Formal and Executable Semantics of the Ethereum Virtual Machine in Dafny

Franck Cassez (✉), Joanne Fuller, Milad K. Ghale, David J. Pearce, and Horacio M. A. Quiles

ConsenSys, New York, USA
firstname.surname@consensys.net

Abstract. The Ethereum protocol implements a replicated state machine. The network participants keep track of the system state by: 1) agreeing on the sequence of transactions to be processed and 2) computing the state transitions that correspond to the sequence of transactions. Ethereum transactions are programs, called *smart contracts*, and computing a state transition requires executing some code. The Ethereum Virtual Machine (EVM) provides this capability and can execute programs written in EVM *bytecode*. We present a formal and executable semantics of the EVM written in the verification-friendly language Dafny: it provides (*i*) a readable, formal and verified specification of the semantics of the EVM; (*ii*) a framework to formally reason about bytecode.

1 Introduction

A distinctive feature of Ethereum is that transactions are programs, *smart contracts*, and computing a state transition requires to run the contract code to compute the next state. This capability is provided by the Ethereum Virtual Machine (EVM) that can execute programs written in EVM *bytecode*. The original and informal specification of the EVM is in the Yellow Paper [28].

As a decentralised platform, Ethereum encourages *client diversity*: network participants are free to choose which implementation of the EVM they want to run, and there are several implementations to choose from written in different languages e.g., Go, Java. All the EVM implementations must agree on the state transitions, otherwise the network would split and the blockchain would *fork*. However, the original specification in the Yellow Paper [28] has some known shortcomings: (*i*) it is hard to read and does not provide a formal semantics of the EVM and the bytecode; (*ii*) the lack of a formal semantics makes it hard for Ethereum client developers to guarantee that they interpret the Yellow Paper in a consistent way; (*iii*) designing compilers from high-level languages (e.g., Solidity[1]) to EVM bytecode without a formal semantics is error-prone and, without a precise semantics of the EVM, it is hard to design *certified* compilers (preserving of semantics from a high to a low-level language.).

[1] The most popular language to write smart contracts.

One can argue that existing implementations of the EVM (e.g., in Go, Java) provide a *de facto* semantics for it. Whilst this is true to some extent, such implementations do not enable *formal reasoning* about bytecode. Furthermore, whilst smart contracts can be written in high-level languages like Solidity, they must be compiled into EVM bytecode before being executed on the EVM. Tools for checking safety properties (e.g., absence of overflow, division by zero, etc) at the Solidity level are problematic if they cannot guarantee such properties hold at the bytecode level. One solution is to design a provably correct compiler, but this is a complex and long-term endeavour [17]. Alternatively we can provide techniques, supported by tools, to reason about properties of the bytecode. This is what we propose to do in this work.

Our Contribution. We present a complete and formal specification of the EVM in Dafny, available at https://github.com/ConsenSys/evm-dafny. We provide a formal semantics where the meaning of an instruction is given as a partial function that maps states to states. Our semantics is language-agnostic, readable and can be used as a reference for developers of EVMs or to aid compiler writers. Moreover, it is a complete and usable framework for formally reasoning about correctness of EVM bytecode using Dafny.

2 Background & Motivation

In this section we give an overview of the EVM and show how our formal specification in Dafny can be used to verify properties of bytecode programs.

The Ethereum blockchain stores the bytecode of the *contracts* into a database and each contract has its own permanent storage. In what follows, we assume a given contract and refer to storage as that allocated to this contract.

Instructions and States. The EVM [28] is a *stack-based* machine [28] which supports 142 instructions: arithmetic operations (e.g., `ADD`, `MUL`), comparisons and bitwise operations (e.g., `ISZERO`, `NOT`), cryptographic primitives (e.g., `SHA3`), environment information (e.g., `BALANCE`, `CALLVALUE`), block information (e.g., `NUMBER`, `GASLIMIT`), stack/memory/control flow (e.g., `PUSH`, `POP`, `MSTORE`, `SLOAD`, `JUMP`), logging (e.g., `LOG1`), and system operations (e.g., `CREATE`, `CALL`, `DELEGATECALL`). An *executing state* of the EVM is a tuple containing several components. We restrict our attention to the following subset of these components:

code: a sequence of n bytes indexed from 0 to $n-1$; The byte at index $0 \leq k < n$ is either an instruction *opcode* or an *immediate operand*. For instance the sequence $s = [\texttt{0x60}, \texttt{0x01}, \texttt{0x60}, \texttt{0x02}, \texttt{0x01}, \texttt{0x50}, \texttt{0x00}]$ corresponds to the *program* "`PUSH1 0x01; PUSH1 0x02; ADD; POP; STOP`". Here, the byte at $s[1]$ (`0x01`) is the operand of the instruction at $s[0]$ (`PUSH1`).

pc: the *program counter* (initially 0) identifies the next instruction to execute. For example, if `pc` is 4, executing the instruction at $s[4]$ (`ADD`) increments it by 1 so $s[5]$ is the next instruction to execute. When executing instructions

with operands (e.g., "`PUSH1 0x01`" at $s[0]s[1]$) the `pc` is incremented by $1 + v$ where v is the number of operands.

stack: a *stack* of 256-bit words (initially empty); instructions can push or pop the stack. For example, starting from an empty stack [], executing the instructions "`PUSH1 0x1; PUSH1 0x2`" gives [`0x02`, `0x01`]. Executing the `ADD` instruction from the stack [`0x02`, `0x01`] pops 2 operands, adds them and pushes the result yielding a new stack [`0x03` = `0x01` + `0x02`].

memory: a 256-bit addressable, contiguous array of bytes (initially empty). Memory is volatile and only available during the current program execution. Memory expands on-demand when a value is read or written to a given location (which incurs some cost in gas). Values can be read from/written to memory using the instructions `MLOAD`, `MSTORE`, `MLOAD8` or `MSTORE8`.

storage: a map from 256-bit addresses to 256-bit words which constitutes the contract's permanent storage. Storage can be read/written using the instructions `SLOAD` or `SSTORE`.

gas: the fuel left for future computations. Executing an instruction consumes *gas* in the EVM, and this ensures that no infinite computation can occur.

In the EVM, program execution may *abort* under exceptional cases including:

Out-of-gas: the gas left in the current state does not cover the cost of executing the next instruction (including cost of memory expansion if any);

Stack exceptions: the stack size cannot exceed 1024. Moreover, some instructions (e.g., `POP`) can only be executed if the stack has enough elements and otherwise the execution should abort.

The EVM has *failure states* to capture aborted computations. As a result, a *state* of the EVM is either a failure state or a non-failure state.

Bytecode Verification. Using our formal semantics, we can guarantee security properties of bytecode programs using the Dafny verifier. Dafny is a verification-friendly language and as such the code can be instrumented with predicates and pre- and postconditions that are checked by the verifier at *compile time*. We use this feature to prove properties on the bytecode. The following simple Dafny program illustrates a proof:

```
1 method AddBytes(x: u8, y: u8) {
2   // Initialise an EVM with some gas and the bytecode to execute.
3   var st := InitEmpty(gas:=1000, code:=[PUSH1,x,PUSH1,y,ADD]);
4   // Execute 3 compute steps
5   st := ExecuteN(st,3);
6   // Check that the top of the stack is the sum of x and y
7   assert st.Peek(0) == (x as u256) + (y as u256);
8 }
```

This simple code snippet illustrates several aspects of the verification process. First we can verify *family of programs* as the parameters `x,y` are arbitrary unsigned integers over 8 bits. This is done by creating an EVM and stepping

through the code, e.g., using the `ExecuteN` function. Second, we specify the expected property of the code using the `assert` statement (line 7) which is a *verification* statement: it is not *executed* at runtime as in conventional programming languages but *checked* at *compile-time*, and must hold for all inputs. For this program Dafny can prove automatically that the `assert` statement is never violated. The proof uses the semantics of opcodes that are invoked in the computation of `ExecuteN`. Note that if we change `u8` to `u256` the property does not hold as an overflow can occur in the execution of `ADD`: this is flagged by the Dafny verifier with "Cannot prove assertion at line 7". Another set of checks that are performed automatically are related to pre- and postconditions. For instance the `ADD` instructions requires at least two elements on the stack. This is specified by a precondition in the function that defines the semantics of `ADD`. If the code above had only one `PUSH1` instruction Dafny would flag that the `ADD` cannot be performed as a precondition is violated. Overall, this short code snippet demonstrates that we can specify and verify functional correctness properties of bytecode, and thanks to the pre- and postconditions used to specify the semantics of the instructions, we can detect/fix possible exceptions (e.g., stack overflow) before runtime.

The example in listing A.1 shows how we can reason about storage updates and exceptions (aborted computations).

Listing A.1: Verifying bytecode with Reverts.

```
1  const INC_CONTRACT := Code.Create([
2    // Put STORAGE[0] on stack and increment by one
3    PUSH1, 0x0, SLOAD, PUSH1, 1, ADD,
4    // If result non-zero branch to JUMPDEST, else REVERT
5    DUP1, PUSH1, 0xf, JUMPI, PUSH1, 0x0, PUSH1, 0x0, REVERT,
6    // Write result back to STORAGE[0] and return
7    JUMPDEST, PUSH1, 0x0, SSTORE, STOP]);
8
9  method IncProof(st: State) returns (st': State)
10     requires st.OK? && st.PC() == 0 && st.Gas() >= 40000 ...
11     requires st.evm.code == INC_CONTRACT
12     ensures st'.REVERTS? || st'.RETURNS?
13     ensures st'.RETURNS? <==> (st.Load(0) as nat) < MAX_U256
14     ensures st'.RETURNS? ==> st'.Load(0) == (st.Load(0) + 1) {
15   // Execute upto (and including) JUMPI.
16   var nst := ExecuteN(st,7);
17   // Consider branches separately
18   if nst.Peek(0) == 0 { // test top of the stack
19       assert nst.PC() == 0xa;
20       nst := ExecuteN(nst,3);
21       assert nst.REVERTS?;
22   } else {
23       assert nst.PC() == 0xf;
24       nst := ExecuteN(nst,4);
25       assert nst.RETURNS?;
26   }
27   return nst;
28 }
```

This contract code maintains a counter at storage location 0 which is incremented by one on every contract call. Initially, the contract storage is unconstrained in the input state `st` and, hence, any location can contain any value. The code of the contract aims to capture overflows and to revert if an overflow occurs. The intent is that either the contract reverts (overflow detected) or the counter is incremented by 1. Listing A.1 gives a Dafny proof of this.[2] The preconditions (lines 10–11) ensure that `st` is an execution (non-failure) state with `pc == 0`, empty stack, enough gas, and has the contract code to execute.

The postconditions (lines 12-14) specify that the computation either increments the counter (at storage location 0) or the computation reverts. The proof divides up into two essential parts: **1.** Execute the first 7 bytecodes and store the intermediate state in `nst`. **2.** An overflow occurs when the result of the addition is 0. So depending on the result at the top of the stack, `nst.Peek(0)`, we decide whether the rest of the computation will either succeed or revert. Dafny successfully verifies this code and guarantees the postconditions on lines 12-14 for all input states `st` satisfying the preconditions (lines 10-11). This provides strong guarantees about the bytecode: (*i*) it either reverts or computes the increment but never runs out of gas, nor ends up in an invalid state (e.g., stack overflow or underflow), (*ii*) the program terminates normally *if and only if* the initial value stored at location 0 is strictly less than `MAX_U256` (line 13), (*iii*) on normal termination, the value in storage location 0 is incremented by one (line 14).

3 The Dafny-EVM

Our EVM is written in Dafny and provides a definition of the semantics as a function mapping states to states. A key design decision made early on was to develop a *functionally pure* formalisation of the EVM. In this section we describe the main components of the Dafny-EVM and conclude with some observations.

Machine State. Line numbers hereafter refer to Listing A.3. A state of the EVM is a record containing various fields such as gas, pc, stack, code, memory.

Each module (state, stack, memory, ...) provides a datatype, possibly incorporating some contraints (e.g., `EvmState.T`). For brevity, we omit some fields which contain information about the enclosing transaction and the so-called *substate*. The `State` datatype (line 7) models normal execution (`OK`), failure (`INVALID`), returning (`RETURNS`), reverting (`REVERTS`), etc.

Stack, Memory and Storage. We have implemented several submodules to provide operations on stack/memory/storage. This is summarised in Fig. 1. We lift the operations on stack/memory/storage into the `State` datatype. In Dafny this is done by adding the functions right after the definition of a datatype (line 11). This allows us to compose them easily and improves readability. For instance the `Add` function that implements the semantics of opcode `ADD` is defined

[2] The code in the paper may not compile or verify as we have simplified it for clarity. The code in https://github.com/ConsenSys/evm-dafny compiles and verifies.

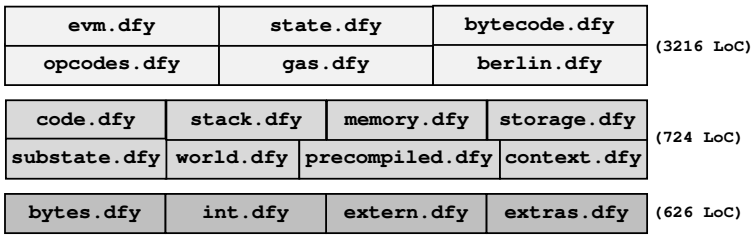

Fig. 1: Source files of the Dafny-EVM. Top group contains bytecode semantics and top-level types. Middle group contains abstractions of the main components. Bottom group are fundamental primitives (e.g. for manipulating bytes and ints). "Loc" (lines of codes) at the time of writing.

Listing A.2: Semantics of `MLOAD`, `Bytecode` module

```
1 function method MLoad(st: State) : State
2   requires st.IsExecuting() {
3 if st.Operands() >= 1 then
4   var loc := st.Peek(0) as nat;
5   var nst := st.Expand(loc,32);          // Break out expanded state
6   nst.Pop().Push(nst.Read(loc)).Next()   // Read from expanded state
7 else
8   State.INVALID(STACK_UNDERFLOW)
9 }
```

using a sequence of operations `st.Pop().Pop().Push(...).Next()` where `st` is an executing state (e.g. `OK`). We employ preconditions (`requires`) to ensure lifted operations are limited to applicable states only (typically executing states, such as `OK`), and also that preconditions of the functions on stack/memory/storage are satisfied (e.g., for `Pop()` the stack size must be large enough); for `Push()` (line 20) the stack cannot be full (stack size is limited to 1024).

In Dafny, preconditions are checked by the verifier and must provably hold at each call site. Notice that Dafny enforces the constraints on integer types so every time we compute (e.g., `ADD`) and store the result in a 256bit word, we must prove that the value is less than 2^{256} (the EVM dictates modulo arithmetic for this). The pre-/post-conditions and type checks enforced by the Dafny verifier help ensure that our EVM specification is consistent and that functions are well-defined.

Memory operations are provided by the `Memory` module, with various functions being attached to `State`, e.g., `Read, Write` lines 26–28. A key observation is that, in both cases, address `addr + 31` must be within allocated memory. This is because memory in the EVM is *byte addressable* and we are reading/writing `u256` values (i.e., which are 32 bytes long). The semantics of `MLOAD` (Listing A.2) highlights the complexity of memory operations. Since `Read(loc)` (line 6) has the precondition `loc + 31 < Memory.Size` (line 26 of Listing A.3), this must hold

for state `nst`. In fact, this follows because the call to `Expand()` (line 5) ensures sufficient memory. If the call to `Expand()` within `MLoad` was not enforcing this constraint, then DAFNY would raise a precondition violation on `nst.Read(loc)`.

Listing A.3: The `EvmState` module (partial)

```
1  module EvmState {
2    datatype Raw = EVM(gas:nat, pc:nat, stack:Stack.T, code:Code.T,
3                       mem:Memory.T, world:WorldState.T, ...)
4
5    type T = c:Raw | c.context.address in c.world.accounts
6
7    datatype State = OK(evm:T) | REVERTS(gas:nat,data:seq<u8>)
8        | RETURNS(gas:nat,data:seq<u8>,...) | INVALID(Error) | ...
9    {
10     // Predicates
11     predicate method IsExecuting(): bool { ... }
12
13     // Stack functions
14     function method Capacity(): nat
15       requires IsExecuting() { Stack.Capacity(evm.stack) }
16     function method Peek(k: nat): u256
17       requires IsExecuting() && k < Stack.Size(evm.stack) { ... }
18     function method Pop(): State
19       requires IsExecuting() && 0 < Stack.Size(evm.stack) { ... }
20     function method Push(v: u256) : State
21         requires IsExecuting()
22         requires Capacity() > 0 {
23             OK(evm.(stack:=Stack.Push(evm.stack,v)))
24     }
25     // Memory functions
26     function method Read(address: nat): u256
27       requires IsExecuting() && (addr+31) < Memory.Size(evm.mem) {...}
28     function method Write(address: nat, val: u256): State
29       requires IsExecuting() && (addr+31) < Memory.Size(evm.mem) {...}
30     ...
31     function method Expand(addr: nat, n: nat): (s': State)
32       requires IsExecuting()
33       ensures s'.IsExecuting() && MemSize() <= s'.MemSize()
34       ensures (addr + n) < MemSize() ==> (evm.mem == s'.evm.mem) {...}
35   }
36   ...
37 }
```

Gas. In our design, we chose to split out the *gas calculation* from the instruction semantics. Whilst this does introduce some repetition, we argue it reduces cognitive load. In particular, since this avoids interweaving the gas calculation throughout the instruction semantics which (for performance reasons) is commonly done in actual implementations (including the *execution specs*[3]).

[3] https://github.com/ethereum/execution-specs

Contract Calls. Various instructions (e.g. `CALL`, `DELEGATECALL`) enable one contract to call another. These differ from others as they can involve executing *arbitrarily* many instructions in the called contract. We implement this using a mechanism akin to *continuations* but, for brevity, omit the details here.

Observations. The Dafny-EVM Code provides a readable and executable specification of the EVM. There are several benefits of using a verification-friendly language: using pre- and postconditions to write the semantics provides a high level of assurance; furthermore, the code is executable and can be compiled into several target languages including Java, C#, Go. We now highlight some observations based on our experiences from this project.

- **Specification**. Dafny treats `function` calls within expressions as *interpreted*, but treats `method` calls as *uninterpreted* [5,15]. Roughly speaking this means that, when verifying a `function` call, the verifier has free access to the function's body. In contrast, for `method` calls, the verifier can only access what is given in the *specification* (i.e. its pre- and postconditions). As such, we consider methods ill-suited for formalising specifications (such as for the EVM). This is because we cannot abstract a specification any further than already done (i.e. we cannot specify a specification).
- **Verification**. Functions can have preconditions that restrict the domain of their inputs. In Dafny preconditions are enforced at each call site. We argue that this results in better code by enforcing consistency across function calls. Dafny enforces that every function must have a proof of termination which guarantees the absence of infinite loops in our state transition function. We believe that this degree of assurance is hard to attain with non verification-friendly languages.
- **Performance**. Code generated from the functionally pure subset of Dafny can perform poorly because of the need to clone compound structures (e.g. maps and arrays) to preserve purity (i.e., referential transparency). Dafny does not, for example, employ *clone elimination* [16,26,19] or *mutable value semantics* [22,21]. Performance was not a critical concern given our aim of developing a formal specification rather than an efficient implementation and in practice, we did not encounter any significant issue here.

During the project, a number of issues and challenges arose. For example, the lack of an exponentiation operator in Dafny meant that, for the `EXP` bytecode, we had to implement this as a recursive function. Some low level operations involving bits & bytes (e.g., shifting) present significant challenges as the native `int` type does not support bitwise operators. One can use a conversion from (e.g. `u256`) into the bitvector types (e.g. `bv256`) provided by Dafny which do support bitwise operations — however, this can lead to problems verifying code.

4 Practical Experiences

From the outset of this project, we were unsure whether Dafny would be practical for this sizeable formalisation task. Overall, however, we are pleased to report that Dafny has, for the most part, proven itself more than capable. Of course, it was not all plain sailing and we encountered several challenges which required developing techniques and/or workarounds.

Code Generation. Dafny can generate code for a variety of targets, including: C#, Go, Java, C++, Python and JavaScript. Furthermore, whilst Dafny does not support I/O operations *per se*, these can be implemented on the target side. We took advantage of this to embed the Dafny-generated code into a thin Java wrapper that performs I/O and allows us to test our EVM against existing implementations. Note that the generated code is not *proved* to be equivalent to the original Dafny code. For various reasons (e.g., knowledge within the team) we chose Java as the target language with `gradle` managing the build. This worked well enough, though there are some points to make:

- **Foreign Function Interface**. Code generated from Dafny does not conform to the stylistic norms of Java, but is otherwise relatively easy to interface with. A runtime library is provided by Dafny against which generated code must be compiled. This provides (amongst other things) alternative collection implementations (e.g. `DafnySequence`, `DafnyMap`, etc).
- **External Code**. For the semantics of `KECCAK256` and some precompiled contracts, we preferred to call out to native Java code (i.e. rather than implement e.g. `sha256` in Dafny itself). However, whilst Dafny does support `extern` declarations, these are not (at the time of writing) well supported by the Java code generator. Instead, we had to give default implementations (e.g. returning `0`) and employ build trickery to make it work.
- **Target language idiosyncrasies**. Translation to a target language introduces risks. E.g., Dafny employs *Euclidean Division* for its integer division operator (i.e. always rounds *down* rather than *towards zero*), which is a trap for the unwary and by chance we identified a bug in the Java code generator where sometimes standard division was being applied.[4] We also encountered unsoundness in the translation of Dafny collections (e.g. `seq<u8>`) to Java[5], and buggy implementation of `datatype` in C#. [6]

Verification and Testing. For completeness, we developed many unit tests for various components of our formalism. The *Ethereum Common Tests* also provide tens of thousands of tests for ensuring EVM compatibility.[7] As such, we have been using these to check our formalisation against existing implementations.

[4] https://github.com/dafny-lang/dafny/issues/2367
[5] https://github.com/dafny-lang/dafny/issues/2859
[6] https://github.com/dafny-lang/dafny/issues/1412
[7] https://github.com/ethereum/tests

This required generating *executable code* from our specification which presented several challenges (discussion of which is unfortunately omitted for brevity). At the time of writing, we have selected around 7500 representative tests out of the 13K Common tests (Berlin hardfork) and 6900 are passing (92%). Of the 143 failing tests, the majority (100) are failing because: some precompiled contracts are not yet fully implemented (44); we do not currently check for branches into instruction operands (56). The remaining (approx. 450) tests are skipped for various reasons e.g., timeout or breaking the testing system. Finally, we note that all of our tests are run as part of Continuous Integration before a pull request can be merged.

5 Related Work

Initial attempt at a formal specification of the EVM may be attributed to Hirai [13] with a formalisation of the EVM in the programming development environment Lem [18]. The formalisation in [13] is restricted to a single contract execution and proving bytecode is limited in terms of automation. Later, Amani *et al.* [3] built upon Hirai's formalisation and proposed an Isabelle/HOL formalisation. Their contribution introduces a program logic to reason about bytecode (restricted to a subset of 36/142 EVM instructions) but they rely on the construction of a control flow graph to define the semantics of a program. Reasoning about bytecode is limited to linear sequences of instructions (blocks) and not fully automated. Another Isabelle/HOL specification was also developed in [9] specialised for gas consumption analysis and for proving termination of bytecode.

More recently, Grishchenko *et al.* [10] have proposed a partial (not all opcodes are supported and the gas cost semamtics is incomplete) formalisation of the EVM in $F^\star$ targeting verification of security properties.

The most advanced formalisation is probably the KEVM [12] using the $\mathbb{K}$ Framework [23]. It provides a formal and executable specification of the syntax and semantics of EVM bytecode. Using the built-in automated tools of the $\mathbb{K}$ Framework, it is possible to generate an interpreter, compiler, debugger and to some extent a verifier that can be used to check the bytecode of some contracts [20]. The default input format (used for KEVM) of the $\mathbb{K}$ Framework is XML-based which may not be the most developer-friendly format. Similarly, IELE [24] attempts to design a more readable language than EVM bytecode and to be the target of high-level languages including Solidity, Vyper, Plutus. IELE is defined using the $\mathbb{K}$ Framework and uses LLVM tools (compiler) as a backend.

There are several implementations of the EVM in different languages and clients e.g., Geth[8], Besu[9], and more recently the *execution-specs* in Python[3]. The implementations in Geth and Besu are respectively in Go and Java and cannot be used to reason about bytecode. The Python implementation relies on specific imperative language features of Python (mutability, exceptions) and does

[8] https://geth.ethereum.org
[9] https://github.com/hyperledger/besu

not provide a functional definition of the instructions semantics nor an explicit specification of exceptional cases: for instance the Python code does not provide preconditions or explicit handling of exceptions, and exceptions can happen deep in the call stack which may hinder readability.

There are several tools Oyente [4], EtherIR [1], eThor [25], Rattle [27], and Certora [14] to perform static analysis of EVM bytecode. There are also extensions to specifically analyse the gas consumption like GASTAP [2], GasReducer [8]. Those tools build an abstract representation of the bytecode and it is unclear whether the abstraction is semantics preserving.

In contrast to the formalisations, implementations and tools referenced above, our formal semantics is language-agnostic (defines the state transition function as a function), easy to read and developer-friendly, provides mathematical and verified pre- and postconditions for the semantics of instructions. Moroever, our semantics can be used to perform *deductive reasoning* about bytecode including gas consumption using standard invariants.

6 Conclusion

We have proposed a formal semantics of the EVM in a pure functional subset of Dafny. Our semantics is *human readable*, *machine checked* and *executable*, and provides a sound framework to formally reason about bytecode.
This opens up the door for several direct applications:[10]

- complete smart contract verification: in practice, this can be a costly process and may require specific verification skills or familiarity with Dafny.
- correctness of compiler optimisations: several gas optimisation patterns e.g., a sequence `SWAP1 POP POP` optimised in `POP POP` can now be verified.
- correctness of under/overflow detection: to detect an overflow in arithmetic modulo `ADD(x, y)` it is common to first compute the result `r = ADD(x, y)` and then check that `r >= x`. We can formally prove that this is sound.
- synthesise verified bytecode: we have designed a methodology [6] to specify and verify smart contracts directly in Dafny. We are exploring *refinement proof techniques* to synthesise bytecode from the verified Dafny code of a contract. Ultimately we may develop a Dafny-to-EVM *certified compiler*.

Although the benefits of our approach are evident in the formal methods' community, adoption of these techniques in the Ethereum ecosystem is still challenging. Whilst established techniques, e.g., using Solidity to write contracts, or using Python to write specifications, can be questionable [11], they are still prevalent in the Ethereum community. The main hurdles for mainstream adoption of our approach are probably two-fold: (*i*) provide developer-friendly tools to write contracts; Dafny and the tool support around it (e.g., verification performance improvement, counter example generation [7], VSCode integration) already partially solves this issue; and (*ii*) educate the Ethereum community to understand the long-term benefits of formal verification for the Ethereum ecosystem.

[10] Examples are available in https://github.com/ConsenSys/evm-dafny.

References

1. Albert, E., Gordillo, P., Livshits, B., Rubio, A., Sergey, I.: Ethir: A framework for high-level analysis of ethereum bytecode. In: Lahiri, S.K., Wang, C. (eds.) Automated Technology for Verification and Analysis. pp. 513–520. Springer International Publishing, Cham (2018)
2. Albert, E., Gordillo, P., Rubio, A., Sergey, I.: Running on fumes - preventing out-of-gas vulnerabilities in ethereum smart contracts using static resource analysis. In: Ganty, P., Kaâniche, M. (eds.) Verification and Evaluation of Computer and Communication Systems - 13th International Conference, VECoS 2019, Porto, Portugal, October 9, 2019, Proceedings. Lecture Notes in Computer Science, vol. 11847, pp. 63–78. Springer (2019). https://doi.org/10.1007/978-3-030-35092-5_5
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.) Proceedings of the 7th ACM SIGPLAN International Conference on Certified Programs and Proofs, CPP 2018, Los Angeles, CA, USA, January 8-9, 2018. pp. 66–77. ACM (2018). https://doi.org/10.1145/3167084
4. Badruddoja, S., Dantu, R., He, Y., Upadhayay, K., Thompson, M.: Making smart contracts smarter. In: 2021 IEEE International Conference on Blockchain and Cryptocurrency (ICBC). pp. 1–3 (2021). https://doi.org/10.1109/ICBC51069.2021.9461148
5. Bradley, A.R., Manna, Z.: The calculus of computation - decision procedures with applications to verification. Springer (2007)
6. Cassez, F., Fuller, J., Anton Quiles, H.M.: Deductive verification of smart contracts with Dafny. In: Groote, J.F., Huisman, M. (eds.) Formal Methods for Industrial Critical Systems - 27th International Conference, FMICS 2022, Warsaw, Poland, September 14-15, 2022, Proceedings. Lecture Notes in Computer Science, vol. 13487, pp. 50–66. Springer (2022). https://doi.org/10.1007/978-3-031-15008-1_5
7. Chakarov, A., Fedchin, A., Rakamaric, Z., Rungta, N.: Better counterexamples for dafny. In: Fisman, D., Rosu, G. (eds.) Tools and Algorithms for the Construction and Analysis of Systems - 28th International Conference, TACAS 2022, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2022, Munich, Germany, April 2-7, 2022, Proceedings, Part I. Lecture Notes in Computer Science, vol. 13243, pp. 404–411. Springer (2022). https://doi.org/10.1007/978-3-030-99524-9_23
8. Chen, T., Li, Z., Zhou, H., Chen, J., Luo, X., Li, X., Zhang, X.: Towards saving money in using smart contracts. In: Zisman, A., Apel, S. (eds.) Proceedings of the 40th International Conference on Software Engineering: New Ideas and Emerging Results, ICSE (NIER) 2018, Gothenburg, Sweden, May 27 - June 03, 2018. pp. 81–84. ACM (2018). https://doi.org/10.1145/3183399.3183420
9. Genet, T., Jensen, T.P., Sauvage, J.: Termination of ethereum's smart contracts. In: Samarati, P., di Vimercati, S.D.C., Obaidat, M.S., Ben-Othman, J. (eds.) Proceedings of the 17th International Joint Conference on e-Business and Telecommunications, ICETE 2020 - Volume 2: SECRYPT, Lieusaint, Paris, France, July 8-10, 2020. pp. 39–51. ScitePress (2020). https://doi.org/10.5220/0009564100390051
10. Grishchenko, I., Maffei, M., Schneidewind, C.: A semantic framework for the security analysis of ethereum smart contracts. In: Bauer, L., Küsters, R. (eds.) Principles of Security and Trust - 7th International Conference, POST 2018, Held as Part of the European Joint Conferences on Theory and Practice of

Software, ETAPS 2018, Thessaloniki, Greece, April 14-20, 2018, Proceedings. Lecture Notes in Computer Science, vol. 10804, pp. 243–269. Springer (2018). https://doi.org/10.1007/978-3-319-89722-6_10
11. Guido, D.: Episode 6: What the hell are the blockchain people doing, and why isn't it a dumpster fire? (2021), https://galois.com/blog/2020/11/introducing-the-building-better-systems-podcast/, In Building Better Systems (podcast), Joey Dodds, Shpat Morina, Galois
12. Hildenbrandt, E., Saxena, M., Rodrigues, N., Zhu, X., Daian, P., Guth, D., Moore, B.M., Park, D., Zhang, Y., Stefanescu, A., Rosu, G.: KEVM: A complete formal semantics of the ethereum virtual machine. In: 31st IEEE Computer Security Foundations Symposium, CSF 2018, Oxford, United Kingdom, July 9-12, 2018. pp. 204–217. IEEE Computer Society (2018). https://doi.org/10.1109/CSF.2018.00022
13. Hirai, Y.: Defining the ethereum virtual machine for interactive theorem provers. In: Brenner, M., Rohloff, K., Bonneau, J., Miller, A., Ryan, P.Y.A., Teague, V., Bracciali, A., Sala, M., Pintore, F., Jakobsson, M. (eds.) Financial Cryptography and Data Security - FC 2017 International Workshops, WAHC, BITCOIN, VOTING, WTSC, and TA, Sliema, Malta, April 7, 2017, Revised Selected Papers. Lecture Notes in Computer Science, vol. 10323, pp. 520–535. Springer (2017). https://doi.org/10.1007/978-3-319-70278-0_33
14. Jackson, D., Nandi, C., Sagiv, M.: Certora technology white paper. Medium Post (2022), https://medium.com/certora/certora-technology-white-paper-cae5ab0bdf1
15. Kroening, D., Strichman, O.: Decision Procedures - An Algorithmic Point of View, Second Edition. Springer-Verlag (2016)
16. Lameed, N., Hendren, L.J.: Staged static techniques to efficiently implement array copy semantics in a MATLAB JIT compiler. In: Proceedings of the confererence on Compiler Construction (CC). pp. 22–41 (2011)
17. Leroy, X.: A formally verified compiler back-end. J. Autom. Reason. **43**(4), 363–446 (2009). https://doi.org/10.1007/s10817-009-9155-4
18. Mulligan, D.P., Owens, S., Gray, K.E., Ridge, T., Sewell, P.: Lem: reusable engineering of real-world semantics. In: Jeuring, J., Chakravarty, M.M.T. (eds.) Proceedings of the 19th ACM SIGPLAN international conference on Functional programming, Gothenburg, Sweden, September 1-3, 2014. pp. 175–188. ACM (2014). https://doi.org/10.1145/2628136.2628143
19. Odersky, M.: How to make destructive updates less destructive. In: Proceedings of the ACM symposium on the Principles Of Programming Languages (POPL). pp. 25–36 (1991)
20. 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.) Computer Aided Verification - 32nd International Conference, CAV 2020, Los Angeles, CA, USA, July 21-24, 2020, Proceedings, Part I. Lecture Notes in Computer Science, vol. 12224, pp. 151–164. Springer (2020). https://doi.org/10.1007/978-3-030-53288-8_8
21. Pearce, D.J., Groves, L.: Designing a verifying compiler: Lessons learned from developing Whiley. Science of Computer Programming pp. 191–220 (2015)
22. Racordon, D., Shabalin, D., Zheng, D., Abrahams, D., Saeta, B.: Implementation strategies for mutable value semantics. Journal of Object Technology **21**(2) (2022)
23. Rosu, G.: $\mathbb{K}$: A semantic framework for programming languages and formal analysis tools. In: Pretschner, A., Peled, D., Hutzelmann, T. (eds.) Dependable Software Systems Engineering, NATO Science for Peace and Security Series - D: Information and Communication Security, vol. 50, pp. 186–206. IOS Press (2017). https://doi.org/10.3233/978-1-61499-810-5-186

24. Runtime Verification: The iele virtual machine. Blog post (2022), https://runtimeverification.com/the-iele-virtual-machine/
25. Schneidewind, C., Grishchenko, I., Scherer, M., Maffei, M.: ethor: Practical and provably sound static analysis of ethereum smart contracts. In: Ligatti, J., Ou, X., Katz, J., Vigna, G. (eds.) CCS '20: 2020 ACM SIGSAC Conference on Computer and Communications Security, Virtual Event, USA, November 9-13, 2020. pp. 621–640. ACM (2020). https://doi.org/10.1145/3372297.3417250
26. Shankar, N.: Static analysis for safe destructive updates in a functional language. In: Proceedings of the Symposium Logic-Based Program Synthesis and Transformation (LOPSTR). pp. 1–24 (2001)
27. Trail of Bits: Rattle – an Ethereum EVM binary analysis framework. Medium Post (2018), https://blog.trailofbits.com/2018/09/06/rattle-an-ethereum-evm-binary-analysis-framework/
28. Wood, G.: Ethereum: a secure decentralised generalised transaction ledger. ethereum project yellow paper (2022), https://ethereum.github.io/yellowpaper/paper.pdf, berlin version d77a387 - 2022-04-26