[msg.sender]) + msg.value
#@ ensures: msg.value < as_wei_value(1, "ether") ==>
#@    not success()
#@ ensures: msg.value >= as_wei_value(1, "ether") ==>
#@    success(if_not=out_of_gas)
@public
@payable
def deposit():
    assert msg.value >= as_wei_value(1, "ether")
    self.balance_of[msg.sender] += msg.value

#@ ensures: success() ==>
#@    sent(msg.sender) >= old(sent(msg.sender)) + amount
@public
def withdraw(amount: wei_value):
    self.balance_of[msg.sender] -= amount
    Client(msg.sender).transfer(value=amount)
```

Listing 3.2: Complete example.

When calling the `transfer` function of the client contract, the balance has not yet been set to 0, therefore if `transfer` contains a call to `withdraw` (i.e., the call reenters the wallet contract), the balance will still be the same and if the total balance of the wallet contract is sufficiently high, the money is payed out multiple times. Listing 3.4 shows an attacker that exploits this vulnerability: It contains a counter `num` that indicates how many times the contract is supposed to reenter (doing it too many times will drain the wallet of all ether and cause a revert when trying to send more ether than is currently on the contract). The `transfer` function then checks if it should reenter, and if so, calls back into `withdraw` which in turn will call `transfer` again, never setting the balance to 0 until the very end.

3.2.1 Invariants

Proving a postcondition in the presence of reentrancy is hard, since external calls may execute arbitrary contract code, and therefore we do not know what the state of the contract will be after the call. However, since a contract's state can only be changed by the contract itself (except for the balance, as discussed later in this section), we only need to consider side effects pro-

```
contract Client:
    def transfer(): modifying

balance_of: map(address, wei_value)

@public
@payable
def deposit():
    self.balance_of[msg.sender] += msg.value

@public
def withdraw():
    amount: wei_value = self.balance_of[msg.sender]
    Client(msg.sender).transfer(value=amount)
    self.balance_of[msg.sender] = 0
```

Listing 3.3: A contract vulnerable to a reentrancy attack.

```
contract Wallet:
    def deposit(): modifying
    def withdraw(): modifying

num: int128

...

@public
@payable
def transfer():
    if num > 0:
        num -= 1
        Wallet(msg.sender).withdraw()
```

Listing 3.4: An attacker contract to exploit the reentrancy vulnerability.

duced by functions found in the contract. It is therefore possible to preserve information about the state given that all functions that could be reentered preserve it. For that we can use *invariants*, as in [15].

Invariants are written similarly to postconditions, but do not belong to a specific function and are therefore declared on the contract level similar to storage variables. They denote general properties of the contract that should be true in any state of the contract in which code other than the contract's code can be executed. In particular, this means that they should hold before a call to a public function (when the execution is taken over from another contract), and they need to be true when it returns or reverts (when control flow is given back), and during a call to an outside contract (while the execution is controlled by the callee). For example,

```
#@ invariant: sum(self.balance_of) <= self.balance
```

states that the sum of all entries of the balance map can be at most the contract's balance, i.e., it always has sufficient ether to pay back the clients. For a correct version of the wallet contract, this invariant will always hold, as any address can only withdraw whatever it has previously deposited. In the incorrect version, the invariant does not hold when control is given to the client contract (line 16) as shown in Listing 3.5, because while the external contract executes the ether has already been sent, but has not been subtracted from `self.balance_of[msg.sender]` yet. It could be that the balance is now smaller than the sum of entries in `self.balance_of`, i.e., the code violates the invariant. Hence, the verifier should report an error.

```
 1  contract Client:
 2      def transfer(): modifying
 3
 4  balance_of: map(address, wei_value)
 5
 6  #@ invariant: sum(self.balance_of) <= self.balance
 7
 8  @public
 9  @payable
10  def deposit():
11      self.balance_of[msg.sender] += msg.value
12
13  @public
14  def withdraw():
15      amount: wei_value = self.balance_of[msg.sender]
16      Client(msg.sender).transfer(value=amount)  # Error
17      self.balance_of[msg.sender] = 0
```

Listing 3.5: Verifying a vulnerable contract.

The invariant allows us the find the bug in the implementation, however, it does not give precise guarantees about the ether flow of this contract. Note that using an equality instead of the inequality (i.e., proving that the balance is equal to the sum of the entries of the balance map) will not work, because it is possible in Ethereum for contracts to receive ether with no contract function being called, either through coinbase transactions[2] or when a contract is removed from the blockchain by a `selfdestruct`. It cannot *lose* ether, though. To show an equality, we can use the `sent` and `received` functions introduced in Section 3.1 and add the invariant

```
#@ invariant: sum(self.balance_of) ==
#@   sum(received()) - sum(sent())
```

stating that the total amount of ether recorded in `balance_of` is equal to the total amount of ether received minus the total amount of ether sent out. Note that `received` only denotes ether received through normal calls, not coinbase transactions or selfdestructs. The equality is even true for each address separately, which we can show with the use of a quantifier:

```
#@ invariant: forall({a: address},
#@    self.balance_of[a] == received(a) - sent(a))
```

This invariant guarantees that no address can withdraw more ether than it previously sent, i.e., it cannot steal ether from other addresses.

3.2.2 History Invariants

Until now, all invariants shown were properties of single states. Often, however, one wants to reason about how a contract changes (or does not change) over time. In Listing 3.6 we extend the wallet contract to a (rudimentary) crowdfunding contract, where users may pledge ether. If the goal is reached, the owner gets the money, else they may cancel the process and donors can withdraw their donations. Some properties that one may want to check are:

- The owner does not change.
- The goal does not change.
- Once crowdfunding has ended, it cannot not start again.
- Before the crowdfunding has ended, the balance may only increase.

These properties are all *history invariants*. History invariants may refer to the previous state by using `old` expressions just like postconditions. Written as invariants, the properties above are:

```
#@ invariant: self.owner == old(self.owner)
#@ invariant: self.goal == old(self.goal)
#@ invariant: old(self.ended) ==> self.ended
#@ invariant: not self.ended ==>
#@    self.balance >= old(self.balance)
```

---

[2] Mining rewards, sent to the address specified by the miner when mining a block

```
owner: address
goal: wei_value
ended: bool
balance_of: map(address, wei_value)

@public
def __init__(_goal: wei_value):
    self.owner = msg.sender
    self.goal = _goal

@public
@payable
def pledge():
    self.balance_of[msg.sender] += msg.value

@public
def withdraw():
    assert self.ended
    amount: wei_value = self.balance_of[msg.sender]
    self.balance_of[msg.sender] = 0
    send(msg.sender, amount)

@public
def end():
    assert msg.sender == self.owner
    self.ended = True
    if self.balance >= self.goal:
        send(msg.sender, self.balance)
```

Listing 3.6: A crowdfunding contract.

Many interesting properties of smart contracts can be expressed as history invariants.

3.2.3 Public States

To get a deeper understanding of postconditions and invariants and how they differ, we look at what may happen to a contract on the blockchain. Figure 3.1 shows the lifetime of a contract and its possible interactions with other contracts. States of the contract while it is itself executing are denoted in blue, while states of other contracts are denoted in yellow. States that are visible to other contracts are represented by large circles. We call such states *public states*. Local states are represented with smaller circles. A contract may receive ether through coinbase transactions and selfdestructs without a function being called. Such states, where the balance changes without the contract executing, are represented in red. Every contract first executes its initializer. After that, a function may be called to initiate a transaction. The function can then call external functions, which in turn may reenter the current contract, or selfdestruct and increase the balance. Between regular transactions, coinbase transactions may occur when a new block is mined, which can increase the balance of a contract. If a function reverts, the state after the function call is simply the same as before the call.

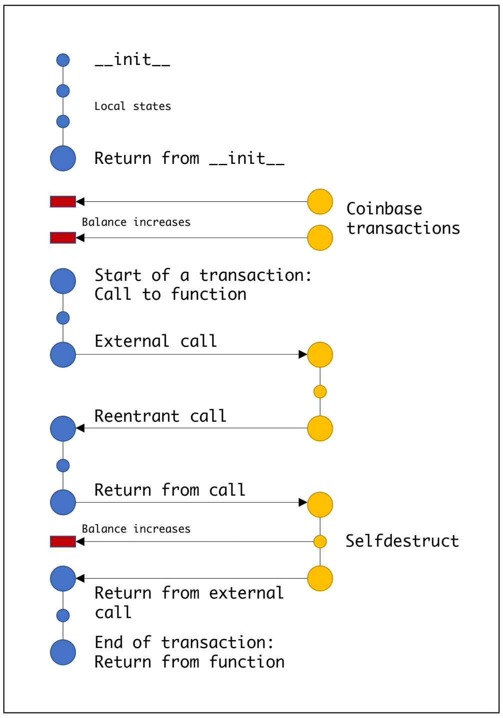

Figure 3.1: Lifetime of a contract.

Postconditions (both normal postconditions and general postconditions) specify properties that should hold after the execution of the function. Figure 3.2 shows the same states as Figure 3.1 with the blue arrows illustrating when postconditions need to hold: The end of the arrow is the state in which it needs to hold, while the start represents the `old` state. Postconditions need to hold both when a function returns and when it reverts, i.e., in the last public state of the call. To prove that a function satisfies a postcondition one therefore needs to check it at the end of the function. If the postconditions uses an `old` expression the state referred to is the state before calling the function, as shown in the figure. A special case is the initializer. Since the contract does not yet exist before the initializer is called one cannot refer to an `old` state. Postconditions of `__init__` may therefore not contain `old` expressions. General postconditions need to hold for the `old` being the same as the current state, as disallowing `old` expressions simply because of the initializer would be very limiting.

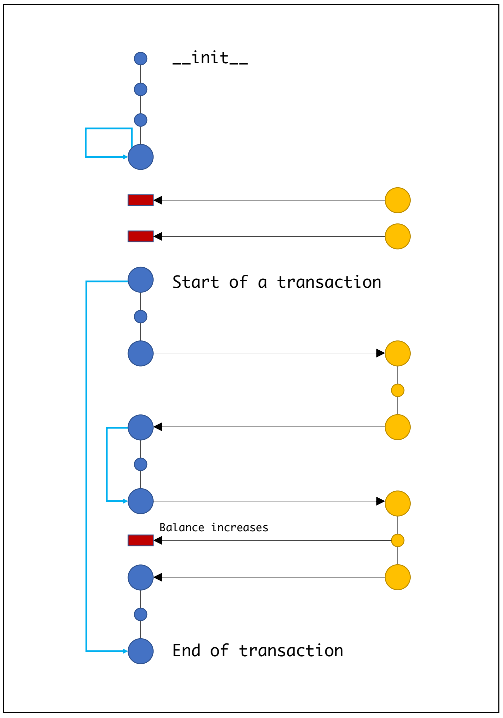

Figure 3.2: Postconditions need to hold at the end of functions, `old` refers to their start.

Contrary to postconditions, the goal of an invariant is to state properties that are generally true for a contract. In particular, they should give guarantees about the state of the contract (1) when a public function is called, and (2) after calling an external contract, even in the presence of reentrancy. To achieve that, invariants are required to hold in any public state[3]. In Figure 3.1, that is between the return of the initializer and the start of the transaction, during the external calls, and at the end of the transaction. Due to the fact that ether may be forced onto a contract without a function being called, this also implies that invariants where the balance is assumed to be constant (like `self.balance == 0`) are invalid. To prove that a function correctly establishes the invariants one needs to check that it holds in all public states that occur when control is given over to another contract, namely on function calls and after the function has been executed. In turn, one can assume that the invariant holds in the public state before the function call (as it has been established by the previous function call) and when returning

---

[3] Note that the invariant holds *after* sending potential ether (i.e., with the reduced balance), as this is the state that can actually be publicly observed.

from an external call.

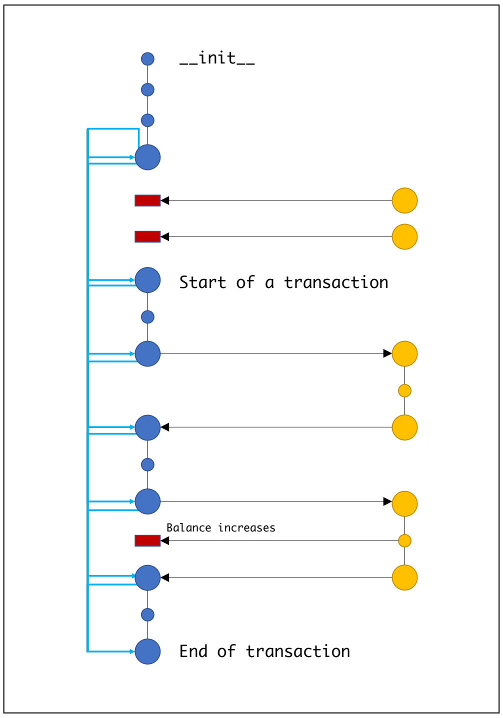

Figure 3.3: Invariants need to hold for each pair of public states, `old` is any previous state.

History invariants complicate the matter somewhat. For example, given the history invariant `self.val == old(self.val)`, when we call an external contract we want the invariant to state that the value of `self.val` is still the same as *before* the call, no matter how many times the call reenters (i.e., no matter how many public states are in-between). We therefore require invariants to be *reflexive*, i.e., they need to hold if the state does not change, and *transitive*, which means that if the invariants hold for each individual pair of consecutive public states, they also hold for each pair of two states, in particular the first (the state before the external call) and the last (the state after the external call) public state. Figure 3.3 shows the states in which invariants have to hold, again by using arrows to indicate the `old` state. Because of transitivity, there is an arrow from every state to all its successors.

A special case is again the initializer: Since there is not really a state before the initializer is called, it does not make sense to refer to the previous state with `old`. Instead, we check the history invariant analogous to the general postcondition just for the end state itself, i.e., both `self` and `old(self)` refer to the state after the initializer has finished executing. By the same reasoning, it does not make sense to require the initializer to satisfy the invariants if it reverts, since no public state is created in that case. Hence, invariants in the initializer are only checked on success.

3.2.4 Transitive Postconditions

Invariants denote general properties of contracts that are true in every public state. However, sometimes one simply wants a property to be true temporarily, while a call that may reenter is going on. An example for that is locking, as shown in Listing 3.7. The contract consists of a storage variable `val` with a setter, and a function that calls the contract `Foo`. A lock variable is used to ensure that the value of `self.val` is not changed by reentrant calls. At the moment we cannot prove the postcondition, since no invariants specify that the value of `self.val` cannot change while `self.lock` is true. Writing the invariant

```
 1  contract Foo:
 2      def foo(): modifying
 3
 4  val: int128
 5  lock: bool
 6
 7  @public
 8  def set_val(new_val: int128):
 9      assert not self.lock
10      self.val = new_val
11
12  #@ ensures: self.val == old(self.val)
13  @public
14  def call_foo():
15      assert not self.lock
16      self.lock = True
17      Foo(msg.sender).foo()
18      self.lock = False
```

Listing 3.7: A contract with a lock to prevent modification of storage.

```
#@ invariant: old(self.lock) ==>
#@   self.lock and self.val == old(self.val)
```

does not help though as the function `call_foo` violates it in line 18. The property is not generally true, but only temporarily until the lock is unlocked. The solution can be found in Figure 3.2: When we call a function, it may reenter, but if that happens, we know that any reentrant function call will be executed to its end. Hence, a general postcondition has to hold in that state, given that it does not refer to local state. Additionally, such a general postcondition needs to be reflexive in order to hold even if no reentrancy occurs, and transitive, so that it holds even after multiple reentrant calls. Since it is hard for the verifier to automatically infer transitivity, one has to tell the verifier that a general postcondition is transitive and can preserve information by annotating it as follows:

```
#@ preserves:
    #@ always ensures: old(self.lock) ==>
    #@   self.lock and self.val == old(self.val)
```

The verifier will check all such general postconditions for transitivity. Reflexivity is checked automatically by all functions since postconditions need to hold even if the function reverts, i.e., if the state does not change. The general postcondition can then be assumed whenever we return from an external function call.

3.3 Ether Lost in Transfer

In Ethereum, ether can be sent to any address. However, since many of these addresses are not used by anyone, ether sent to one of these addresses is simply lost and cannot be retrieved anymore. Since there is no way to check whether an address is used, there is no general way of verifying that a contract never does that. However, we can use invariants to prove it for certain types of contracts.

A simple case is specifying that a contract never sends ether to the 0x0 address, a special address that does not belong to anyone, which can be done with the following invariant:

```
#@ invariant: sent(ZERO_ADDRESS) == 0
```

A stronger property that some contracts satisfy is to require that ether is only sent to address from which one has, at some point, received ether. This means that the address has definitely been used at some point. The invariant to express this property is

```
#@ invariant: forall({a: address},
#@   sent(a) > 0 ==> received(a) > 0)
```

In Listing 3.8, a slightly shortened version of the wallet contract discussed in earlier sections, both properties hold, since `msg.sender` is never 0x0, and every address may only withdraw their own money. The second invariant is needed to show the third, as we need a way to link `sent` and `received` to `self.balance_of`.

```
balance_of: map(address, wei_value)

#@ invariant: sent(ZERO_ADDRESS) == 0
#@ invariant: forall({a: address},
#@   self.balance_of[a] == received(a) - sent(a))
#@ invariant: forall({a: address},
#@   sent(a) > 0 ==> received(a) > 0)

@public
@payable
def deposit():
    self.balance_of[msg.sender] += msg.value

@public
def withdraw():
    amount: wei_value = self.balance_of[msg.sender]
    self.balance_of[msg.sender] = 0
    send(msg.sender, amount)
```

Listing 3.8: A contract that does not lose ether in transfer.

3.4 Access Control

Often one wants to write contracts where not every participant is ’equal’. For example, in the crowdfunding contract in Listing 3.6, the owner of the contract is special as only they can end the crowdfunding, which then either allows participants to withdraw their funds if the goal has not been reached or pays out the funds to the owner. To ensure that only the owner may end the crowdfunding, an assertion is used. Forgetting proper access control, however, can be catastrophic. Even the simple crowdfunding contract leaves its entire funds open for anyone to steal if the assertion is missing. More subtle bugs can also occur in contracts where ownership can be transferred, e.g., mistakenly letting anyone change ownership to themselves.

To ensure proper access control the verifier supports *checks*. Checks are structurally similar to invariants, however, while invariants are general properties of every public state in the contract and are used to preserve information throughout call, checks only takes code into account that belongs to the contract to be verified. It may therefore the environment variables

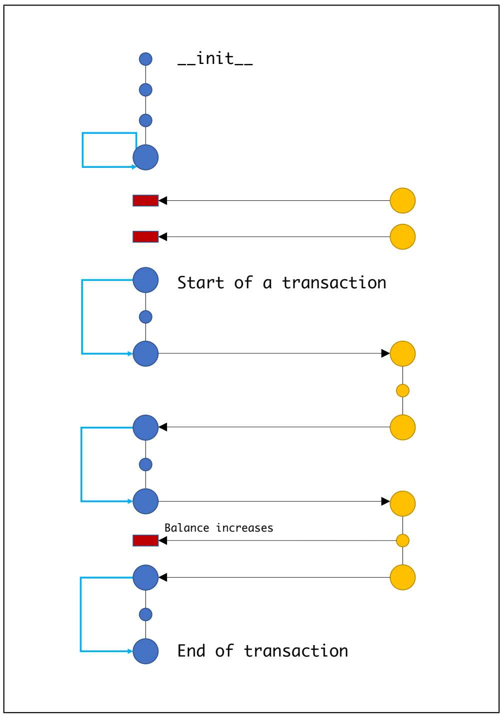

Figure 3.4: General checks need to hold for all consecutive public states, where the contract to be verified executes code.

`msg` and `block`. A check needs to hold whenever we give control to some other contract, either via an external call, or via a return, with `old` being the last state where some other contract had control. To express it in terms of public states, a check needs to hold for each pair of consecutive public states, given that our contract executes code in-between. An illustration can be found in Figure 3.4. Listing 3.9 shows an example of a check which states that only `self.owner` may change `self.val`. The function `good` correctly enforces this, while `bad` allows any caller to increment `self.val` (line 23). Note that a postcondition stating the same would not necessarily hold, as that would state that the value is not allowed to change if the current `msg.sender` is `self.owner`. With a check it is possible that an external call invokes `self.owner` which can in turn reenter this contract and change `self.val`. What is checked is that *inside the function body* only `self.owner` may change `self.val`.

Similar to postconditions, there is also a more convenient notation for checks that need to hold for all functions, i.e., *general checks*:

```
#@ always check: msg.sender != self.owner ==>
#@   self.val == old(self.val)
```

```
contract Foo:
   def foo(): modifying

val: int128
owner: address

...

#@ check: msg.sender != self.owner ==>
#@   self.val == old(self.val)
@public
def good():
   Foo(msg.sender).foo()
   if msg.sender == self.owner:
      self.val += 1
   Foo(msg.sender).foo()

#@ check: msg.sender != self.owner ==>
#@   self.val == old(self.val)
@public
def bad():
   Foo(msg.sender).foo()
   self.val += 1 # Error
   Foo(msg.sender).foo()
```

Listing 3.9: Using checks to ensure correct access control.

For access control this is almost always the case, as usually one is interested in the entire contract not allowing certain actions to everyone.

3.5 Functional Correctness Revisited

Incidentally, the checks introduced in the previous section are also useful to specify when events should be emitted. Similarly to access control, one is generally not interested in events emitted by reentrant calls, but one wants that the contract emits a specific event every time some condition is met. A typical example is to emit an event every time a payment is made, i.e., the balance decreases. Given the event declaration

```
Payment: event({amount: wei_value})
```

this can be specified as

```
#@ always check: self.balance < old(self.balance) ==>
```

```
#@   event(Payment(old(self.balance) - self.balance))
```

by using the `event` function and passing the event with arguments to it.

3.6 Denial-of-Service

Access control is used to prevent attackers from doing something bad, however, they could also prevent everyone else from doing anything good. Generally, if a function depends on another contract to execute, that contract could deny all progress if it always fails, i.e., cause a *denial-of-service*. This is why it is good practice to let users retrieve their ether (*pull*) instead of sending it to them directly (*push*). Consider Listing 3.10: Both `bad` and `good` are supposed let `self.owner` pay out a price to the winner, while also getting a share of the balance. However, `bad` contains two sends, one to `self.winner` and one to `self.owner`. Sending ether to `self.winner` executes their fallback function, which means that if it always fails, the call will be reverted and `self.owner` will also not get their ether. The second send could still fail, but this simply means that `self.owner` will lock their own

```
owner: address
winner: address
payouts: map(address, wei_value)

...

@public
def bad():
    assert msg.sender == self.owner
    send(self.winner, as_wei_value(1, "ether"))
    send(self.owner, as_wei_value(1, "ether"))

@public
def good():
    assert msg.sender == self.owner
    self.payouts[self.winner] += as_wei_value(1, "ether")
    send(self.owner, as_wei_value(1, "ether"))

@public
def withdraw():
    amount: wei_value = self.payouts[msg.sender]
    self.payouts[msg.sender] = 0
    send(msg.sender, amount)
```

Listing 3.10: Contract with both push and pull.

ether in the contract, which is their own fault. *Pushing* the ether to the winner contract is potentially dangerous. The function `good`, one the other hand, records how much ether `self.winner` receives in a map, so they may withdraw their ether in a separate transaction. Letting them *pull* their ether is safe.

The essential problem with a function like `bad` is that its successful execution depends not only on `msg.sender`, but some other contract, which may lead to ether being locked inside the contract. To prove that this does not happen, the verifier provides an additional version of `success`. Analogous to `success(if_not=out_of_gas)` introduced in Section 3.1 one can use `success(if_not=sender_failed)` to specify that the function will definitely succeed given that `msg.sender` does not cause the call to fail by having a fallback function that reverts or by not attaching sufficient gas. This means that the former implies the latter, as `msg.sender` may always start a transaction and attach sufficient gas to execute the function without causing an out-of-gas exception. Hence, the postcondition

```
#@ ensures: msg.sender == self.owner ==>
#@   success(if_not=sender_failed)
```

holds for `good`, but not `bad`, given that we can prove that there is always sufficient ether on the contract.

Since denial-of-service often causes ether to be locked in a contract, the verifier provides a special annotation for the case where one wishes to prove that it is possible to access some amount of ether. For Listing 3.10, one can write

```
#@ invariant: accessible(self.winner,
#@                       self.payouts[self.winner],
#@                       self.withdraw())
```

to express that `self.winner` can access at least `self.payouts[self.winner]` amount of ether by calling `withdraw` without any arguments. Using this with quantifiers is often desired, e.g.

```
#@ invariant: forall({a: address}, accessible(a,
#@    self.payouts[a], self.withdraw()))
```

can be used to specify that accessing the ether recorded in `self.payouts` is always possible, again by calling `withdraw`.

The verifier contains simple syntactic heuristics to determine which function can be used to access the ether. To use that, the contract must contain a function that contains a send or call with a value attached, and which has either no arguments or a single `wei_value` argument. The verifier will then assume that to be the appropriate function, with the potential argument being the amount of wei to withdraw. If multiple such functions exists, it will prefer ones called `withdraw` or similar. In cases where the heuristics determine the correct function, one only needs to give the first two arguments.

3.7 Unpredictable State

When a user submits a transaction to the network, it will not be executed immediately. Instead, miners may choose to include the transaction in the next block if the gas price is high enough, but other transactions may run first. This can lead to unexpected outcomes, if one assumes that some state does not change until the transaction is finally executed, as any other transaction that is executed first may call functions and change state. Such state changes are not arbitrary, though. The invariants still hold between the state in which a transaction was issued and the state in which it is executed. To prove properties of the contract given some issued state one can use the `issued` function. It works similar to `old`, but instead of the state before the function it gives us access to the state in which the transaction was issued. For example, in Listing 3.11 we have a contract where money is payed out after some period ended, i.e., after `self.ended` has been set to true. From the invariant we know that the period can never 'unend', i.e., once it is true

```
ended: bool
payouts: map(address, wei_value)

#@ invariant: sum(self.payouts) <= self.balance
#@ invariant: old(self.ended) ==> self.ended

...

#@ ensures: issued(self.ended) ==>
#@   success(if_not=sender_failed)
@public
def withdraw():
    assert self.ended
    amount: wei_value = self.payouts[msg.sender]
    self.payouts[msg.sender] = 0
    send(msg.sender, amount)
```

Listing 3.11: Contract specification using issued state.

it stays that way. Therefore, we can prove that if when the transaction was issued `self.ended` was already true, the function will succeed, given that the sender does not fail (lines 9 and 10).

3.8 Problems Solved by Vyper

Some security problems mentioned in the literature are solved by using Vyper instead of Solidity. Apart from bugs resulting from a non-understanding of the Solidity semantics (with Vyper generally being simpler to understand), the most obvious one is overflows. Vyper checks every operation for over- and underflows, reverting the transaction when they occur. This means that it is not possible to go from a very small unsigned wei value to a huge one because of an underflow, which could potentially result in ether being stolen.

A frequently mentioned security problem results from the use of the low-level `call` and `send` functions. In Solidity, instead of reverting the current call when the external contract they called fails, i.e., propagating the exception, they return a Boolean value to indicate success or failure. If one forgets to handle a failure, bugs which can lead to loss of ether can occur. In Vyper, on the other hand, exceptions always propagate.

A third security problem that can happen in Solidity concerns the usage of resizable arrays. If a contract allows anyone to append elements to the array, and it also contains code that iterates over the entire array, it is possible to carry out a denial-of-service attack: One appends as many elements to the array as are needed to make a call to the function with the iteration so gas expensive that it no longer fits on a block, which makes it impossible to execute. Since Vyper only has constant arrays, the maximum gas cost of a call is known at compile time, where one can check that such a thing is not possible.

3.9 Summary

This section summarizes all specifications introduced in the previous sections and serves as an overview of the verifier's functionality.

All specifications start with `#@`, followed by a keyword, and a colon. The following table shows all types of specifications and what they do:

Keyword	Description	Level
`ensures`	A postcondition that specifies the behavior of the function.	Function
`check`	A check before each point where the execution changes to some other contract.	Function
`invariant`	A condition that is true in any public state. Used to specify general properties of a contract.	Contract
`always ensures`	A postcondition that needs to hold for every function.	Contract
`always check`	A check that needs to hold for every function.	Contract
`preserves`	Tells the verifier that a general postcondition may preserve information throughout an external call.	Contract

Function-level specifications need to be declared before their respective function, while contract-level specifications may occur anywhere.

All specifications may use special functions defined by the verifier. The functions and what they do are listed in the next table:

Function	Description
`implies(a, b)`	Logical implication.
`forall(vars, triggers, expr)`	A quantifier: `vars` is a map from variable names to types, `triggers` is an optional list of triggers[4], and `expr` is the quantified expression.
`sum(mp)`	The sum of all entries of the map `mp`.
`sent()`	The map that maps addresses to the amount of wei sent to that address.
`sent(a)`	The amount of wei sent to address `a`.
`received()`	The map that maps addresses to the amount of wei received from that address.
`received(a)`	The amount of wei received from address `a`.
`result()`	The return value of a function.
`old(expr)`	The expression `expr` evaluated in, depending on the context, the state before the function was called, or the last seen state.
`issued(expr)`	The expression `expr` evaluated in the state when the transaction was issued.
`accessible(a, v, func)`	States that at least `v` amount of wei is accessible to address `a`, (optional) by calling `func`.
`event(e, n)`	States that event `e` has been fired (optional) exactly `n` many times.
`success()`	True if the function succeeded, false if it reverted.
`success(if_not=out_of_gas)`	True if the function always succeeds, given that the transaction does not run out of gas, false otherwise.
`success(if_not=sender_failed)`	True if the function always succeeds, given that the sender does not fail and the transaction does not run out of gas, false otherwise.

The table lists all available functions, however, not all functions can be used in every type of specification. The last table shows which function (and

---

[4] Triggers are a way to tell the underlying SMT solver when to use the information provided by the quantifier.

which of the variables `msg` and `block`) may be used in postconditions, checks, invariants, or transitive postconditions:

	`ensures`	`check`	`invariant`	`preserves`
`implies`	✓	✓	✓	✓
`forall`	✓	✓	✓	✓
`sum`	✓	✓	✓	✓
`sent`	✓	✓	✓	✓
`received`	✓	✓	✓	✓
`result`	✓			
`old`	✓	✓	✓	✓
`issued`	✓	✓		
`accessible`			✓	
`event`		✓		
`success`	✓	✓		
`msg`	✓	✓		
`block`	✓	✓		✓

For example, `result` is not available anywhere but in the postconditions because it is not a general property of the contract, and is not defined anywhere but at the end of a function, while `msg` is defined in the body of functions, therefore it can additionally be used in checks, and `block` can be used in transitive postconditions because it does not change throughout a transaction.

Chapter 4

Verification and Encoding to Viper

To verify that a Vyper contract adheres to its specification, 2VYPER encodes it to a Viper program, and then invokes a Viper backend to verify it. The encoding is in many cases straightforward, as Viper is an imperative language with native support for methods and almost all data types used in Vyper. This chapter first gives an overview of the Viper language features relevant for this thesis in Section 4.1, followed by the encoding of the contract state in Section 4.2, the general structure in Section 4.3, and other parts of the encoding in the rest of the chapter.

4.1 Viper

Viper includes a lot of syntax used in normal programming languages, extended with special constructs for verification. Listing 4.1 shows a basic example of a Viper program: New data types can be declared as a *domain*, here we declare a polymorphic pair type. Domains may include *function declarations*, and *axioms* to define properties of these functions. The `init_ax` defines that the first element of an initialized pair is the first argument to the initializer, and that the second element of the initialized pair is the second argument to the initializer. We then use the domain in the method `pair_op`. Many statements (like `var`, `if`, `goto`) and expressions (like addition, multiplication) are similar to C-style programming languages, however, some are a bit different: Instead of returning a value from a function via a `return` statement, the result variable `r` needs to be declared and can then be assigned to like a normal variable. The `assume` statement can be used to assume conditions, e.g., in line 15 we assume that the first element of `p` is non-negative. Conversely, `assert` tries to prove that the condition holds at that point. There are also variants of these two statements can have the side effect of adding *permissions* to heap locations to the state, as in Viper heap accesses are only allowed if one has the appropriate permission for it. These variants are called `inhale` and `exhale`. Both permissions to fields and permissions to recursive *predicates*, abstractions of multiple heap locations, can be inhaled and exhaled.

```
domain Pair[T1, T2] {
   function first(p: Pair[T1, T2]): T1
   function second(p: Pair[T1, T2]): T2

   function init(t1: T1, t2: T2): Pair[T1, T2]

   axiom init_ax {
      forall t1: T1, t2: T2 :: { init(t1, t2) }
         first(init(t1, t2)) == t1 &&
         second(init(t1, t2)) == t2
   }
}

method pair_op(p: Pair[Int, Int]) returns (r: Int) {
   assume first(p) >= 0
   assume second(p) >= 0

   if (first(p) % 2 == 1) {
      goto end
   }

   var i: Int := 2
   r := i * (first(p) + second(p))

   assert r >= 0

   label end
}
```

Listing 4.1: Viper encodings of pairs with usage.

4.2 Types

Basic types can easily be encoded to Viper, as they can be modeled by native types with some additional assumptions for each variable. The encodings are shown in the following table:

Vyper	Viper	Type assumptions
`int128`	`Int`	
`bool`	`Bool`	
`uint256`	`Int`	`x >= 0`
`wei_value`	`Int`	`x >= 0`
`timestamp`	`Int`	`x >= 0`
`timedelta`	`Int`	`x >= 0`
`decimal`	`Int`	
`address`	`Int`	
`bytes32`	`Seq[Int]`	`|b| == 32`
`bytes[n]`	`Seq[Int]`	`|b| <= n`
`string[n]`	`Seq[Int]`	`|b| <= n`

All integer types are encoded as Viper integers. Unsigned integers additionally need a type assumption that they are non-negative. Decimals are fixed-point with 10 decimals, they can therefore be encoded as scaled integers[1], where the decimal `d` is encoded as the integer 10^{10} `d`. Currently, all number types are treated as unbounded, i.e., we do not assume maximum and minimum values that are caused by limited number of bits, however, this could be done in the future. Byte arrays and strings are encoded as sequences of bytes, which in turn are encoded as integers. For all array-like types we can add assumptions about their length, since the length is known from the Vyper type system. General arrays are encoded similarly, but one additionally needs to consider the type assumptions of the element type. For example, `uint256[100]` will be translated as `Seq[Int]` with the type assumptions (for a variable `a`)

```
|a| == 100 &&
forall i: Int :: 0 <= i && i < |a| ==> a[i] >= 0
```

i.e., that the length is 100 and all elements are non-negative.

4.2.1 Maps

Viper does not have built-in support for maps, Vyper maps are therefore encoded as a custom domain. Listing 4.2 contains the encoding of a polymorphic map used in 2vyper. We provide four functions:

1. An initializer: Since Vyper has total maps, we can simply pass the default value of the map as an argument to the initializer, which then assumes all (infinitely many) values to be equal to that value.
2. An equality function: Equality for maps is defined as all values for all keys being equal.

---

[1] This is also how they are implemented at the bytecode level.

```
domain Map[K, V] {

   function map_init(v: V): Map[K, V]
   function map_eq(m: Map[K, V], n: Map[K, V]): Bool
   function map_get(m: Map[K, V], k: K): V
   function map_set(m: Map[K, V], k: K, v: V): Map[K, V]

   axiom map_init_ax {
      forall v: V, k: K :: { map_get(map_init(v), k) }
         map_get(map_init(v), k) == v
   }

   axiom map_eq_ax {
      forall m: Map[K, V], n: Map[K, V] :: { map_eq(m, n) }
         (map_eq(m, n) <==> m == n) &&
         (map_eq(m, n) <==>
            forall k: K :: { map_get(m, k), map_get(n, k) }
               map_get(m, k) == map_get(n, k))
   }

   axiom map_set_ax {
      forall m: Map[K, V], k: K, v: V , kk: K::
         { map_get(map_set(m, k, v), kk) }
            map_get(map_set(m, k, v), kk) ==
               (k == kk ? v : map_get(m, kk))
   }
}
```

Listing 4.2: Map encoding.

3. A getter: The getter looks up a value for a key.
4. A setter: The setter returns a new map where the value for a key has been set.

The axioms for the initializer and the equality function are straightforward, for the getter and setter it is enough to model how a lookup changes after setting a value: For the key that was set, a lookup will return the new value, for any other key it will stay the same.

4.2.2 Structs

Structs are similar to maps: If one assigns an index to each struct member, one gets a map where the keys are the member indices and the values are the values stored in the struct. However, this is not possible with the encoding described before, as the members have different types. We therefore create

```
domain Struct {

   function struct_loc(s: Struct, m: Int): Int

}

domain StructOps[T] {

   function struct_get(l: Int): T
   function struct_set(s: Struct, m: Int, t: T): Struct

   axiom get_set_0_ax {
      forall s: Struct, m: Int, t: T ::
         { struct_loc(struct_set(s, m, t), m) }
            struct_get(struct_loc(struct_set(s, m, t), m))
               == t
   }

   axiom get_set_1_ax {
      forall s: Struct, m: Int, n: Int, t: T ::
         { struct_loc(struct_set(s, n, t), m) }
            m != n ==> struct_loc(s, m) ==
               struct_loc(struct_set(s, n, t), m)
   }
}
```

Listing 4.3: Struct encoding.

two domains: The `Struct` domain creates the type for all struct variables, and the `StructOps[T]` domain contains the struct operations. This allows us to have a getter and a setter for each possible member type, i.e., if we want to look up a Boolean we can simply use the getter that returns a `bool`, if we want an integer we would use `Int`, etc. This essentially defines each member for every possible type, but since we only look up a member with the type we know from the Vyper program, this is sound. Unfortunately, we cannot simply pass a member index to the getter, as then one could not specify when setting a value that all other values do not change, as that would require quantifying over types, which Viper does not allow. As a workaround, we add a layer of indirection: The `struct_loc` function takes a struct and a member index and maps it to an integer, the *location*. The getter (for some type) then takes such a location and returns the value stored at that location (of that type). When setting a value, instead of stating that all other values do not change, we can then simply state that the locations do not change, which is a quantification over integers, and ensures that the getter will still return the same value.

Similar to arrays, type assumptions for structs consist of all type assumptions for the members.

4.3 General Structure

The general structure of a contract consists of its state and its functions. The state can be encoded by using the struct encoding. We model the contract as a struct with each storage variable being a member. Additionally, we add the built-in member `balance` (a `wei_value`) and the ghost members `sent` and `received` (both maps from `address` to `wei_value`). The local variables `msg` and `block` are similarly encoded as structs.

Vyper functions are encoded as methods that take the same arguments, and return a success variable and potentially a result. Listing 4.4 shows the general structure of the function encoding[2] without the encoding of verification functions explained later in this chapter (for a full version see Appendix A). There are three `self` states:

- `self`: The current state of the contract.
- `old_self`: The last public state, is set to `self` before and after every function call. This is also the state to which `old` in checks refers, and which is used to assume and assert invariants.
- `pre_self`: The state of the contract before the function has been executed, i.e., the state to which `old` in postconditions refers to. Is set to `self` in the beginning.

Additionally, we declare the `block` and `msg` variables. For all these variables, and also the arguments, we assume the type assumptions. Then we assume the *unchecked invariants*, invariants that the verifier knows to be true and therefore never checks, and the invariants written by the user. Note that the `self` and `old` state of the invariant are given in parentheses). Next, we set `old_self` and `pre_self` to `self`, as they need to refer to the 'old' state before the function execution, and the success variable to `true`, since we did not (yet) revert. The last part of the set-up consists of assuming information about `msg`, namely that `msg.sender` is never 0 and for non-payable functions that `msg.value` is 0. For payable functions we instead increase `self.balance` and `self.received[msg.sender]`. After the body, the label `return` marks the code that is executed after a successful return. We non-deterministically branch as we cannot know whether we ran out

---

[2] For ease of presentation, we use the syntax `self.balance := self.balance + 1` instead of `self := struct_set(self, 0, struct_get(struct_loc(self, 0)) + 1)`.

```
method foo(...) returns (succ: Bool, ...) {
   var self: Struct
   var pre_self: Struct
   var old_self: Struct

   var block: Struct
   var msg: Struct

   <Inhale type assumptions of arguments>
   <Inhale type assumptions of self, block, and msg>
   <Inhale unchecked invariants>
   <Inhale invariants(self, self)>

   old_self := self
   pre_self := self
   succ := true

   inhale msg.sender != 0
   <If the function is payable>
   self.balance += msg.value
   self.received[msg.sender] += msg.value
   <else>
   inhale msg.value == 0
   <end>

   <Body>

   label return
   var out_of_gas: Bool
   if (out_of_gas) {
      goto revert
   }
   goto end

   label revert
   succ := false
   self := pre_self
   old_self := pre_self

   label end
   <Assert postconditions(self, pre_self)>
   <Assert checks(self, old_self)>
   var havoc: Int
   inhale havoc >= 0
   self.balance += havoc
   <Assert invariants(self, old_self)>
}
```

Listing 4.4: General structure of the function encoding.

of gas or not. If we did, we go to the label `revert`, where the success variable is set to `false`, and set the states of `self` and `old_self` back to the state before the function call. Finally, the `end` label marks the checking of specifications: First the postconditions and checks, then, after havocing the balance to model coinbase transactions, the invariants.

4.4 Calls

Internal calls, i.e., calls to functions of one's own contract, are always inlined. This is possible, since Vyper does not allow recursion. The encoding of external calls (Listing 4.5, again simplified, for the full version see Appendix A), on the other hand, is more complicated and consists of three stages: the set-up, the call itself, and the clean-up. First, we evaluate the arguments.

```
<Evaluate arguments>
if (self.balance < amount) {
   goto revert
}
self.sent := map_set(self.sent, to, map_get(self.sent, to) +
     amount)
self.balance := self.balance - amount
<Assert checks(self, old_self)>
<Assert invariants(self, old_self)>
old_self := self

var send_fail: Bool
if (send_fail) {
   goto revert
}
var havoc: Struct
self := havoc
<Assume type assumptions for self>

<Assume unchecked invariants>
<Assume invariants(self, old_self)
<Assume transitive postconditions(self, old_self)>
old_self := self
```

Listing 4.5: Encoding of a call to contract `to` with `amount` wei attached.

The only important arguments are `to`, the address of the contract we call, and `amount`, the amount of wei attached to the call. If none was given, we assume it to be 0. After that, we check the contract has enough ether. If not, we revert by jumping to the appropriate label. We then increase `sent(to)` by `amount` and decrease `self.balance` by the same amount. We are now in the public state in which we then have to assert both the checks and the invariants. Moreover, this is the public state to which we want to refer when we assume the invariant after the call, therefore we save it in `old_self`. The call itself simply consists of a revert if it failed, followed by havocing the state. This represents the arbitrary changes to the contract that could happen because of reentrancy. However, changes are not completely arbitrary. For one, the type assumptions still hold, of course. But also, in the last phase, we can assume the unchecked invariants and user-specified invariants, as we know that they did hold before the call, and are therefore preserved. The transitive postconditions can also be assumed, as we definitely know that either no reentrancy has occurred, or that the last reentrant call was executed to its end, therefore the postcondition holds. The last step is to again save to current public state in `old_self`, as this is the state to which the next check or invariant will need to refer to as their `old` state.

4.5 Statements

Various statements need a special encoding, as there is no built-in support for them in Viper.

4.5.1 Assertions

Assertions revert the current call, if the expression given is false. We can therefore encode assertions the same way we encoded failing sends, namely by going to the `revert` label if the condition does not hold. An assertion `assert x > 0` is therefore translated as

```
if (!(x > 0)) {
   goto revert
}
```

similar to Listing 4.5.

4.5.2 Loops

Vyper loops always have a statically known number of iterations. When encoding them to Viper, we can therefore unroll them. This has the advantage of not needing to write loop invariants, but may become very slow for a large number of iterations. Fortunately, because of gas consumption, loops in smart contracts are usually short enough for unrolling to be viable.

When unrolling the loop, `break` and `continue` become forward jumps. For example, Figure 4.1 shows the encoding of a Vyper loop to an unrolled Viper loop with jumps.

```
for i in range(2):
    if i == 0:
        continue
    if i == 1:
        break
    x += 1
else:
    x = -1
```

Listing 4.6: Vyper loop.

```
var i: Int
i := 0
if (i == 0) {
    goto continue_0
}
if (i == 1) {
    goto break
}
x := x + 1
label continue_0
i := i + 1
if (i == 0) {
    goto continue_1
}
if (i == 1) {
    goto break
}
x := x + 1
label continue_1
x := -1
label break
```

Listing 4.7: Viper encoding.

Figure 4.1: Loop encoding via unrolling and jumps.

4.6 Arithmetic Operations

Most Vyper arithmetic operations are natively supported in Viper. Addition and multiplication can encoded directly, as we use unbounded integers to represent the Vyper types. Subtraction of unsigned integers additionally requires a check that the result is non-negative, else the execution reverts. The same is true for division by 0, both for the division and the modulo operator. Since Viper uses floor division but Vyper division truncates towards 0, division and modulo are encoded as functions, as is the power operator `**`.

4.7 Verification Functions

Some verification functions, like `implies` and `forall`, can be encoded directly to their Viper equivalents. Additionally, the encoding of `sent` and `received` has already been discussed in Section 4.3.

4.7.1 Sum

The `sum` function denotes the infinite sum over all entries of a map from some key type to integers. In Listing 4.8 the encoding is given. We use a domain with a function `map_sum` to represent the sum of the map passed. Two axioms describe its properties: First, the sum of a map where all values are initialized to 0 is 0. Second, if we update a value in the map, the sum decreases by the old value and increases by the new value. This encoding is sufficient to track how the sum of a map changes when the map is updated. However, this does not give any information about the values in the map given the sum. For maps of unsigned values, we additionally add the type assumption that all values are at most as large as the sum, i.e.,

```
domain MapInt[K] {

   function map_sum(m: Map[K, Int]): Int

   axiom map_sum_init_ax {
      map_sum(map_init(0)) == 0
   }

   axiom map_sum_set_ax {
      forall m: Map[K, Int], k: K, v: Int ::
         { map_sum(map_set(m, k, v)) }
            map_sum(map_set(m, k, v)) ==
               map_sum(m) - map_get(m, k) + v
   }
}
```

Listing 4.8: Encoding of the `sum` function.

```
forall k: K :: map_get(m, k) <= map_sum(m)
```

for map `m` and key type `K`. This is useful when trying to prove that withdrawing ether is successful. For example, if each address has its balance stored in a map `self.balance_of`, and the sum of the ether values in the map is at most `self.balance`, we can prove that `self.balance_of[a] <= self.balance` for any `a` of type `address`, i.e., that a withdrawal of all ether recorded in `self.balance_of` is possible.

4.7.2 Success

The function `success()` without any arguments simply accesses the variable `succ` as shown in Listing 4.4. Success given that the execution does not run out of gas, written as `success(if_not=out_of_gas)`, is encoded as

```
!out_of_gas ==> succ
```

by using the `out_of_gas` variable also shown in Listing 4.4. To encode `success(if_not=sender_failed)` we add a variable of type `Bool` named `sender_failed`. In every external call we then change branch, where we revert if the function fails, to

```
if (send_fail) {
   inhale to == msg.sender ==> sender_failed
   goto revert
}
```

which assumes `sender_failed` if the recipient of the failed external call was the current `msg.sender`. Additionally, we change the gas check to

```
if (out_of_gas) {
   inhale sender_failed
   goto revert
}
```

With that we can encode calls to `success(if_not=sender_failed)` as

```
!sender_failed ==> succ
```

similar to `success(if_not=out_of_gas)`.

4.7.3 Events

Encoding events consists of two steps: Encoding the emission of an event, and encoding the `event` function. For this, we use Viper predicates, i.e., we encode an event as an abstract heap location. For example, the event `Transfer: event({to: address, val: wei_value})` becomes the predicate

```
predicate Transfer(to: Int, val: Int)
```

in Viper. When emitting an event we simply inhale the predicate with the appropriate arguments, e.g.,

```
inhale Transfer(1, 5)
```

adds the predicate `Transfer(1, 5)` with permission value 1 to the state. To then check that the event has been emitted, we then check its permission value, i.e., `event(1, 5)` is encoded as

```
perm(Transfer(1, 5)) == 1/1
```

where `1/1` is the permission value. To check that an event has been emitted `n` times one just needs to check for a permission value of `n`.

4.7.4 Issued State

The `issued` state is the state of the blockchain in which the current transaction was issued. To encode it, we add an additional self variable `issued_self`, for which we can assume the type assumptions and invariants. Also, we know that the invariants must hold between the issued state and the state at the beginning of the function, therefore we assume them as well. Then, we use the `issued_self` variable inside the `issued` function. For example, `issued(self.balance + 1)` is translated as `issued_self.balance + 1`.

4.7.5 Accessible

If some amount of ether is `accessible` to an address, we need to show that it is possible to withdraw the ether from the contract. For example, given the invariant

```
#@ invariant: accessible(self.balance, self.owner, self.
    withdraw())
```

we need to show that it is possible for `self.owner` to withdraw `self.balance` wei by calling `self.withdraw()`. We can do that as follows: We represent `accessible` as a predicate

```
    predicate accessible(tag: Int, to: Int, amount: Int)
```

with arguments to the function also added as arguments to the predicate (here none are required). The `tag` argument is a number associated with the invariant, and is used to detect which invariant failed if ether is not accessible. When inhaling the invariant in the function that is supposed to withdraw the ether, we will inhale the predicate. This means that we have a predicate with arguments `to` and `amount` in our state if and only if we need to prove that `amount` of wei is accessible to address `to` by calling the current function. At the end of the function, we check for each tag that, given that we need to prove that ether has been withdrawn, the ether has actually been withdrawn if the sender did not fail. For a given tag `tag`, this amounts to asserting that

```
forall t: Int, a: Int :: perm(accessible(tag, t, a) > none
    ==>
  (!sender_failed ==> succ) &&
    (map_get(pre_self.sent, t) - map_get(self.sent, t) >= a)
```

where the left-hand side of the `&&` means that the function succeeds, given that the sender does not fail, and the right-hand side means that the difference of ether sent between the beginning of the function and the end of the function is at least the amount of ether that should be accessible.

4.8 Additional Checks

To make verification sound we need some additional checks. First, invariants and transitive postconditions need to be checked for transitivity. For that, we create a new method and three self states. Then we assume the invariants for the second state where the first state is the `old` state, and also for the third state where the second state is the `old` state. To prove transitivity we assert that the invariant now holds for the third state with the first state being the `old` state. Checking transitivity for transitive postconditions works equivalently.

The second condition that needs to be checked is that we cannot prove that `self.balance` stays constant. We check invariants in a state where the balance has been havoced, but for transitive postconditions no such thing is done. Therefore, we add another method where we assume the transitive postconditions, increase the balance by an unknown amount, and then check them again.

Chapter 5

Implementation

The verifier (excluding the backends) is implemented in Python and partially based on Nagini [16]. In Section 5.1 we first describe the architecture of the verifier, then, in Section 5.2, some limitations are discussed.

5.1 Architecture

The entire verification process is split into five phases, each of which is described in this section.

Preprocessing and Parsing

The first step is to call the Vyper compiler to make sure that the passed file is a valid Vyper file. After that, an abstract syntax tree (AST) needs to be created. Vyper is almost a subset of Python, with the exception of `struct` and `contract` declarations. Therefore, the preprocessor replaces these declaration by class declarations, and changes specifications written in comments to actual Python statements. After that, the 'normal' Python parser (in the `ast` module) can be used to create the AST. The parser also collects the various types of specifications (invariants, general postconditions, checks, etc.) so they can be used in the translation phase.

Transformations

The transformer is used to replace declared constants by their value. This is important, because loop unrolling requires knowing the number of loop iterations, which may be a constant. For constants that depend on each other, or that contain operations, there is an interpreter that determines the value.

Analysis

Many operations behave differently based on their type, e.g., unsigned and signed subtraction. The analyzer annotates all AST nodes with types for the translator to use. It also checks whether function specifications use issued state, so that if not, it can be optimized away. Additionally, it calculates the heuristics for `accessible`.

Translation

The key phase is the translation. The work is split into multiple translators, each responsible for translating some substructure of a contract, namely functions, statements, expressions, specifications and types. Based on the encoding described in Chapter 4, the translators create a Viper AST to be used by the verifier. Each Viper AST node has a `position` with an ID attached. With the ID the corresponding Vyper node is recorded, so that in the end a potential verification error can be associated with Vyper code that caused it. Additionally, an error transformation rule may be attached. An error transformation is necessary if the error reason given by Viper does not match the desired output of the Vyper verifier. In such a case, the error is transformed, e.g., from an assertion failure to an invariant violation.

Verification

Verification consists to a large extent of the interaction with the Viper backend. The translated Viper program is passed to one of the Viper backends, which may produce verification errors. These errors are then mapped back to the Vyper level using the position IDs created by the translator. In the end, the verification result is output to the user.

5.2 Limitations

The verifier supports most of the Vyper language except for some built-in functions, most notably `selfdestruct`, `create_forwarder_to`, `sqrt`, and `method_id`, gas, and type conversions that require knowledge of the byte representation. These limitations are not inherent to the approach and could be lifted in the future. However, since the contract is ultimately translated to SMT, functions like `sqrt` for decimals and byte representations are hard to reason about precisely, and may need to be translated similarly to `sum` (see Section 4.7.1) as a function with important properties added as assumptions.

The implementation is sound, except that we do not model overflows of integer types due to their finite bit representation. This limitation could be lifted in the future as well.

Chapter 6

Evaluation

To evaluate the usability of the verifier we used it on two example contracts that can be found on the Vyper Github repository [17]: An implementation of an Ethereum token and a simple auction.

6.1 ERC-20 Token

The *ERC-20 Token Standard* [18] defines a common interface for Ethereum tokens that can then be traded by using smart contracts. Many such tokens exist in the Ethereum ecosystem. A Vyper implementation of a token conforming to the standard can be found at [19].

6.1.1 State

The contract state is shown in Listing 6.1. The `name`, `symbol`, and `decimals` storage variables are mostly for convenience. The token balance of each address is tracked in `balanceOf`, whereas `allowances` denotes the number of tokens that an address may spend for another address, e.g., the expression `self.allowances[0x1][0x2]` denotes the number of tokens that address `0x2` may spend for address `0x1`. The total supply of tokens is stored in `total_supply`. Last, the `minter` is the address that created the token, and may create new tokens. Two events are also declared: `Transfer` and `Approval`.

6.1.2 Functions

The contract has two types of functions. Functions that are used to move tokens are shown in Listing 6.2. A simple `transfer` is used to give tokens to some other address. The `transferFrom` function transfers tokens from any address to another. This operation is only allowed if `msg.sender` has been previously given permission to do so by calling the `approve` function.

Functions used to control the amount of tokens are given in Listing 6.3. The minter is allowed to create new tokens by calling `mint`. Anyone can destroy their own tokens with `burn`, or other's tokens with `burnFrom`, if they have permission. Both variations of `burn` internally use the private `_burn` function to destroy the tokens.

```
Transfer: event({
   _from: address, _to: address, _value: uint256
})
Approval: event({
   _owner: address, _spender: address, _value: uint256
})

name: public(string[64])
symbol: public(string[32])
decimals: public(uint256)

balanceOf: public(map(address, uint256))
allowances: map(address, map(address, uint256))
total_supply: uint256
minter: address

@public
def __init__(_name: string[64], _symbol: string[32],
  _decimals: uint256, _supply: uint256):
    init_supply: uint256 = _supply * 10 ** _decimals
    self.name = _name
    self.symbol = _symbol
    self.decimals = _decimals
    self.balanceOf[msg.sender] = init_supply
    self.total_supply = init_supply
    self.minter = msg.sender
    log.Transfer(ZERO_ADDRESS, msg.sender, init_supply)
```

Listing 6.1: State declaration of an ERC-20 token contract.

6.1.3 Verification

We will first prove that the implementation of `transfer` adheres to the ERC-20 standard:

> Transfers `_value` amount of tokens to address `_to`, and MUST fire the `Transfer` event. The function SHOULD throw if the message caller's account balance does not have enough tokens to spend.

The first part (transfers ...) of the standard is fairly imprecise. More rigorously stated, it means that balance of `msg.sender` decreases by `_value`,

```
@public
def transfer(_to: address, _value: uint256) -> bool:
    self.balanceOf[msg.sender] -= _value
    self.balanceOf[_to] += _value
    log.Transfer(msg.sender, _to, _value)
    return True

@public
def transferFrom(_from: address, _to: address,
  _value: uint256) -> bool:
    self.balanceOf[_from] -= _value
    self.balanceOf[_to] += _value
    self.allowances[_from][msg.sender] -= _value
    log.Transfer(_from, _to, _value)
    return True

@public
def approve(_spender: address, _value: uint256) -> bool:
    self.allowances[msg.sender][_spender] = _value
    log.Approval(msg.sender, _spender, _value)
    return True
```

Listing 6.2: Token transfer functions of an ERC-20 token contract.

whereas the balance of `_to` increases by the same amount. Postconditions to express that look as follows:

```
#@ ensures: success() and _to != msg.sender ==>
#@   self.balanceOf[msg.sender] ==
#@      old(self.balanceOf[msg.sender]) - _value
#@         and self.balanceOf[_to] ==
#@            old(self.balanceOf[_to]) + _value
#@ ensures: success() and _to == msg.sender ==>
#@   self.balanceOf == old(self.balanceOf)
```

The special case where `msg.sender == _to` is treated separately. The second part states that a `Transfer` event needs to be fired. Events can be specified via checks, in this case

```
#@ check: success() ==>
#@   event(Transfer(msg.sender, _to, _value))
```

works. The last part is optional, but this specific implementation adheres to it. Therefore, we can use the fact that postconditions may specify conditions in which a function is supposed to fail, and write the postcondition

```
@public
def mint(_to: address, _value: uint256):
    assert msg.sender == self.minter
    assert _to != ZERO_ADDRESS
    self.total_supply += _value
    self.balanceOf[_to] += _value
    log.Transfer(ZERO_ADDRESS, _to, _value)

@private
def _burn(_to: address, _value: uint256):
    assert _to != ZERO_ADDRESS
    self.total_supply -= _value
    self.balanceOf[_to] -= _value
    log.Transfer(_to, ZERO_ADDRESS, _value)

@public
def burn(_value: uint256):
    self._burn(msg.sender, _value)

@public
def burnFrom(_to: address, _value: uint256):
    self.allowances[_to][msg.sender] -= _value
    self._burn(_to, _value)
```

Listing 6.3: Token creation and destruction functions of an ERC-20 token contract.

```
#@ ensures: _value > old(self.balanceOf[msg.sender]) ==>
#@     not success()
```

to check that `transfer` correctly reverts when the balance is not sufficiently high for a transfer. A specification similar to the one for `transfer` can be done for every function in the ERC-20 token contract.

The standard also specifies when events need to be emitted in general. For `Transfer` it states

> MUST trigger when tokens are transferred, including zero value transfers.
>
> A token contract which creates new tokens SHOULD trigger a `Transfer` event with the `_from` address set to `0x0` when tokens are created.

To prove that every transfer of tokens results in an event, we specify that, whenever the balance of an address `b` decreases while the balance of some other address `a` increases, we see an event `Transfer` from the first address to the second of amount `old(self.balance[a]) - self.balance[a]`. Written as a check, this gives

```
#@ always check: forall({a: address, b: address},
#@   self.balanceOf[a] > old(self.balanceOf[a]) and
#@      self.balanceOf[b] < old(self.balanceOf[b]) ==>
#@         event(Transfer(b, a, self.balanceOf[a] -
#@            old(self.balanceOf[a]))))
```

Unfortunately, this does not include zero transfers. Since zero transfers do not change the state, we cannot know that it occurred in a general check. However, we can use a normal check in the specification of the `transfer` function, as described earlier. The second part of the standard states that a `Transfer` event should also occur when creating tokens. To prove that, we specify that whenever the balance of an address `a` increases, but no other address `b` where `b != a` changes, we see a `Transfer` event from `ZERO_ADDRESS` to `a`:

```
#@ always check: forall({a: address},
#@  old(self.balanceOf[a]) < self.balanceOf[a] and
#@   forall({b: address}, b != a ==>
#@    self.balanceOf[b] == old(self.balanceOf[b]))
#@      ==> event(Transfer(ZERO_ADDRESS, a,
#@        self.balanceOf[a] - old(self.balanceOf[a]))))
```

Similarly, it can also be proven that `Approval` events are fired correctly.

The verifier can also prove properties about this specific implementation. We will prove that:

1. The total amount of tokens recorded in `self.balanceOf` is equal to the total supply of tokens `self.totalSupply`.
2. The minter does not change.
3. Only the minter is allowed to create new tokens.

The first two are general properties of the contract. They can be expressed as invariants as follows:

```
#@ invariant: self.total_supply == sum(self.balanceOf)
#@ invariant: self.minter == old(self.minter)
```

The third property concerns access control. We use a check

```
#@ always check: msg.sender != self.minter ==>
#@   old(self.total_supply) >= self.total_supply
```

to specify that every time `msg.sender` is not the minter, the total supply of tokens may only be at most the old total supply.

6.1.4 Summary

With a combination of invariants, checks, and postconditions it was possible to prove that the implementation of the ERC-20 token adheres to the standard. Of particular importance are the quantifiers, as they allowed us to prove properties of, e.g., all pairs of addresses, and the `event` function, since most of the properties that standard requires concern the correct firing of events. Additionally, we were able to use checks to prove correct access control for minting tokens. The entire example with specifications can be found in Appendix B.

6.2 Auction

To evaluate the usability of the verifier for specifying valid ether flow, we examine a contract specifying a simple open auction[1] [20].

6.2.1 State

Listing 6.4 shows the state of the auction contract. It consists of the beneficiary, i.e., the address that is selling something, the timestamps marking the auction start and end, the highest bidder, which is initially `0x0`, and the current highest bid, also initially 0. The contract start in a state where `self.ended` is false, and all entries in `self.pendingReturns` are 0.

6.2.2 Functions

The contract has three functions, as shown in Listing 6.5: `bid` can be called with an ether amount to overbid the current highest bidder. If the amount is high enough, `msg.sender` becomes the new highest bidder, `msg.value` becomes the new highest bid, and the pending amount of ether for the previous highest bidder is increased by the old highest bid, so they can get a refund. To actually get a refund, one has to call `withdraw`, which simply sends all ether recorded in `self.pendingReturns` for `msg.sender` back to them. The auction expires after some time. To end it and send the highest amount to the beneficiary, someone has to call `end`.

---

[1] The code was slightly adapted so that the beneficiary of the auction cannot be a bidder at the same time. This makes the presentation of the verification easier, as one does not need to specify this special case separately.

```
beneficiary: public(address)
auctionStart: public(timestamp)
auctionEnd: public(timestamp)

highestBidder: public(address)
highestBid: public(wei_value)

ended: public(bool)

pendingReturns: public(map(address, wei_value))

@public
def __init__(_beneficiary: address,
       _bidding_time: timedelta):
   assert _beneficiary != ZERO_ADDRESS

   self.beneficiary = _beneficiary
   self.auctionStart = block.timestamp
   self.auctionEnd = self.auctionStart + _bidding_time
```

Listing 6.4: State declaration of an auction contract.

6.2.3 Verification

The goal is to verify that the contract correctly handles all ether, i.e., that the only effective ether flow goes from the highest bidder in the end to the beneficiary. Everyone else can get a refund of all sent ether. Additionally, we specify some functional properties that should hold, namely:

1. The beneficiary does not change.
2. Once the auction has ended, it does not start again.
3. The highest bidder is only `0x0` if the highest bid is 0 (i.e., in the beginning).
4. The highest bid cannot decrease.
5. Once the auction has ended, the highest bidder and the highest bid do not change anymore.
6. If someone sends an amount higher than the highest bid, they become the new highest bidder.

The first five are general properties of the contract and we can express them as invariants. The last property requires local state, namely `msg.sender` and `msg.value`, hence we use a general postcondition. Translated to specifications, the properties are:

```
@public
@payable
def bid():
   assert block.timestamp < self.auctionEnd
   assert not self.ended
   assert msg.value > self.highestBid
   assert msg.sender != self.beneficiary

   self.pendingReturns[self.highestBidder] +=
      self.highestBid
   self.highestBidder = msg.sender
   self.highestBid = msg.value


@public
def withdraw():
   amount: wei_value = self.pendingReturns[msg.sender]
   self.pendingReturns[msg.sender] = 0
   send(msg.sender, amount)


@public
def endAuction():
   assert block.timestamp >= self.auctionEnd
   assert not self.ended

   self.ended = True

   send(self.beneficiary, self.highestBid)
```

Listing 6.5: Functions declarations of an auction contract.

```
#@ invariant: self.beneficiary == old(self.beneficiary)
#@ invariant: old(self.ended) ==> self.ended

#@ invariant: self.highestBidder == ZERO_ADDRESS ==>
#@    self.highestBid == 0
#@ invariant: self.highestBid >= old(self.highestBid)
#@ invariant: old(self.ended) ==>
#@   self.highestBid == old(self.highestBid) and
#@      self.highestBidder == old(self.highestBidder)
#@ always ensures: success() and
#@   msg.value > old(self.highestBid) ==>
#@      msg.sender == self.highestBidder
```

We now prove correct ether flow. We start with all the ether on the contract. Before the end of the auction, the total amount of ether on the contract, not including coinbase transactions, is the sum of all entries in `self.pendingReturns` plus `self.highestBid`, afterwards its is just the former. This can be expressed as

```
#@ invariant: not self.ended ==>
#@   sum(self.pendingReturns) + self.highestBid
#@      == sum(received()) - sum(sent())
#@ invariant: self.ended ==> sum(self.pendingReturns
#@   == sum(received()) - sum(sent())
#@ invariant: sum(received()) - sum(sent()) <= self.balance
```

Note that `sum(received()) - sum(sent())` denotes the total amount of ether received through function calls minus the total amount of ether sent, which, if we never send ether received through coinbase transactions, is the total amount of ether on the contract, again not including coinbase transactions.

Knowing that the total ether on the contract is as intended does not give precise guarantees. Therefore, we add the invariants

```
#@ invariant: self.highestBidder != self.beneficiary
#@ invariant: self.pendingReturns[self.beneficiary] == 0
#@ invariant: not self.ended ==> sent(self.beneficiary) == 0
#@ invariant: self.ended ==> sent(self.beneficiary) ==
#@   self.highestBid
```

to specify when the beneficiary is allowed to receive ether. The first invariant is required to show the second, that the beneficiary's pending returns are always 0, therefore they never get sent any ether before the auction ends. Afterwards, the ether sent to the beneficiary is exactly equal to the highest bid. For the highest bidder we have

```
#@ invariant: sent(self.highestBidder) + self.highestBid +
#@   self.pendingReturns[self.highestBidder] ==
#@      received(self.highestBidder)
```

i.e., that the total amount of ether already sent to the highest bidder (from previous bids that were overbid), the current highest bid, and the pending returns added sum up to exactly the total amount of ether received from the highest bidder. For all other addresses, it should hold that the total amount of ether received from the address is equal to the total amount of ether already sent back plus what is currently pending, as every non-successful address gets a complete refund. This can be expressed by the invariant

```
#@ invariant: forall({a: address}, a != self.highestBidder
#@   and a != self.beneficiary ==>
#@       sent(a) + self.pendingReturns[a] == received(a))
```

by using a quantifier. Note that the last two invariants also prove that we only send ether to addresses from which we already received ether, i.e., which are in use, and the beneficiary.

What remains to prove is that pending returns can actually be accessed by their respective owners. For that, we can use the `accessible` function to write

```
#@ invariant: forall({a: address, v: wei_value},
#@   v == self.pendingReturns[a] ==> accessible(a, v))
```

In this case, the heuristics are good enough so we do not need to pass the function as the third argument.

6.2.4 Summary

In this example we were able to show non-trivial properties about the behavior of the contract. In particular, the `sent` and `received` functions allow one to precisely specify how the contract handles ether, and that ether does not get lost. Furthermore, we could use `accessible` to show that every participant will get back their refunds. The entire example with specifications can be found in Appendix C.

6.3 Performance

We measured the performance of the verifier for the two examples. All measurements were taken on a 2019 iMac with a 3.6 GHz Intel Core i9 processor with 2VYPER at commit `b048f25fee6c77cf3a43b7ff811031a4b65f4520`. The results are shown in the following table:

Contract	Avg Time Normal [s]	Avg Time Benchmark [s]
ERC-20	8.55	5.29
Auction	7.54	4.10

The table shows the averages over 20 runs, where 'Normal' mode is simply an invocation of 2VYPER from the command line, including the JVM startup on other one-time setup, while 'Benchmark' mode only measures the time to verify the example, which is more indicative of the time it would take in a potential interactive mode used in an IDE. Both measurements used the symbolic execution backend of Viper. The results show that verification of Vyper contracts can be done efficiently in practice, which would allow the tool to be integrated in an IDE alongside the compiler.

In terms of scalability, loops are the most important issue. As loops are unrolled in the encoding, many loop iterations ($\gtrsim 50$) slow down the verification, especially if it contains many branches. Often, the symbolic execution Viper backend cannot handle those cases, as it needs to consider each branch separately, therefore the verification condition generation backend should be used.

Chapter 7

Conclusion and Future Work

In this thesis, we introduced 2VYPER, a verifier for smart contracts written in Vyper. We described specification types for functional correctness and to prove the absence of various security problems. This includes 'traditional' specification types like postconditions and invariants, but also specifications for sent and received ether, success and failure of a function, access control and events, absence of denial-of-service, and unpredictable state. We showed how to encode these specifications to the intermediate verification language Viper, where they ultimately get checked by an SMT solver. Moreover, we showed that the verifier is able to prove non-trivial properties of smart contracts by using these specifications.

Possible future work includes overflow checks, multi-contract specifications like checking that a contract adheres to an annotated interface, and a few currently unsupported language features. Additionally, integration with the Vyper compiler could be used to precisely handle gas, which could also give more precise information about the possible control flow to the compiler.

Listings

Bibliography

[1] G. Wood, "Ethereum: A secure decentralised generalised transaction ledger," 2014. `https://ethereum.github.io/yellowpaper/paper.pdf`.

[2] "Ethereum Foundation: The Solidity contract-oriented programming language." `https://github.com/ethereum/solidity`.

[3] "Ethereum Foundation: Pythonic smart contract language for the EVM." `https://github.com/ethereum/vyper`.

[4] N. Atzei, M. Bartoletti, and T. Cimoli, "A survey of attacks on Ethereum smart contracts sok," in *Proceedings of the 6th International Conference on Principles of Security and Trust - Volume 10204*, (New York, NY, USA), pp. 164–186, Springer-Verlag New York, Inc., 2017.

[5] "Mythril." `https://github.com/ConsenSys/mythril`.

[6] L. Luu, D.-H. Chu, H. Olickel, P. Saxena, and A. Hobor, "Making smart contracts smarter," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, CCS '16, (New York, NY, USA), pp. 254–269, ACM, 2016.

[7] P. Tsankov, A. Dan, D. Drachsler-Cohen, A. Gervais, F. Bünzli, and M. Vechev, "Securify: Practical security analysis of smart contracts," in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, CCS '18, (New York, NY, USA), pp. 67–82, ACM, 2018.

[8] J. Feist, G. Greico, and A. Groce, "Slither: A static analysis framework for smart contracts," in *Proceedings of the 2Nd International Workshop on Emerging Trends in Software Engineering for Blockchain*, WETSEB '19, (Piscataway, NJ, USA), pp. 8–15, IEEE Press, 2019.

[9] R. DeLine and R. Leino, "BoogiePL: A typed procedural language for checking object-oriented programs," Tech. Rep. MSR-TR-2005-70, March 2005.

[10] K. R. M. Leino, “This is Boogie 2,” June 2008.

[11] A. Permenev, D. Asenov, P. Tsankov, D. Drachsler-Cohen, and M. Vechev, “VerX: Safety verification of smart contracts.” IEEE S&P 2020. To appear.

[12] P. Müller, M. Schwerhoff, and A. J. Summers, “Viper: A verification infrastructure for permission-based reasoning,” in *Proceedings of the 17th International Conference on Verification, Model Checking, and Abstract Interpretation - Volume 9583*, VMCAI 2016, (New York, NY, USA), pp. 41–62, Springer-Verlag New York, Inc., 2016.

[13] “Vyper documentation.” `https://vyper.readthedocs.io/en/latest/index.html`.

[14] V. Dhillon, D. Metcalf, and M. Hooper, *The DAO Hacked*, pp. 67–78. Berkeley, CA: Apress, 2017.

[15] Á. Hajdu and D. Jovanovic, “SOLC-VERIFY: A modular verifier for Solidity smart contracts,” *CoRR*, vol. abs/1907.04262, 2019.

[16] M. Eilers and P. Müller, “Nagini: A static verifier for Python,” in *Computer Aided Verification* (H. Chockler and G. Weissenbacher, eds.), (Cham), pp. 596–603, Springer International Publishing, 2018.

[17] “Vyper Github repository.” `https://github.com/ethereum/vyper`.

[18] “ERC-20 token standard.” `https://github.com/ethereum/EIPs/blob/master/EIPS/eip-20.md`.

[19] “Vyper ERC-20 example contract.” `https://github.com/ethereum/vyper/blob/master/examples/tokens/ERC20.vy`.

[20] “Vyper auction example contract.” `https://github.com/ethereum/vyper/blob/master/examples/auctions/simple_open_auction.vy`.

Appendix A

Viper Encoding

```
method foo(...) returns (succ: Bool, ...) {
   var self: Struct
   var pre_self: Struct
   var old_self: Struct

   var block: Struct
   var msg: Struct

   <Inhale type assumptions of arguments>
   <Inhale type assumptions of self, block, and msg>
   <Inhale unchecked invariants>
   <Inhale invariants(self, self)>

   old_self := self
   pre_self := self
   succ := true

   inhale msg.sender != 0
   <If the function is payable>
   self.balance += msg.value
   self.received = map_set(self.received, map_get(self.received, msg.sender) + msg.value)
   <else>
   inhale msg.value == 0
   <end>

   <Body>

   label return
   var out_of_gas: Bool
   if (out_of_gas) {
      inhale sender_failed
      goto revert
   }
   goto end

   label revert
   succ := false
   self := pre_self
   old_self := pre_self

   label end
   <Assert postconditions(self, pre_self)>
   <Assert checks(self, old_self)>
   var havoc: Int
   inhale havoc >= 0
   self.balance += havoc
   <Assert invariants(self, old_self)>

   <For all tags tag>
   assert forall t: Int, a: Int :: perm(accessible(tag, t, a) > none ==> (!sender_failed ==> succ)
     && (map_get(pre_self.sent, t) — map_get(self.sent, t) >= a)
   <end>
}
```

Listing A.1: Entire function encoding.

```
<Evaluate arguments>
if (self.balance < amount) {
   goto revert
}
self.sent := map_set(self.sent, to, map_get(self.sent, to) + amount)
self.balance := self.balance − amount
<Assert checks(self, old_self)>
<Assert invariants(self, old_self)>
old_self := self

var send_fail: Bool
if (send_fail) {
   inhale to == msg.sender ==> sender_failed
   goto revert
}
var havoc: Struct
self := havoc
<Assume type assumptions for self>

<Assume unchecked invariants>
<Assume invariants(self, old_self)
<Assume transitive postconditions(self, old_self)>
old_self := self
```

Listing A.2: Entire encoding of a call to contract `to` with `amount` wei attached.

Appendix B

ERC-20 Token

```
#
# The MIT License (MIT)
#
# Copyright (c) 2015 Vitalik Buterin
#
# Permission is hereby granted, free of charge, to any person obtaining a copy
# of this software and associated documentation files (the "Software"), to deal
# in the Software without restriction, including without limitation the rights
# to use, copy, modify, merge, publish, distribute, sublicense, and/or sell
# copies of the Software, and to permit persons to whom the Software is
# furnished to do so, subject to the following conditions:
#
# The above copyright notice and this permission notice shall be included in
# all copies or substantial portions of the Software.
#
# THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR
# IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY,
# FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE
# AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER
# LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM,
# OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN
# THE SOFTWARE.
#

# This file was adapted from https://github.com/ethereum/vyper/blob/master/examples/tokens/ERC20.vy

# @dev Implementation of ERC-20 token standard.
# @author Takayuki Jimba (@yudetamago)
# https://github.com/ethereum/EIPs/blob/master/EIPS/eip-20.md

from vyper.interfaces import ERC20

implements: ERC20

Transfer: event({_from: indexed(address), _to: indexed(address), _value: uint256})
Approval: event({_owner: indexed(address), _spender: indexed(address), _value: uint256})

name: public(string[64])
symbol: public(string[32])
decimals: public(uint256)

balanceOf: public(map(address, uint256))
allowances: map(address, map(address, uint256))
total_supply: uint256
minter: address

#@ invariant: self.total_supply == sum(self.balanceOf)
#@ invariant: self.minter == old(self.minter)
#@ always check: implies(msg.sender != self.minter, old(self.total_supply) >= self.total_supply)
#@ always check: forall({a: address, b: address}, {self.balanceOf[a], self.balanceOf[b]}, implies(self.balanceOf[a] > old(self.balanceOf[a]) and self.balanceOf[b] < old(self.balanceOf[b]), event(Transfer(b, a, self.balanceOf[a] - old(self.balanceOf[a])))))
#@ always check: forall({a: address}, {self.balanceOf[a]}, {old(self.balanceOf[a])}, implies(old(self.balanceOf[a]) > self.balanceOf[a] and forall({b: address}, {old(self.balanceOf[b])}, {self.balanceOf[b]}, implies(b != a, self.balanceOf[b] == old(self.balanceOf[b]))), event(Transfer(a, ZERO_ADDRESS, old(self.balanceOf[a]) - self.balanceOf[a]))))
```

```
#@ always check: forall({a: address}, {self.balanceOf[a]}, {old(self.balanceOf[a])}, implies(old(self.balanceOf[a]) < self.balanceOf[a] and forall({b: address}, {self.balanceOf[b]}, {old(self.balanceOf[b])}, implies(b != a, self.balanceOf[b] == old(self.balanceOf[b]))), event(Transfer(ZERO_ADDRESS, a, self.balanceOf[a] - old(self.balanceOf[a])))))
#@ always check: forall({a: address, b: address}, {self.allowances[a][b]}, {old(self.allowances[a][b])}, implies(old(self.allowances[a][b]) < self.allowances[a][b], event(Approval(a, b, self.allowances[a][b]))))

@public
def __init__(_name: string[64], _symbol: string[32], _decimals: uint256, _supply: uint256):
   init_supply: uint256 = _supply * 10 ** _decimals
   self.name = _name
   self.symbol = _symbol
   self.decimals = _decimals
   self.balanceOf[msg.sender] = init_supply
   self.total_supply = init_supply
   self.minter = msg.sender
   log.Transfer(ZERO_ADDRESS, msg.sender, init_supply)

#@ ensures: implies(success(), result() == sum(self.balanceOf))
@public
@constant
def totalSupply() -> uint256:
   return self.total_supply

@public
@constant
def allowance(_owner: address, _spender: address) -> uint256:
   return self.allowances[_owner][_spender]

#@ ensures: implies(success() and _to != msg.sender,
#@   self.balanceOf[msg.sender] == old(self.balanceOf[msg.sender]) - _value
#@      and self.balanceOf[_to] == old(self.balanceOf[_to]) + _value)
#@ ensures: implies(success() and _to == msg.sender, self.balanceOf == old(self.balanceOf))
#@ ensures: implies(_value > old(self.balanceOf[msg.sender]), not success())
#@ check: implies(success(), event(Transfer(msg.sender, _to, _value)))
@public
def transfer(_to: address, _value: uint256) -> bool:
   self.balanceOf[msg.sender] -= _value
   self.balanceOf[_to] += _value
   log.Transfer(msg.sender, _to, _value)
   return True

#@ ensures: implies(_value > old(self.allowances[_from][msg.sender]), not success())
#@ check: implies(success(), event(Transfer(_from, _to, _value)))
@public
def transferFrom(_from : address, _to: address, _value: uint256) -> bool:
   self.balanceOf[_from] -= _value
   self.balanceOf[_to] += _value
   self.allowances[_from][msg.sender] -= _value
   log.Transfer(_from, _to, _value)
   return True

#@ check: implies(success(), event(Approval(msg.sender, _spender, _value)))
@public
def approve(_spender: address, _value : uint256) -> bool:
   self.allowances[msg.sender][_spender] = _value
   log.Approval(msg.sender, _spender, _value)
   return True

@public
def mint(_to: address, _value: uint256):
   assert msg.sender == self.minter
   assert _to != ZERO_ADDRESS
   self.total_supply += _value
   self.balanceOf[_to] += _value
   log.Transfer(ZERO_ADDRESS, _to, _value)

@private
def _burn(_to: address, _value: uint256):
   assert _to != ZERO_ADDRESS
   self.total_supply -= _value
   self.balanceOf[_to] -= _value
   log.Transfer(_to, ZERO_ADDRESS, _value)

@public
def burn(_value: uint256):
   self._burn(msg.sender, _value)

@public
def burnFrom(_to: address, _value: uint256):
   self.allowances[_to][msg.sender] -= _value
   self._burn(_to, _value)
```

Appendix C

Auction

```
#
# The MIT License (MIT)
#
# Copyright (c) 2015 Vitalik Buterin
#
# Permission is hereby granted, free of charge, to any person obtaining a copy
# of this software and associated documentation files (the "Software"), to deal
# in the Software without restriction, including without limitation the rights
# to use, copy, modify, merge, publish, distribute, sublicense, and/or sell
# copies of the Software, and to permit persons to whom the Software is
# furnished to do so, subject to the following conditions:
#
# The above copyright notice and this permission notice shall be included in
# all copies or substantial portions of the Software.
#
# THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR
# IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY,
# FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE
# AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER
# LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM,
# OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN
# THE SOFTWARE.
#

# This file was adapted from https://github.com/ethereum/vyper/blob/master/examples/auctions/simple_open_auction.vy


beneficiary: public(address)
auctionStart: public(timestamp)
auctionEnd: public(timestamp)

highestBidder: public(address)
highestBid: public(wei_value)

ended: public(bool)

pendingReturns: public(map(address, wei_value))


#@ invariant: self.beneficiary == old(self.beneficiary)
#@ invariant: implies(old(self.ended), self.ended)

#@ invariant: implies(self.highestBidder == ZERO_ADDRESS, self.highestBid == 0)
#@ invariant: self.highestBid >= old(self.highestBid)
#@ invariant: implies(old(self.ended), self.highestBid == old(self.highestBid) and self.highestBidder == old(self.highestBidder))
#@ always ensures: implies(success() and msg.value > old(self.highestBid), msg.sender == self.highestBidder)

#@ invariant: implies(not self.ended, sum(self.pendingReturns) + self.highestBid == sum(received()) - sum(sent()))
#@ invariant: implies(self.ended, sum(self.pendingReturns) == sum(received()) - sum(sent()))
#@ invariant: sum(received()) - sum(sent()) <= self.balance

#@ invariant: self.highestBidder != self.beneficiary
#@ invariant: self.pendingReturns[self.beneficiary] == 0
```

```
#@ invariant: implies(not self.ended, sent(self.beneficiary) == 0)
#@ invariant: implies(self.ended, sent(self.beneficiary) == self.highestBid)

#@ invariant: sent(self.highestBidder) + self.highestBid + self.pendingReturns[self.highestBidder] == received(self.highestBidder)
#@ invariant: forall({a: address}, {received(a)}, implies(a != self.highestBidder and a != self.beneficiary, sent(a) + self.pendingReturns[a] == received(a)))

#@ invariant: forall({a: address, v: wei_value}, {accessible(a, v)}, implies(v == self.pendingReturns[a], accessible(a, v)))


@public
def __init__(_beneficiary: address, _bidding_time: timedelta):
    assert _beneficiary != ZERO_ADDRESS

    self.beneficiary = _beneficiary
    self.auctionStart = block.timestamp
    self.auctionEnd = self.auctionStart + _bidding_time


@public
@payable
def bid():
    assert block.timestamp < self.auctionEnd
    assert not self.ended
    assert msg.value > self.highestBid
    assert msg.sender != self.beneficiary

    self.pendingReturns[self.highestBidder] += self.highestBid
    self.highestBidder = msg.sender
    self.highestBid = msg.value


@public
def withdraw():
    amount: wei_value = self.pendingReturns[msg.sender]
    self.pendingReturns[msg.sender] = 0
    send(msg.sender, amount)


@public
def endAuction():
    assert block.timestamp >= self.auctionEnd
    assert not self.ended

    self.ended = True

    send(self.beneficiary, self.highestBid)
```

Appendix D

Declaration of Originality

ETH

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Declaration of originality

The signed declaration of originality is a component of every semester paper, Bachelor's thesis, Master's thesis and any other degree paper undertaken during the course of studies, including the respective electronic versions.

Lecturers may also require a declaration of originality for other written papers compiled for their courses.

I hereby confirm that I am the sole author of the written work here enclosed and that I have compiled it in my own words. Parts excepted are corrections of form and content by the supervisor.

Title of work (in block letters):

Verification of Ethereum Smart Contracts Written in Vyper

Authored by (in block letters):
For papers written by groups the names of all authors are required.

Name(s):	**First name(s):**
Sierra	Robin

With my signature I confirm that
- I have committed none of the forms of plagiarism described in the 'Citation etiquette' information sheet.
- I have documented all methods, data and processes truthfully.
- I have not manipulated any data.
- I have mentioned all persons who were significant facilitators of the work.

I am aware that the work may be screened electronically for plagiarism.

Place, date	**Signature(s)**
Zürich, 29 September 2019	R. Sierra

For papers written by groups the names of all authors are required. Their signatures collectively guarantee the entire content of the written paper.