




Deductive Verification of Smart Contracts with Dafny

Franck Cassez^(✉) , Joanne Fuller, and Horacio Mijail Antón Quiles

ConsenSys, New York, USA

{franck.cassez,joanne.fuller,horacio.mijail}@consensys.net

Abstract. We present a methodology to develop verified smart contracts. We write smart contracts, their specifications and implementations in the verification-friendly language DAFNY. In our methodology the ability to write specifications, implementations and to reason about correctness is a primary concern. We propose a simple, concise yet powerful solution to reasoning about contracts that have *external calls*. This includes arbitrary re-entrancy which is a major source of bugs and attacks in smart contracts. Although we do not yet have a compiler from DAFNY to EVM bytecode, the results we obtain on the DAFNY code can reasonably be assumed to hold on Solidity code: the translation of the DAFNY code to Solidity is straightforward. As a result our approach can readily be used to develop and deploy safer contracts.

1 Introduction

The Ethereum network provides the infrastructure to implement a decentralised distributed ledger. At the core of the network is the Ethereum Virtual Machine [29] (EVM) which can execute programs written in EVM *bytecode*. This remarkable feature means that transactions that update the ledger are not limited to assets' transfers but may involve complex business logic that can be executed *programmatically* by *programs* called *smart contracts*.

Smart Contracts are Critical Systems. Smart contracts are programs and may contain bugs. For example, in some executions, a counter may over/underflow, an array dereference may be outside the range of the indices of the array. These runtime errors are vulnerabilities that can be exploited by malicious actors to attack the network: the result is usually a huge loss of assets, being either stolen or locked forever. There are several examples of smart contract vulnerabilities that have been exploited in the past: in 2016, a *re-entrance* vulnerability in the Decentralised Autonomous Organisation (DAO) smart contract was exploited to steal more than USD50 Million [13]. The total value netted from DeFi hacks in the first four months of 2022 [8], \$1.57 billion, has already surpassed the amount netted in all of 2021, \$1.55 billion.

Beyond runtime errors, some bugs may compromise the business logic of a contract: an implementation may contain subtle errors that make it deviate

from the initial intended specifications (e.g., adding one to a counter instead of subtracting one).

The presence of bugs in smart contracts is exacerbated by the fact that the EVM bytecode of the contract is recorded in the immutable ledger and cannot be updated. The EVM bytecode of a contract is available in the ledger, and sometimes the corresponding source code (e.g., in Solidity [11], the most popular language to write smart contracts) is available too, although not stored in the ledger. Even if the source code is not available, the bytecode can be decompiled which makes it a target of choice for attackers. Overall smart contracts have all the features of safety critical systems and this calls for dedicated techniques and tools to ensure they are reliable and bug-free.

Smart Contracts are Hard to Verify. Ensuring that a smart contract is bug-free and correctly implements a given business logic is hard. Among the difficulties that software engineers face in the development process of smart contracts are:

- The most popular languages Solidity, Vyper [27] (and in the early development stage its offspring Fe [12]) to write smart contracts have cumbersome features. For instance there is a default *fallback* function that is executed when a contract is called to execute a function that is not in its interface. Some features like the composition of function modifiers have an ambiguous semantics [31] and developing a formal semantics of Solidity is still a challenge [19]. There are defensive mechanisms (reverting the effects of a transaction, enforce termination with gas consumption) that aim to provide some safety. However, these mechanisms neither prevent runtime errors nor guarantee functional correctness of a contract.
- Most of the languages (e.g., Solidity, Vyper for Ethereum) used to develop smart contracts are not *verification-friendly*. It is hard to express safety (and functional correctness properties) within the language itself. Proving properties of a contract usually requires learning another specification language to write specifications and then embed the source code into this specification language.
- Smart contracts operate in an *adversarial environment*. For instance, a contract can call other contracts that are untrusted, and that can even call back into the first contract. This can result in subtle vulnerabilities like *re-entrancy*, which are caused by other contracts.

Our Contribution. We present a methodology to develop verified smart contracts. First, we write smart contracts, their specifications and implementations in the verification-friendly language DAFNY. This is in contrast to most of the verification approaches for smart contracts that build on top of existing languages like Solidity or Vyper and require annotations or translations from one language to another. In our methodology the ability to write specifications, implementations and to reason about correctness is a primary concern. Second, we use a minimal number of contract-specific primitives: those offered at the EVM level. This has the advantage of reducing the complexity of compiling a high-level language like DAFNY to EVM bytecode. Third, we propose a

simple, concise yet powerful solution to reasoning about contracts that have *external calls*. This includes arbitrary re-entrancy which is a major source of bugs and attacks in smart contracts. To summarise, our methodology comprises 3 main steps: 1) reason about the contract in isolation, *closed* contract, Sect. 2; 2) take into account possible exceptions, Sect. 3.1; 3) take into account arbitrary external calls, Sect. 3.2. Although we do not yet have a compiler from Dafny to EVM bytecode, the results we obtain on the DAFNY code can reasonably be assumed to hold on Solidity code: the translation of the DAFNY code in Solidity is straightforward. As a result our approach can readily be used to develop and deploy safer contracts.

Related Work. Due to the critical nature of smart contracts, there is a huge body of work and tools to test or verify them. Some of the related work targets highly critical contracts, like the deposit smart contracts [7, 22, 23], including the verification of the EVM bytecode.

More generally there are several techniques and tools¹ e.g., [2, 3, 9, 14, 15, 30], for auditing and analysing smart contracts written in Solidity [11] or EVM bytecode, but they offer limited capabilities to verify complex functional requirements or do not take into account the possibility of re-entrant calls.

Most of the techniques [1, 4, 10, 16, 18, 20, 26, 28] for the verification of smart contracts using high-level code implement a translation from Solidity (or Michelson for other chains) to some automated provers like Why3, F*, or proof assistants like Isabelle/HOL, Coq.

The work that is closest to our approach is [5]. In [5] a principled solution to check smart contracts with re-entrancy is proposed and based on instrumenting the code. Our solution (Sect. 3.2) is arguably simpler. Another difference is that [5] does not use the *gas* resource and is restricted to safety properties. Our approach includes the proof of termination using the fact that each computation has a bounded (though potentially arbitrary large) amount of resources. Modelling the gas consumption is instrumental in the solution we propose in Sect. 3.2 as it enables us to prove termination and to reason by well-founded induction on contracts with external calls.

2 Verification of Closed Smart Contracts

In this section, we introduce our methodology in the ideal case where the code of a smart contract is *closed*. By closed, we mean that there are no calls to functions outside (e.g., an external library or another smart contract) of the contract itself.

An Abstract View of the EVM. The Ethereum platform provides a global computer called the Ethereum Virtual Machine, EVM, to execute smart contracts.

In essence, smart contracts are similar to classes/objects in OO programming languages: they can be created/destroyed, they have a non-volatile *state*, and they offer some *functions* (interface) to modify their state. Smart contracts are

¹ <https://github.com/leonardoalt/ethereum.formal.verification.overview>.

usually written in high-level languages like Solidity or Vyper and compiled into low-level EVM *bytecode*. The EVM bytecode of a contract is recorded in the ledger and is immutable. The state of the contract can be modified by executing some of its functions and successive states' changes are recorded in the ledger.

Transactions and Accounts. Participants in the Ethereum network interact by submitting *transactions*. Transactions can be simple ETH (Ethereum's native cryptocurrency) transfer requests or requests to execute some code in a smart contract. The initiator of a transaction must bound the resources they are willing to use by providing a maximum amount of *gas* $maxG$. In the EVM, each instruction consumes a given (positive) amount of gas. The execution of a transaction runs until it (normally) ends or until it runs out of gas. Before running a computation, the initiator agrees on a *gas price*, gp , i.e., how much one gas unit is worth in ETH. At the end of the computation, if there is gl gas units left,² the initiator is charged with $(maxG - gl) \times gp$ ETH. To implement this type of bookkeeping, the initiator must have an *account*, the *balance* of which is larger than the maximum fee of $maxG \times gp$ ETH, before executing the transaction.

There are two types of accounts in Ethereum: a *user* account which is associated with a physical owner; and a *contract* account which is associated with a piece of code stored in the ledger. Both have a *balance*, stored in the ledger, which is the amount of ETH associated with the account. An account is uniquely identified by its (160-bit) *address*.

Execution of a Transaction. The execution of a transaction involving a contract account can be thought of as a *message call*: an account m sends a message to a contract account c to execute one of its functions $f(\cdot)$ with parameters x ; this call is denoted $c.f(x)$. The call can originate from a user account or from a contract account. When executing $c.f(x)$ some information about the caller m is available such as m 's account's address and the maximum amount of gas m is willing to pay for the execution of $c.f(x)$. The caller m can also transfer some ETH to c at the beginning of the transaction. The general form of a transaction initiated by m and involving a contract c is written:

$$m \rightarrow v, g, c.f(x)$$

where m is the *initiator* of the transaction, v the *amount of ETH* to be transferred to c before executing $f(x)$, and g the *maximum amount of gas* m is willing to pay to execute $c.f(x)$. To reason about the correctness of smart contracts in a high-level language (not EVM bytecode), we use some features that are guided by the EVM semantics:

- the values of m, v, g in a transaction are fixed; this means that we can write a transaction as a standard method call of the form $c.f(x, msg, g)$ where $msg = (m, v)$ by just adding these values as (read-only) parameters to the original function f . We specify all the contracts' functions in this form $c.f(x, msg, g)$. In msg , m is the *message sender*, $msg.sender$, v the message value, $msg.value$.

² The EVM tracks the amount of gas left relative to the maximum.

- The only requirement on the gas consumption is that every function consumes at least one unit of gas, and similar for every iteration of a loop. We use the gas value to reason about termination, and we do not take into account the actual gas cost that only makes sense on the EVM bytecode.

Specification with Dafny. To mechanically and formally verify smart contracts, we use the verification-friendly language DAFNY [17]. DAFNY natively supports Hoare style specification in the form of pre- and post-conditions, as well as writing proofs as programs, and offers both imperative, object-oriented and functional programming styles. The DAFNY verification engine checks that the methods satisfy their pre- and post-conditions specifications and also checks for the runtime errors like over/underflows. The result of a verification can be either “no errors” – all the methods satisfy their specifications –, “possible violation” of a specification – this may come with a counter-example – or the verification can time out. The form of verification implemented in DAFNY is *deductive* as the verifier does not try to synthesise a proof but rather checks that a program adheres to its specification using the available hints. The hints can range from bounds on integer values to more complex lemmas. We refer the reader to [17] for a more detailed introduction to the language and its implementation.

To model the concepts (transaction, accounts) introduced so far, we provide some data types and an `Account` trait, Listing A.1. A trait is similar to an interface in Java. It can be mixed in a class or in another trait to add some specific capabilities. The trait `Account` provides two members: the balance of the account and its type³ (contract or non-contract which is equivalent to user). For example, a user account can be created as an instance of the `UserAccount` class, line 16. A contract account is created by mixing in the `Account` trait and by setting the type of the contract accordingly: for instance, the `Token` contract, Listing A.2, mixes in `Account` providing the `balance` and `isContract` members. For high-level reasoning purposes it is enough to define a type `Address` as a synonym for `Account`.

Example: A Simplified Token Contract. We now show how to use our methodology to specify, implement and reason about a simple contract: a simplified `Token` contract. This contract implements a cryptocurrency: tokens can be minted and transferred between accounts. The contract’s functionalities are:

- the contract’s *creator* (an account) can mint new tokens at any time and immediately assign them to an account. This is provided by the `mint` function;
- tokens can be sent from an account `from` to another `to` provided the sender’s (`from`) balance allows it. This is provided by the `transfer` function.

The complete DAFNY code (specification and implementation) for the `Token` contract is given in Listing A.2:

- the contract is written as a class and has a *constructor* that initialises the values of the state variables;

³ In this paper we do not use any specific features related to the type of an account.

Listing A.1. Datatypes and Account Trait in DAFNY.

```

1  /** A message. */
2  datatype Msg = Msg(sender: Account, value: uint256)
3
4  type Address = Account
5
6  /** Provide an Account. */
7  trait Account {
8    /** Balance of the account. */
9    var balance : uint256
10
11   /** Type of account. */
12   const isContract: bool
13 }
14
15 /** A user account. */
16 class UserAccount extends Account {
17
18   constructor(initialBal: uint256)
19     ensures balance == initialBal
20   {
21     balance := initialBal;
22     isContract := false;
23   }
24 }

```

- each method has a *specification* in the standard form of predicates: the *pre-conditions*, *requires*, and the *post-condition*, *ensures*;
- the `Token` contract has a *global invariant*, `Ginv()`. The global invariant must be maintained by each method call. To ensure that this is the case, `Ginv()` is added to the pre- and post-conditions of each method⁴ (inductive invariant);
- the contract is instrumented with *ghost* variables, and possibly ghost functions and proofs. Ghost members are only used in proofs and do not need to be executable. Moreover, a ghost variable cannot be used to determine the behaviour of non-ghost methods for example in the condition of an `if` statement;
- the `sum(m)` function is not provided but computes the *sum* of the *values* in the map `m`;
- each method consumes at least one unit of gas and returns the gas left after when it completes.

The `Token` contract has two non-volatile state variables: `minter` and `balances`. The `minter` is the creator of the instance of the contract (constructor) and is a `constant`, which enforces it can be written to only once. Initially no tokens have been minted and the map that records the balances (in `Token`, not `ETH`) is empty (line 20). In this specification the creator of the contract is free to deposit some `ETH` into the contract account. Note that we can also specify Solidity-like attributes: for instance, `payable` is a Solidity attribute that can be assigned to a function to allow a contract to receive `ETH` via a call to this function. If a function is not payable, `ETH` cannot be deposited in the contract via this function. In our setting we can simply add a pre-condition: `msg.value == 0` (Listing A.2, line 36).

⁴ For the constructor it is only required to hold after the constructor code is executed.

The global invariant of the contract (line 9) states that the total amount of tokens is assigned to the accounts in `balances`. The ghost variable `totalAmount` keeps track of the number of minted tokens. The transfer method (line 32) requires that the source account is in the `balances` map whereas the target account may not be in yet. In the latter case it is added to the map. Note that the initiator must be the source account (`msg.sender == from`, line 36).

Verification of the Simplified Token Contract. The DAFNY verification engine can check whether implementations satisfy their pre-/post-conditions. In the case of the `Token` contract, DAFNY reports “no errors” which means that:

- there are no runtime errors in our program. For instance the two requirements `balances[from] >= amount` (line 34) and `balances[to] as nat + amount as nat <= MAX_UINT256` guarantee that the result of the operation is an `uint256` and there is no over/underflows at lines 50, 51. The update of a map m is written $m[k := v]$ and results in a map m' such that $m'[w] = m[w]$, $k \neq w$ and $m'[k] = v$ (lines 50, 51, 72).
- The global invariant `GInv()` must be preserved by each method call: if it holds at the beginning of the execution of a method, it also holds at the end. This global invariant must also hold after the constructor has completed. If DAFNY confirms `GInv()` holds everywhere, we can conclude that `GInv()` holds after any finite number of calls to either `mint` or `transfer`.
- There are some other pre- and post-conditions that are in the specifications. For example, the `old` keyword refers to the value of a variable at the beginning of the method and line 41 states that the balance of the `from` account has been decreased by `amount`.

The specification of the `Token` contract presented in this section assumes the pre-conditions hold for each message (method) call. In practice, this has to be ensured at runtime: it is impossible to force an initiator to submit a transaction that satisfies the pre-conditions of a method. However, this is a reasonable assumption as in case the pre-conditions do not hold, we can simply abort the execution. This kind of behaviour is supported by the EVM semantics where it is possible to return a status of a computation and abort (similar to an exception) the execution of the function and *revert* its effects on the contract’s state. Another more serious simplification of the `Token` contract is that there is no *external call* to another contract’s method. It turns out that external calls can be problematic in smart contracts and are the source of several attacks.

Listing A.2. A Simple Token Contract in DAFNY.

```

1  class Token extends Account {
2
3      const minter: Address // minter cannot be updated after creation
4      var balances : map<Address, uint256>
5
6      ghost var totalAmount: nat
7
8      /** Contract invariant. */
9      predicate GInv()
10         reads this`totalAmount, this`balances
11     {
12         totalAmount == sum(balances)
13     }
14
15     /** Initialise contract. */
16     constructor(msg: Msg)
17         ensures GInv()
18         ensures balance == msg.value && minter == msg.sender
19     {
20         isContract, minter, balances, balance := true, msg.sender, map[], msg.value;
21         totalAmount := 0;
22     }
23
24     /**
25     * @param from      Source Address.
26     * @param to        Target Address.
27     * @param amount    The amount to be transferred from `from` to `to`.
28     * @param msg       The `msg` value.
29     * @param gas       The gas allocated to the execution.
30     * @returns         The gas left after executing the call.
31     */
32     method transfer(from:Address,to:Address,
33         amount:uint256,msg:Msg,gas: nat) returns (g:nat)
34         requires from in balances && balances[from] >= amount && msg.sender == from
35         requires gas >= 1
36         requires msg.sender == from && msg.value == 0;
37         requires to !in balances ||
38             balances[to] as nat + amount as nat <= MAX_UINT256
39         requires GInv()
40         ensures GInv()
41         ensures from in balances && balances[from] >= old(balances[from]) - amount
42         ensures to in balances
43         ensures to != from ==> balances[to] >= amount
44         decreases gas
45         modifies this
46     {
47         balance := balance + msg.value;
48         var newAmount: uint256 := balances[from] - amount ;
49         balances :=
50             balances[to := (if to in balances then balances[to] else 0) + amount];
51         balances := balances[from := newAmount];
52     }
53
54     /**
55     * @param to        Target Address.
56     * @param amount    The amount to receiving the newly minted tokens
57     * @param msg       The `msg` value.
58     * @param gas       The gas allocated to the execution.
59     * @returns         The gas left after executing the call.
60     */
61     method mint(to:Address,amount: uint256,msg:Msg,gas:nat) returns (g:nat)
62         requires msg.sender == minter
63         requires gas >= 1
64         requires to !in balances ||
65             balances[to] as nat + amount as nat <= MAX_UINT256
66         requires GInv()
67         ensures totalAmount == old(totalAmount) + amount as nat
68         ensures GInv()
69         modifies this`balances, this`totalAmount
70     {
71         balances :=
72             balances[to := (if to in balances then balances[to] else 0) + amount];
73         // The total amount increases.
74         totalAmount := totalAmount + amount as nat;
75         g := gas - 1;
76     }
77 }

```

In the next section we show how to reason about smart contracts under adversarial conditions: exceptions and external calls.

3 Verification Under Adversarial Conditions

In this section we show how to take into account adversarial conditions: in the first section we describe how to move pre-conditions into runtime checks and enrich our specifications to precisely account for when a function call should revert. In the second part we propose a general mechanism to capture the possible adversarial effects of external calls.

3.1 Aborting a Computation

As mentioned before we cannot enforce the initiator of a transaction to satisfy any pre-conditions when calling a method in a smart contract. However, a simple way to handle exceptional cases is to explicitly check that some conditions are satisfied before executing the actual body of a method, and if it is not the case to abort the computation. In the EVM semantics this is known as a *revert* operation that restores the state of the contract before the transaction. The EVM has a special opcode, `Revert` to return the status of a failed computation.

In the previous section, we used pre-conditions to write the specification of the methods. We can automatically push these pre-conditions into runtime checks at the beginning of each method. To take into account the possibility of *exceptions* in a clean way, we lift the return values of each method to capture the status of a computation using a standard return generic type of the form `datatype Try<T> = Revert | Success(v: T)`. If a computation is successful, the value `v` of type `T` is returned and boxed in the `Success` constructor, otherwise `Revert` is returned.⁵

The implementation of the methods⁶ `mint` and `transfer` can be lifted using the return `datatype Try<T>` as in Listing A.3, line 1. This datatype allows for the return of arbitrary values of type `T` and as a special case when no value is returned, we can set `T = Unit` the type inhabited by a single value.

The new code (Listing A.3) introduces the following features:

- the conditions under the first `if` statements of `mint` and `transfer` (respectively at lines 22 and 47) are the negation of the conjunction of all the pre-conditions that are in Listing A.2.
- The pre-condition `GInv()` remains in the code. It is not a runtime check but a property of the contract that has to be preserved. This invariant is not part of the executable code.
- In this example of a closed contract we can characterise exactly when the transaction should revert (`r.T?` is true if and only if `r` is of type `T`). For instance the post-condition at line 7 precisely defines the conditions under which the method should not abort.

⁵ `Revert` is sometimes called `Failure` and can return a string error message.

⁶ The constructor has no pre-condition, so we can assume it always succeeds.

- The post-conditions at lines 15 and 40 ensures that the state of the contract (`balances`) is unchanged.

DAFNY returns “no errors” for this program, and we can conclude that the global invariant is always satisfied after any number of calls to `mint` or `transfer`. The code for each method does not enforce any pre-condition on the caller and can be translated into runtime checks at the EVM bytecode level.

Listing A.3. The Token Contract with Revert.

```

1  datatype Try<T> = Revert() | Success(v: T)
2
3  method transfer(from:Address,to:Address,amount:uint256,msg:Msg,gas:nat)
4  returns (g: nat, r: Try<>>)
5
6  requires GInv()
7  ensures // if r is of type Success
8  r.Success? <==>
9  (from in old(balances)
10  && old(balances[from]) >= amount
11  && msg.sender == from
12  && gas >= 1
13  && (to !in old(balances)||old(balances[to]) as nat + amount as nat<=MAX_UINT256))
14  /** State is unchanged on an revert. */
15  ensures r.Revert? ==> balances == old(balances)
16  ensures g == 0 || g <= gas - 1
17  ensures GInv()
18
19  decreases gas
20  modifies this
21  {
22  if !(from in balances && balances[from]>=amount && msg.sender==from && gas>=1
23  && (to !in balances || balances[to] as nat + amount as nat<=MAX_UINT256) ) {
24  return (if gas >= 1 then gas - 1 else 0), Revert();
25  }
26  var newAmount := balances[from] - amount;
27  balances := balances[to := (if to in balances then balances[to] else 0) + amount];
28  balances := balances[from := newAmount];
29  g, r := gas - 1, Success();
30  }
31
32  method mint(to:Address,amount:uint256,msg:Msg,gas:nat) returns (g:nat,r: Try<>>)
33  requires GInv()
34  ensures r.Success? ==> totalAmount == old(totalAmount) + amount as nat
35  ensures r.Revert? <==>
36  !(msg.sender == minter && gas >= 1 &&
37  (to !in old(balances)||
38  old(balances[to]) as nat + amount as nat<=MAX_UINT256))
39  // state unchanged on a revert.
40  ensures r.Revert? ==> balances == old(balances)
41  ensures g == 0 || g <= gas - 1
42  ensures GInv()
43
44  modifies this`balances, this`totalAmount
45  decreases gas
46  {
47  if !(msg.sender == minter && gas >= 1 &&
48  (to !in balances || balances[to] as nat + amount as nat<=MAX_UINT256)) {
49  return (if gas >= 1 then gas - 1 else 0), Revert();
50  }
51  balances := balances[to := (if to in balances then balances[to] else 0) + amount];
52  // The total amount increases.
53  totalAmount := totalAmount + amount as nat;
54  g, r := gas - 1, Success();
55  }

```

3.2 Reasoning with Arbitrary External Calls

We now turn our attention to smart contracts that have *external calls*. The semantics of the EVM imposes the following restrictions on the mutations of

Listing A.4. The Token Contract with a Notification.

```

method transfer(...) returns (g: nat, r: Try<()>)
{
  ...
  balances := balances[to := (if to in balances then balances[to] else 0) + amount];
  balances := balances[from := newAmount];
  // External call to contract `to`.
  // If we notify before updating balances, a re-entrant call may drain the contract
  // of its tokens.
  g, status := to.notify(from, amount, gas - 1);
  ...
}

```

state variables for contracts: the state variables of a contract c can only be updated by a call to a method⁷ in c . In other words another contract $c' \neq c$ cannot write the state variables of c .

Assume that when we transfer some tokens to a contract via the `transfer` method, we also *notify* the receiver. The corresponding new code for `transfer` is given in Listing A.4. If the method `notify` in contract `to` does not perform any external call itself, the segment `to.notify(·)` cannot modify the state variables of the `Token` contract, and the `Token` contract invariant `GInv()` is preserved. We may not have access to the code of `notify(·)` and may be unable to check whether this is the case.

If `notify` can call another contract it may result in unexpected consequences. For instance if the external call to the method `to.notify(·)` occurs before the update of `balances[from]`, `to.notify` may itself call (and collude with) `from` and call `from` to do the same transfer again. As a result many transfers will be performed (as long as some gas is left) and tokens will be *created* without a proper call to `mint`. The result is that the number of minted tokens does not correspond anymore to the number of tokens allocated to accounts, and the global invariant `Ginv()` does not hold anymore after `transfer`. This type of issue is commonly known as the *re-entrancy problem*. This vulnerability was exploited in the past in the so-called DAO-exploit [13].

There are several solutions to mitigate the re-entrancy problem. A simple solution is to require that calls to external contracts occur only as the last instruction in a method (Check-Effect-Interaction pattern [6]). This is a *syntactic sufficient condition* to ensure that every update on a contract's state occurs before any external calls. This enforces re-entrant calls to happen sequentially. A *semantic* approach for taking into external calls is proposed in [5] and rely on identifying segments of the code with external calls and adding some local variables to capture the effects of a call and reason about it.

We propose a similar but hopefully simpler technique⁸ to model external calls and their effects. Similar to [5] we do not aim to identify re-entrant calls but we want to *include and model* the effect of possible external (including re-entrant) calls and check whether the contract invariant can be violated or not. For the

⁷ We assume that all methods are *public*.

⁸ It can be implemented directly in DAFNY with no need for extra devices.

sake of simplicity we describe our solution to this problem on the `to.notify(·)` example, Listing A.4, and make the following (EVM enforced) assumptions on `to.notify(·)`:

- it always terminates and returns the gas left and the status of the call (revert or success),
- it consumes at least one unit of gas,
- it may itself make arbitrary external calls including callbacks to `transfer` and `mint` in the `Token` contract. As a result there can be complex nested calls to `transfer` and `mint`.

Our solution abstracts the call to `to.notify(·)` into a generic `externalCall`. The new code for the `transfer` method is given in Listing A.5. We model the effect of the external call `to.notify(·)` (line 17) by a call to the `externalCall(·)` method.

The idea is that `externalCall(·)` is going to generate all possible re-entrant calls including nested calls to `transfer`. To do so, we introduce some *non-determinism* to allow an external call to callback `transfer` and `mint`. This occurs at lines 51 and 57. Note that the parameters (`from`, `to`, `amount`, `msg`) of the re-entrant calls are randomly chosen using the `havoc<T>(·)` method that returns an arbitrary value of type `T`.

The code of `externalCall` works as follows:

- non-deterministically pick k and use it to decide whether a re-entrant call occurs or not (lines 42–58). There are three options, and we use $k \bmod 3$ to select among them. If $k \bmod 3 = 0$ (and there is enough gas left), a re-entrant call to `transfer` occurs. If $k \bmod 3 = 1$ a re-entrant call to `mint` occurs. Otherwise, ($k \bmod 3 = 2$), no re-entrant call occurs.
- finally (lines 61–69), we non-deterministically pick a boolean variable b to decide whether a new external call occurs.

We do not provide a formal proof that this captures all the possible re-entrant calls⁹, but rather illustrate that it models several cases. First, `externalCall` can simulate an arbitrary sequence `mint*` of calls to `mint`. This is obtained by selecting successive values of k such that $k \bmod 3 = 1$ and then selecting $b = \text{true}$. For instance, the sequence of values $k = 1, b = \text{true}, k = 1, b = \text{true}, k = 2, b = \text{false}$ simulates two reentrant calls to `mint`, i.e., `mint.mint`. As the gas value is also a parameter of all the methods and can be arbitrarily large, this model can generate all the sequences of calls in `mint*`. Second, `externalCall` can also simulate nested `transfer/mint` calls. For instance, the sequence of values $k = 0, b = \text{true}, k = 1, b = \text{false}$, simulates two reentrant calls to `transfer` with a nested call to `mint`. Third, nested calls to `transfer` can also be generated by `externalCall`. The sequence of values $k = 0, b = \text{true}, k = 0, b = \text{false}$ simulates two nested re-entrant calls to `transfer`.

The re-entrant calls can be executed with arbitrary inputs and thus the input parameters are *havoced* i.e., non-deterministically chosen and `externalCall` can

⁹ This is beyond the scope of this paper.

Listing A.5. The Token Contract with External Calls.

```

1  method transfer(from:Address,to:Address,amount:uint256,msg:Msg,gas:nat)
2      returns (g:nat,r:Try<()>)
3
4      ... // Ensures and requires same as Listing A.3
5
6  {
7      if !(from in balances && balances[from]>=amount && msg.sender==from && gas>=1
8          && (to !in balances || balances[to] as nat + amount as nat <= MAX_UINT256) ) {
9          return (if gas >= 1 then gas - 1 else 0), Revert();
10     }
11     var newAmount := balances[from] - amount;
12     balances := balances[to := (if to in balances then balances[to] else 0) + amount];
13     balances := balances[from := newAmount];
14     // If we swap the line above and the externalCall,
15     // we cannot prove invariance of GInv()
16     // At this location GInv() must hold which puts a restriction
17     // on where external call can occur.
18     var g1, r1 := externalCall(gas - 1); // to.notify( from, amount );
19     assert g1 == 0 || g1 <= gas - 1;
20     // We can choose to propagate or not the failure of external call.
21     // Here choose not to.
22     g, r := (if g1 >= 1 then g1 - 1 else 0), Success();
23 }
24
25 /**
26  * Simulate an external call with possible re-entrant calls.
27  *
28  * @param gas The gas allocated to this call.
29  * @returns The gas left after execution of the call and the status of the call.
30  *
31  * @note The state variables of the contract can only be modified by
32  * calls to mint and transfer.
33  */
34 method externalCall(gas: nat) returns (g: nat, r: Try<()>)
35     requires GInv()
36     ensures GInv()
37     ensures g == 0 || g <= gas - 1
38     modifies this
39     decreases gas
40 {
41     g := gas;
42     // Havoc `k` to introduce non-determinism.
43     var k: nat := havoc();
44     // Depending on the value of k % 3,
45     // re-entrant call or not or another external call.
46     if k % 3 == 0 && g >= 1 {
47         // re-entrant call to transfer.
48         var from: Address := havoc();
49         var to: Address := havoc();
50         var amount: uint256 := havoc();
51         var msg: Msg := havoc();
52         g, r := transfer(from, to, amount, msg, g - 1);
53     } else if k % 3 == 1 && g >= 1 {
54         // re-entrant call to mint.
55         var to: Address := havoc();
56         var amount: uint256 := havoc();
57         var msg: Msg := havoc();
58         g, r := mint(to, amount, msg, g - 1);
59     }
60     // k % 3 == 2, no re-entrant call.
61     // Possible new external call
62     var b:bool := havoc();
63     if b && g >= 1 {
64         // external call makes an external call.
65         g, r := externalCall(g - 1);
66     } else {
67         // external call does not make another external call.
68         g := if gas >= 1 then gas - 1 else 0;
69         r := havoc();
70     }
71 }
72
73 /** Havoc a given type. */
74 method {:extern} havoc<T>() returns (a: T)

```

generate an arbitrary number of external and re-entrant calls including nested calls to `transfer` and `mint`.

The key ingredient that allows us to reason and prove correctness is the `gas` value. We require that `gas` strictly decreases (line 38) after each recursive call. This is stated in DAFNY with the `decreases` clause. DAFNY checks that the value of the `gas` parameter strictly decreases on mutually recursive calls.

Our objective is now to prove, using this model of external calls, that the global invariant `GInv()` of the contract is always satisfied. This seems to be a complex task as our model includes an arbitrary and unbounded number of possibly nested external calls. The result is a mutually recursive program: `transfer` can call `externalCall` and `externalCall` can call `transfer` or `externalCall`. However, the property that the `gas` value strictly decreases on every call enables us to reason by induction. As the gas decreases on each new call, the induction is well-founded. And DAFNY can indeed prove that the global invariant `GInv()` is preserved by all the methods including an arbitrary number of possibly re-entrant `externalCalls`. Our solution provides a way to model the effects of external calls abstractly but conservatively while still being able to prove properties in modular manner in an adversarial environment modelled by `externalCall`. Compared to other approaches we also guarantee termination because we take into account the minimum amount of gas that computations take.

Note that `externalCall` has the pre-condition `GInv()`. This means that in `transfer` the predicate `GInv()` must be true before the call to `externalCall`. This amounts to having restrictions on where external calls can occur. However, without any knowledge of what external calls can do, this seems to be a reasonable restriction. For instance, if the external call has a callback to `mint` we can only prove the preservation of the invariant `Ginv()` if it holds before the call to `mint`. Of course if we have more information about an external call, e.g., we know it does not call back, we can also take it into account with our model: we can adjust `externalCall` to reflect this knowledge.

In our example, if we swap the lines 12 and 17 (Listing A.5), DAFNY cannot verify that `GInv()` is preserved by `transfer`. The reason is that the invariant `Ginv()` does not hold before the external call.

To the best of our knowledge this solution is the first that does not require any specific reasoning device or extension to prove properties of smart contracts under adversarial conditions, but can be encoded directly in a verification-friendly language.

Running Dafny and Reproducing the Verification Results. The code used in this paper omits some functions and proofs hints (like `sum`) and may not be directly verifiable with DAFNY. The interested reader is invited to check out the code in the repository <https://github.com/ConsenSys/dafny-sc-fmics> to get the full version of our contracts. The repository contains the code of the `Token` contract, a simplified auction contract and instructions how to reproduce the DAFNY verification results.

The auction contract demonstrates that global invariants (`GInv()` in `Token`) are not limited to specifying *state* properties but can also capture two-state

or multi-state properties. This can be achieved by adding ghost history variables using sequences. This type of specifications is expressive enough to encode some standard temporal logic properties on sequences of states of a contract, e.g., “Once the variable `ended` is set to true, it remains true for ever” in the simplified auction contract.

In our experiments, DAFNY can handle complex specifications and the contracts we have verified are checked with DAFNY within seconds on a standard laptop (MacBook Pro). The performance does not seem to be an issue at that stage, and if it would become an issue, there are several avenues to mitigate it: DAFNY supports modular verification, so we can break down our code into smaller methods; DAFNY has built-in strategies to manipulate the verification conditions and break them into simpler ones that can be checked independently.

4 Conclusion

We have proposed a methodology to model and reason about (Ethereum) smart contracts using the verification-friendly language DAFNY. The main features of our approach are: *i*) we encode the specifications and implementations of contracts directly in DAFNY with no need for any language extensions; *ii*) we take into account the possibility of *failures* and (arbitrary number of) *external calls*; *iii*) we specify the main properties of a contract using contract *global invariants* and prove these properties in a modular manner by a conservative abstraction of external calls with no need to know the code of externally called contracts.

To the best of our knowledge, our abstract model of the effect of external calls is new and the associated proof technique (mutually recursive method calls) is readily supported by DAFNY which makes it easy to implement.

We have tested our methodology on several contracts (e.g., Token, Simple Auction, Bank) and we believe that this technique can be used to verify larger contracts. Indeed, we can take advantage of the modular proof approach (based on pre- and post-conditions) supported by Dafny to design scalable proofs.

Our current work aims to automate the methodology we have presented by automatically generating the different versions of a given contract (closed, revert, external calls) from a simple source contract.

The approach we have presented is general and not exclusive to Dafny, and our methodology can be implemented within other verification-friendly languages like Why3 [20], Whiley [25], or proof assistants like Isabelle/HOL [21] or Coq [24].

References

1. Ahrendt, W., Bubel, R.: Functional verification of smart contracts via strong data integrity. In: Margaria, T., Steffen, B. (eds.) ISoLA 2020. LNCS, vol. 12478, pp. 9–24. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-61467-6_2

2. Alt, L., Reitwiessner, C.: SMT-based verification of solidity smart contracts. In: Margaria, T., Steffen, B. (eds.) *ISoLA 2018*. LNCS, vol. 11247, pp. 376–388. Springer, Cham (2018). https://doi.org/10.1007/978-3-030-03427-6_28
3. Amani, S., Bégel, M., Bortin, M., Staples, M.: Towards verifying ethereum smart contract bytecode in Isabelle/HOL. In: Andronick, J., Felty, A.P. (eds.) *7th ACM SIGPLAN International Conference on Certified Programs and Proofs, CPP 2018*, pp. 66–77. ACM (2018). <https://doi.org/10.1145/3167084>
4. Bhargavan, K., et al.: Formal verification of smart contracts. In: *PLAS@CCS 2016*. pp. 91–96. ACM (2016). <https://doi.org/10.1145/2993600.2993611>
5. Bräm, C., Eilers, M., Müller, P., Sierra, R., Summers, A.J.: Rich specifications for ethereum smart contract verification. *Proc. ACM Program. Lang.* **5**(OOPSLA), 1–30 (2021). <https://doi.org/10.1145/3485523>
6. Britten, D., Sjöberg, V., Reeves, S.: Using coq to enforce the checks-effects-interactions pattern in DeepSea smart contracts. In: Bernardo, B., Marmsoler, D. (eds.) *3rd International Workshop on Formal Methods for Blockchains, FMBC@CAV 2021*. OASICS, vol. 95, pp. 3:1–3:8. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2021). <https://doi.org/10.4230/OASICS.FMBC.2021.3>
7. Cassez, F.: Verification of the Incremental Merkle Tree Algorithm with Dafny. In: Huisman, M., Păsăreanu, C., Zhan, N. (eds.) *FM 2021*. LNCS, vol. 13047, pp. 445–462. Springer, Cham (2021). https://doi.org/10.1007/978-3-030-90870-6_24
8. Choo, L.: Crypto is crumbling, and DeFi hacks are getting worse. <https://www.protocol.com/fintech/defi-hacks-web3>
9. ConsenSys Diligence: Mythx, <https://mythx.io/>
10. Dharanikota, S., Mukherjee, S., Bhardwaj, C., Rastogi, A., Lal, A.: Celestial: a smart contracts verification framework. In: *Formal Methods in Computer Aided Design, FMCAD 2021*, New Haven, CT, USA, pp. 133–142. IEEE (2021). https://doi.org/10.34727/2021/isbn.978-3-85448-046-4_22
11. Ethereum Foundation: Solidity documentation (2022). <https://docs.soliditylang.org/en/v0.8.14/>
12. Fe Team: Fe: an emerging smart contract language for the Ethereum blockchain (2022). <https://github.com/ethereum/fe>
13. Güçlütürk, O.G.: The DAO hack explained: unfortunate take-off of smart contracts (2018). <https://ogucluturk.medium.com/the-dao-hack-explained-unfortunate-take-off-of-smart-contracts-2bd8c8db3562>
14. Hajdu, Á., Jovanović, D.: SOLC-VERIFY: a modular verifier for solidity smart contracts. In: Chakraborty, S., Navas, J.A. (eds.) *VSTTE 2019*. LNCS, vol. 12031, pp. 161–179. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-41600-3_11
15. Hajdu, Á., Jovanovic, D., Ciocarlie, G.F.: Formal specification and verification of Solidity contracts with events (short paper). In: Bernardo, B., Marmsoler, D. (eds.) *2nd Workshop on Formal Methods for Blockchains, FMBC@CAV 2020*. OASICS, vol. 84, pp. 2:1–2:9. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2020). <https://doi.org/10.4230/OASICS.FMBC.2020.2>
16. da Horta, L.P.A., Reis, J.S., de Sousa, S.M., Pereira, M.: A tool for proving Michelson smart contracts in WHY3. In: *IEEE International Conference on Blockchain, Blockchain 2020*. Rhodes, Greece, pp. 409–414. IEEE (2020). <https://doi.org/10.1109/Blockchain50366.2020.00059>
17. Leino, K.R.M.: Accessible software verification with Dafny. *IEEE Softw.* **34**(6), 94–97 (2017). <https://doi.org/10.1109/MS.2017.4121212>

18. Marescotti, M., Otoni, R., Alt, L., Eugster, P., Hyvärinen, A.E.J., Sharygina, N.: Accurate smart contract verification through direct modelling. In: Margaria, T., Steffen, B. (eds.) ISO_{LA} 2020. LNCS, vol. 12478, pp. 178–194. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-61467-6_12
19. Marmsoler, D., Brucker, A.D.: A denotational semantics of solidity in isabelle/HOL. In: Calinescu, R., Păsăreanu, C.S. (eds.) SEFM 2021. LNCS, vol. 13085, pp. 403–422. Springer, Cham (2021). https://doi.org/10.1007/978-3-030-92124-8_23
20. Nehai, Z., Bobot, F.: Deductive proof of industrial smart contracts using why3. In: Sekerinski, E., et al. (eds.) FM 2019. LNCS, vol. 12232, pp. 299–311. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-54994-7_22
21. Nipkow, T., Paulson, L.C., Wenzel, M.: Isabelle/HOL — A Proof Assistant for Higher-Order Logic, LNCS, vol. 2283. Springer Heidelberg (2002). <https://doi.org/10.1007/3-540-45949-9>
22. Park, D., Zhang, Y.: Formal verification of the incremental Merkle tree algorithm (2020). <https://github.com/runtimeverification/verified-smart-contracts/blob/master/deposit/formal-incremental-merkle-tree-algorithm.pdf>
23. Park, D., Zhang, Y., Rosu, G.: End-to-end formal verification of ethereum 2.0 deposit smart contract. In: Lahiri, S.K., Wang, C. (eds.) CAV 2020. LNCS, vol. 12224, pp. 151–164. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-53288-8_8
24. Paulin-Mohring, C.: Introduction to the coq proof-assistant for practical software verification. In: Meyer, B., Nordio, M. (eds.) LASER 2011. LNCS, vol. 7682, pp. 45–95. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-35746-6_3
25. Pearce, D.J., Utting, M., Groves, L.: An introduction to software verification with whiley. In: Bowen, J.P., Liu, Z., Zhang, Z. (eds.) SETSS 2018. LNCS, vol. 11430, pp. 1–37. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-17601-3_1
26. Schiffl, J., Ahrendt, W., Beckert, B., Bubel, R.: Formal analysis of smart contracts: applying the KeY system. In: Ahrendt, W., Beckert, B., Bubel, R., Hähnle, R., Ulbrich, M. (eds.) Deductive Software Verification: Future Perspectives. LNCS, vol. 12345, pp. 204–218. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-64354-6_8
27. Vyper Team: Documentation (2020). <https://vyper.readthedocs.io/en/stable/>
28. Wesley, S., Christakis, M., Navas, J.A., Trefer, R., Wüstholtz, V., Gurfinkel, A.: Verifying SOLIDITY smart contracts via communication abstraction in SMARTACE. In: Finkbeiner, B., Wies, T. (eds.) VMCAI 2022. LNCS, vol. 13182, pp. 425–449. Springer, Cham (2022). https://doi.org/10.1007/978-3-030-94583-1_21
29. Wood, D.: Ethereum: a secure decentralised generalised transaction ledger (2022). <https://ethereum.github.io/yellowpaper/paper.pdf>
30. Wüstholtz, V., Christakis, M.: Harvey: A Greybox Fuzzer for Smart Contracts, pp. 1398–1409. Association for Computing Machinery, New York (2020). <https://doi.org/10.1145/3368089.3417064>
31. Zakrzewski, J.: Towards verification of Ethereum smart contracts: a formalization of core of solidity. In: Piskac, R., Rümmer, P. (eds.) VSTTE 2018. LNCS, vol. 11294, pp. 229–247. Springer, Cham (2018). https://doi.org/10.1007/978-3-030-03592-1_13