arXiv:2303.00152v1 [cs.LO] 1 Mar 2023

Dafny 中以太坊虚拟机的形式化与可执行语义

Franck Cassez、Joanne Fuller、Milad K. Ghale、DavidJ. Pearce 和Horacio M. A. Quiles

ConsenSys，美国纽约，
firstname.surname@consensys.net

摘要。 以太坊协议实现了一种复制状态机。网络参与者通过以下方式跟踪系统状态：1）就待处理交易的顺序达成一致；2）计算与该交易序列相对应的状态转换。以太坊交易是称为 *smart contracts* 的程序，而计算状态转换需要执行一些代码。以太坊虚拟机（EVM）提供了这一能力，并可执行以太坊虚拟机 *bytecode* 编写的程序。我们提出了一种使用适用于验证的语言Dafny编写的以太坊虚拟机形式化且可执行的语义：它提供了（i）一种易读、形式化且经验证的以太坊虚拟机语义规范；（ii）一个用于对字节码进行形式化推理的框架。

1 引言

以太坊的一个显著特点是，交易本身就是程序，即智能合约；计算状态转换时，需要运行合约代码来计算下一个状态。以太坊虚拟机（EVM）能够执行以 EVM字节码编写的程序，正是它提供了这一能力。EVM 最初的非正式规范见黄皮书 [28]。

作为一个去中心化平台，以太坊鼓励客户端多样性：网络参与者可以自由选择要运行的 EVM 实现，而目前有多种使用不同语言编写的实现可供选择，例如 Go 和 Java。所有 EVM 实现都必须对状态转换达成一致，否则网络就会分裂，区块链也会分叉。然而，黄皮书 [28] 中的原始规范存在一些已知缺陷：(*i*) 难以阅读，且没有提供 EVM 和字节码的形式语义；(*ii*) 缺乏形式语义，使以太坊客户端开发者难以确保他们以一致的方式解读黄皮书；(*iii*) 在缺乏形式语义的情况下，设计从高级语言（例如 Solidity[1]）到 EVM 字节码的编译器容易出错；而且，如果没有对 EVM 的精确语义，也难以设计经认证的 编译器（确保语义从高级语言到低级语言得以保留）。

有人可能会认为，现有的以太坊虚拟机实现（例如用 Go、Java 编写的实现）为其提供了一种事实上的语义。虽然这种说法在一定程度上成立，但这类实现无法支持对字节码进行形式化推理。此外，智能合约可以用 Solidity 等高级语言编写，但在以太坊虚拟机上执行之前，必须先编译为 EVM 字节码。如果无法保证安全属性（例如不会发生溢出、除零等）在字节码层面同样成立，那么在 Solidity 层面检查这些属性的工具就存在问题。一种解决方案是设计可证明正确的编译器，但这是一项复杂且耗时的长期工作 [17]。另一种做法是提供由工具支持的技术，以便对字节码的属性进行推理。这正是本文的工作目标。

我们的贡献。我们提出了以太坊虚拟机的完整形式化规范，采用 Dafny编写，可在 https://github.com/ConsenSys/evm-dafny 获取。我们提供了一套形式化语义，将指令的含义定义为一个把状态映射到状态的偏函数。我们的语义与语言无关、易于阅读，可作为以太坊虚拟机开发者的参考，也可帮助编译器编写者。此外，它还是一个完整且实用的框架，可用于借助 Dafny 对 EVM 字节码的正确性进行形式化推理。

[1]编写智能合约最常用的语言。

2 背景与动机

本节将概述以太坊虚拟机，并展示如何使用 Dafny 中的形式化规范验证字节码程序的属性。

以太坊区块链将合约 的字节码存储在数据库中，并且每个合约都有自己的永久存储。下文中，我们假设给定一个合约，并将分配给该合约的存储称为存储。

指令与状态。以太坊虚拟机 [28] 是一种基于栈的机器 [28]，支持 142 条指令：算术运算（例如 ADD、MUL）、比较运算和按位运算（例如 ISZERO、NOT）、密码学原语（例如 SHA3）、环境信息（例如 BALANCE、CALLVALUE）、区块信息（例如 NUMBER、GASLIMIT）、栈/内存/控制流操作（例如 PUSH、POP、MSTORE、SLOAD、JUMP）、日志记录（例如 LOG1）以及系统操作（例如 CREATE、CALL、DELEGATECALL）。以太坊虚拟机的执行状态是一个包含若干组件的元组。我们仅关注这些组件中的以下子集：

代码： 由 n 个字节组成的序列，索引从 0 到 $n-1$；索引为 $0 \leq k < n$ 的字节要么是指令操作码，要么是立即数操作数。例如，操作码序列 $s = [$0x60、0x01、0x60、0x02、0x01、0x50、0x00$]$ 对应于更易读的程序“PUSH1 0x01；PUSH1 0x02；ADD；POP；STOP”。这里，位于 $s[1]$ 的字节（0x01）是位于 $s[0]$ 的指令（PUSH1）的操作数。

程序计数器： 程序计数器（初始值为 0）标识下一条要执行的指令。例如，如果程序计数器为 4，执行位于 $s[4]$ 的指令（ADD）后，程序计数器加 1，因此 $s[5]$ 是下一条要执行的指令。执行带操作数的指令时（例如位于 $s[0]s[1]$ 的“PUSH1 0x01”），程序计数器会增加 $1+v$，其中 v 是操作数的数量。

栈： 一个由 256 位字组成的栈（初始为空）；指令可以将值压入栈或从栈中弹出。例如，从空栈开始，执行指令“PUSH1 0x1; PUSH1 0x2”会得到栈 $[], $x02，0x01]。从栈 [0x02，0x01] 执行 ADD 指令时，会弹出 2 个操作数，将它们相加并压入结果，从而得到新栈 [0x03 = 0x01 + 0x02]。

内存： 一个可通过 256 位地址寻址、连续排列的字节数组（初始为空）。内存具有易失性，仅在当前程序执行期间可用。当读取或写入某个位置的值时，内存会按需扩展（这会消耗一定的燃料）。可以使用 MLOAD、MSTORE、MLOAD8 或 MSTORE8 指令从内存中读取值或向其中写入值。

存储： 一个从 256 位地址映射到 256 位字的映射，构成合约的永久存储。可以使用 SLOAD 或 SSTORE 指令读取或写入存储。

g即： 为未来的计算留下的燃料。在以太坊虚拟机中，执行一条指令会消耗燃料，从而确保不会发生无限计算。

在以太坊虚拟机中，程序执行可能会在包括以下情况在内的异常情况下中止：

燃料耗尽： 当前状态中剩余的燃料不足以支付执行下一条指令的成本（包括内存扩展的成本，如有）；

堆栈异常： 堆栈大小不能超过 1024。此外，某些指令（例如 POP）只有在栈中有足够元素时才能执行，否则执行应中止。

以太坊虚拟机具有失败状态，用于表示中止的计算。因此，以太坊虚拟机的状态 要么是失败状态，要么是非失败状态。

字节码验证。借助我们的形式语义，我们可以使用 Dafny 验证器来保证字节码程序的安全属性。Dafny 是一种适用于验证的语言，因此可以在代码中加入谓词以及前置条件和后置条件，并由验证器在编译时进行检查。我们利用这一特性来证明字节码的性质。以下简单的 Dafny 程序演示了一个证明：

```
1
方
变
4
6
(y
```

这段简单的代码片段展示了验证过程的几个方面。首先，由于参数 x、y 是 8 位的任意无符号整数，我们可以验证一类程序。这通过创建一个以太坊虚拟机并逐步执行代码来完成，例如使用 ExecuteN 函数。其次，我们使用断言语句（第 7 行）来指定代码的预期属性。断言语句是一种验证语句：与传统编程语言不同，它不会在运行时执行，而是在编译时检查，并且必须对所有输入都成立。对于这个程序 Dafny 可以自动证明断言语句永远不会被违反。该证明使用了 ExecuteN 计算过程中调用的操作码语义。请注意，如果我们将 u8 改为 u256，该属性便不再成立，因为执行 ADD 时可能发生溢出； Dafny 验证器会以“无法证明第 7 行的断言”标记这一问题。另一类自动执行的检查与前置条件和后置条件有关。例如，ADD 指令要求栈中至少有两个元素。这由定义 ADD 语义的函数中的一个前置条件指定。如果上面的代码只有一条 PUSH1 指令， Dafny 就会指出 ADD 无法执行，因为前置条件未得到满足。总的来说，这段简短的代码片段表明，我们可以指定并验证字节码的功能正确性属性；此外，借助用于指定指令语义的前置条件和后置条件，我们可以在运行前检测并修复可能出现的异常（例如栈溢出）。

示例 见清单 A.1 展示了我们如何推理存储更新和异常（中止的计算）。

此合约代码在存储位置 0 维护一个计数器，每次合约调用时该计数器都会递增 1。最初，输入状态中的合约存储不受约束`st` 因此，任何位置都可以包含任意值。合约代码旨在捕获溢出，并在发生溢出时回滚。其目标是：要么合约回滚（检测到溢出），要么计数器递增 1。清单 A.1 给出了这一点的 Dafny 证明。[2] 前置条件（第 10–11 行）确保`st` 是一个执行（未失败）状态，且 程序计数器 `== 0`、栈为空、燃料充足，并且具有待执行的合约代码。

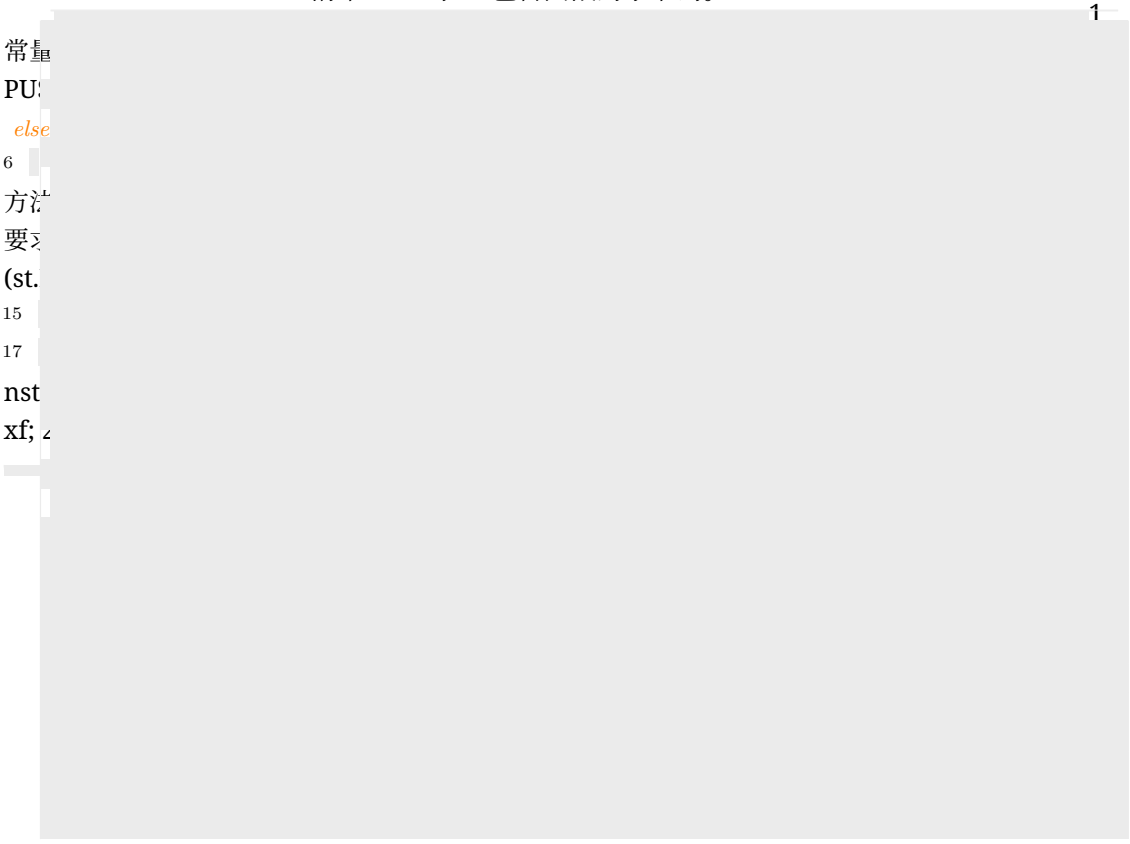
后置条件（第 12-14 行）规定，计算要么递增计数器（位于存储位置 0），要么回滚。证明分为两个关键部分：

1. 执行前 7 条字节码，并将中间状态存储在 nst 中。2. 当加法结果为 0 时，就会发生溢出。因此，我们根据栈顶的结果 nst.Peek(0)，决定其余计算是成功还是回滚。

Dafny 成功验证了这段代码，并保证对于所有满足前置条件（第 10–11 行）的输入状态 st，第 12–14 行中的后置条件均成立。这为字节码提供了强有力的保证：(*i*) 它要么回滚，要么计算递增后的值，但绝不会耗尽燃料，也不会进入无效状态（例如栈溢出或下溢）；(*ii*) 当且仅当存储位置 0 中的初始值严格小于 MAX_U256（第 13 行）时，程序才会正常终止；(*iii*) 正常终止时，存储位置 0 中的值会递增 1（第 14 行）。

[2]论文中的代码为便于理解进行了简化，因此可能无法编译或验证。以下代码 https://github.com/ConsenSys/evm-dafny 可以编译并通过验证。

清单 A.1：验证包含回滚的字节码。



3 Dafny-EVM

我们的以太坊虚拟机以 Dafny 编写，并将语义定义为一个将状态映射到状态的函数。早期做出的一项关键设计决策是，为以太坊虚拟机开发一种函数式纯形式化。我们将在本节介绍 Dafny-EVM 的主要组件，并以一些观察作为总结。

机器状态。 下文的行号均指清单 A.3。以太坊虚拟机的状态是一个记录，包含燃料、程序计数器、栈、代码、内存等多个字段。

每个模块（状态、栈、内存，.....）都提供一个数据类型，并且可能包含一些约束（例如 EvmState.T）。为简洁起见，我们省略了一些字段，这些字段包含有关外围交易以及所谓子状态的信息。State 数据类型（第 7 行）对正常执行（OK）、失败（无效）、返回（返回）、回滚（回退）等情况进行建模。

栈、内存和存储。 我们实现了多个子模块，用于提供对栈、内存和存储的操作。这些内容总结于图 1 中。我们将对栈、内存和存储的操作提升到 状态 数据类型中。在 Dafny 中，这是通过在数据类型定义之后（第 11 行）立即添加函数来完成的。这样可以方便地组合这些函数，并提高可读性。例如，实现操作码 ADD 语义的 Add 函数，使用如下操作序列进行定义：`st.Pop().Pop().Push(...).Next()` 其中 `st` 是一个执行状态（例如 `OK`）。我们使用前置条件（`requires`）来确保提升后的操作仅限于适用状态（通常是执行状态，例如 `OK`），同时确保栈、内存和存储相关函数的前置条件得到满足（例如，对于 `Pop()`，栈大小必须足够）；对于 `Push()`（第 20 行），栈不能已满（堆栈大小上限为 1024）。

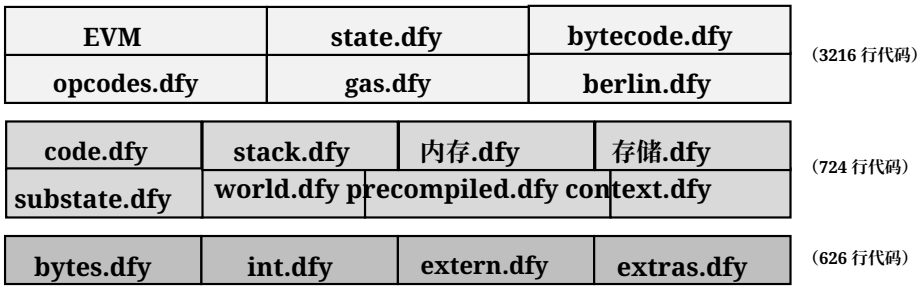


图 1：Dafny-EVM 的源文件。顶部组包含字节码语义和顶层类型。中间组包含主要组件的抽象。底部组是基础原语（例如，用于处理字节和整数）。此处的“Loc”（代码行数）指撰写本文时的行数。

清单 A.2：MLOAD 的语义，Bytecode 模块

```
函
则
//
8
```

在 Dafny 中，验证器会检查前置条件，并且必须能够证明这些条件在每个调用点都成立。请注意， Dafny 会强制执行整数类型的约束，因此每次计算（例如 ADD）并将结果存入一个 256 位字时，我们都必须证明该值小于 2^{256} （以太坊虚拟机规定对此使用模运算）。Dafny 验证器强制执行的前置条件、后置条件和类型检查，有助于确保我们的以太坊虚拟机规范保持一致，且函数定义良好。

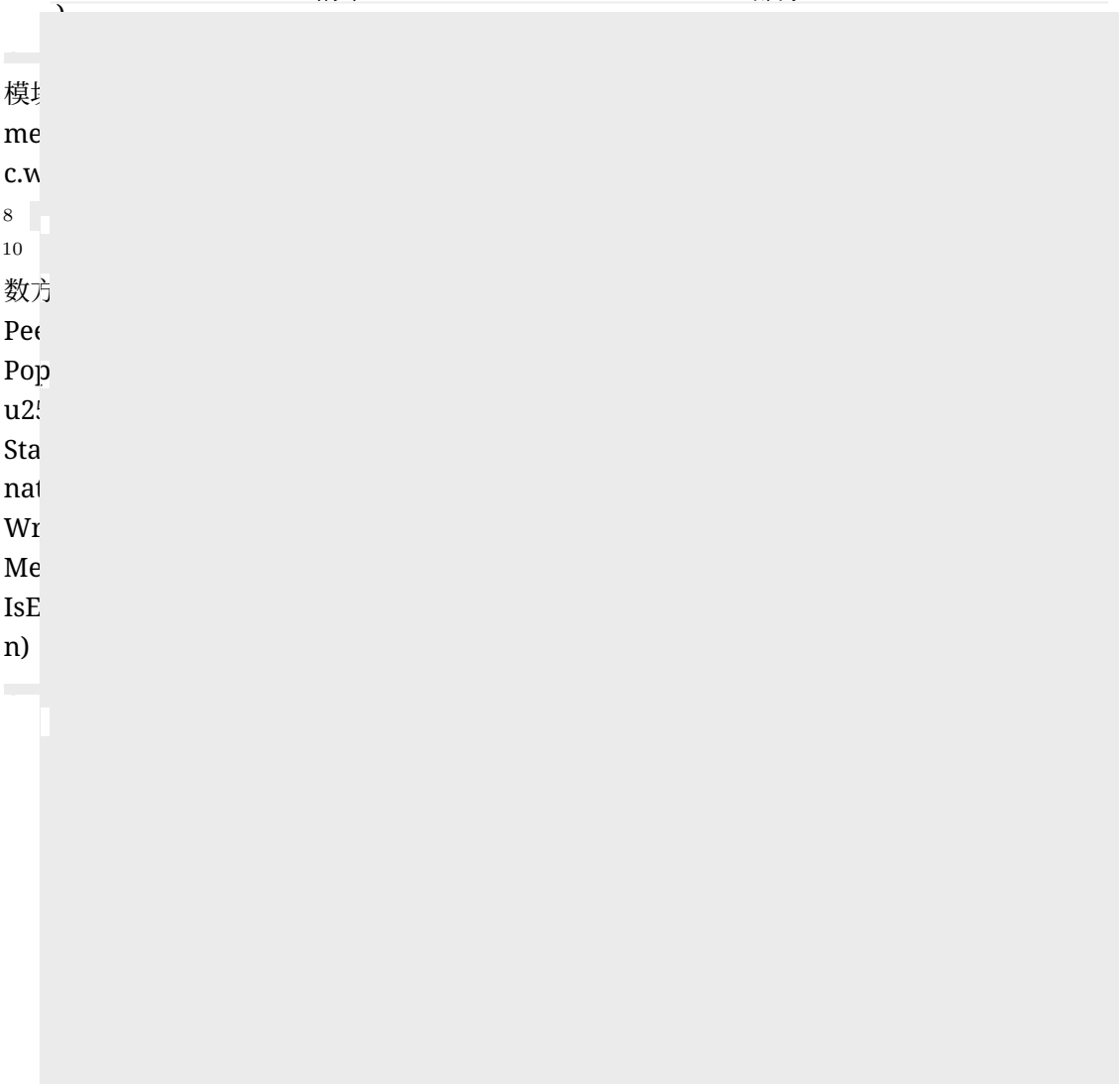
内存操作由 `Memory` 模块提供，其中的各种函数都附属于 `State`，例如 `Read`，`Write` （第 26–28 行）。一个关键观察是，在这两种情况下，地址`addr + 31` 都必须位于已分配内存范围内。这是因为以太坊虚拟机的内存按字节寻址，而我们读取/写入的是`u256` 值（即长度为 32 字节的值）。`MLOAD` 的语义（清单 A.2）凸显了内存操作的复杂性。由于 `Read(loc)` （第 6 行）的前置条件是 `loc + 31 < Memory.Size` （清单 A.3第 26 行），因此该条件必须对状态 `nst`成立。实际上，这是因为调用 `Expand()` （第 5 行）保证了内存充足。如果 MLoad 中的 `Expand()` 调用没有强制执行这一约束，那么 Dafny 就会在 `nst.Read(loc)` 处引发前置条件违例。

燃料。 在我们的设计中，我们选择将燃料计算从指令语义中分离出来。虽然这会带来一些重复，但我们认为这样可以降低认知负荷。尤其是，这种做法避免了在指令语义中穿插燃料计算；而出于性能考虑，实际实现（包括执行规范[3]）通常都会这样做。

合约调用。 各种指令（例如 CALL、DELEGATECALL）使一个合约能够调用另一个合约。这些指令与其他指令不同，因为它们可能涉及在被调用的合约中执行任意 多条指令。我们使用一种类似于延续 的机制实现这一点，但为简洁起见，此处省略细节。

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

清单 g A.3：EvmState 模块 （p部分



观察。 Dafny-EVM 代码为以太坊虚拟机提供了可读且可执行的规范。使用适用于验证的语言有几个好处：使用前置条件和后置条件来编写语义，可提供高度保证；此外，代码可执行，也可以编译为多种目标语言，包括 Java、C# 和 Go。现在，我们根据这个项目的经验，重点介绍一些观察结果。

– **规范**。 Dafny 将 函数调用视为表达式中的已解释，但将方法 调用视为未解释 [5,15]。粗略地说，这意味着验证函数调用时，验证器可以自由访问函数体。相比之下，对于方法调用，验证器只能访问规范中给出的内容（即其前置条件和后置条件）。因此，我们认为方法不适合用于形式化规范（例如以太坊虚拟机的规范）。这是因为我们无法对规范进行比现有程度更进一步的抽象（也就是说，我们无法再指定一个规范）。

– **验证**。函数可以包含限制其输入定义域的前置条件。在Dafny 中，前置条件会在每个调用点强制执行。我们认为，这通过确保函数调用之间的一致性，能够产生更优质的代码。 Dafny 要求每个函数都必须具备终止性证明，从而保证我们的状态转移函数中不存在无限循环

。我们认为，使用不利于验证的语言很难达到这种程度的保证。

– **性能**。由 DAFNY 函数式纯子集生成的代码可能性能不佳，因为为了保持纯性（即引用透明性），需要克隆复合结构（例如映射和数组）。 DAFNY 例如，并未采用克隆消除 [16,26,19] 或可变值语义 [22,21]。鉴于我们的目标是开发形式化规范，而非高效的实现，性能并非关键关注点；在实践中，我们也没有遇到任何重大问题。

在项目过程中，出现了许多问题和挑战。例如， DAFNY 中缺少幂运算符，这意味着对于EXP 字节码，我们必须将其实现为递归函数。一些涉及位和字节的底层操作（例如移位）带来了重大挑战，因为原生`int` 类型不支持位运算符。可以将（例如`u256`）转换为`bv256`）这类位向量类型，后者由 DAFNY 提供并支持按位运算——但这可能导致代码验证出现问题。

4 实践经验

从项目一开始，我们就不确定 DAFNY 是否适合这项规模庞大的形式化任务。不过总体而言，我们很高兴地报告， DAFNY 在很大程度上证明了自身完全能够胜任。当然，过程并非一帆风顺，我们也遇到了几个需要开发技术和/或变通方案来解决的挑战。

代码生成。 DAFNY 可以为多种目标生成代码，包括：C#、Go、Java、C++、Python 和 JavaScript。此外，尽管 DAFNY 本身不支持输入/输出操作 *perse*，但这些操作可以在目标端实现。我们利用这一点，将 DAFNY生成的代码嵌入一个执行输入/输出操作的精简 Java 包装器中，从而能够针对现有实现测试我们的以太坊虚拟机。请注意，生成的代码并未证明 与原始 Dafny 代码等价。由于各种原因（例如团队内部的知识储备），我们选择 Java 作为目标语言，并使用 Gradle 管理构建过程。虽然这一方案已经足够奏效，但仍有几点需要说明：

– **外部函数接口**。由 DAFNY 生成的代码不符合 Java 的风格规范，但除此之外，与其进行接口交互相对容易。 DAFNY 提供了一个运行时库，生成的代码必须针对该库进行编译。该库还提供了替代性的集合实现（例如 DafnySequence、DafnyMap 等）。

– **外部代码**。对于 KECCAK256 的语义以及某些预编译合约，我们更倾向于调用原生 Java 代码（即不在 DAFNY 自身中实现例如 sha256）。然而，尽管 DAFNY 支持 extern 声明，但在撰写本文时，Java 代码生成器对这些声明的支持并不完善。因此，我们不得不提供默认实现（例如返回 0），并采用构建技巧使其正常工作。

– **目标语言的特性**。翻译到目标语言会带来风险。例如，DAFNY 使用欧几里得除法作为其整数除法运算符（即总是向下取整，而不是向零取整），这对粗心者来说是个陷阱。我们也因此偶然发现 Java 代码生成器中的一个错误：有时会采用标准除法。[4] 我们还发现，将 DAFNY 集合（例如`seq<u8>`）翻译为 Java 时存在不健全之处[5], ，而 C# 中数据类型的实现也有错误。 [6]

[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

验证与测试。为求完整，我们为形式体系的各个组件开发了许多单元测试。以太坊通用测试还提供了数万个测试，用于确保 EVM 兼容性。[7] 因此，我们一直使用这些测试，将我们的形式化结果与现有实现进行比对。这需要从我们的规范生成可执行代码，而这一过程带来了若干挑战（篇幅所限，遗憾的是此处不作讨论）。截至撰写本文时，我们从 13K 个通用测试（柏林硬分叉）中选取了约 7500 个有代表性的测试，其中 6900 个通过（92%）。143 个未通过的测试中，大多数（100 个）未通过的原因是：部分预编译合约尚未完全实现（44 个）；我们目前不会检查跳转到指令操作数的情况（56 个）。其余约 450 个测试因各种原因被跳过，例如超时或导致测试系统中断。最后需要指出的是，在拉取请求合并之前，我们会将所有测试作为持续集成的一部分运行。

5 相关工作

最初对以太坊虚拟机进行形式化规范的尝试可归功于 Hirai [13]，他在程序开发环境 Lem [18] 中对以太坊虚拟机进行了形式化。[13] 中的形式化仅限于单个合约的执行，且字节码证明在自动化方面受到限制。后来，Amani 等人 [3] 在 Hirai 的形式化工作基础上，提出了 Isabelle/HOL 形式化。他们的工作引入了程序逻辑，用于推理字节码（仅支持 142 条以太坊虚拟机指令中的 36 条），但他们依赖构建控制流图来定义程序语义。字节码推理仅限于线性指令序列（块），且未实现完全自动化。[9] 中还开发了另一种 Isabelle/HOL 规范，专门用于 Gas 消耗分析和证明字节码的终止性。

最近，Grishchenko 等人 [10] 提出了以太坊虚拟机的一种部分形式化（并非支持所有操作码，且燃料成本语义并不完整），$F^\star$ 的目标是验证安全属性。

最先进的形式化工作或许是 KEVM [12]，它使用 $\mathbb{K}$ 框架 [23]。它为 EVM 字节码的语法和语义提供了形式化且可执行的规范。借助 $\mathbb{K}$ 框架内置的自动化工具，可以生成解释器、编译器、调试器，在一定程度上还可以生成验证器，用于检查某些合约的字节码 [20]。$\mathbb{K}$ 框架的默认输入格式（KEVM 使用的格式）基于 XML，可能不太便于开发者使用。类似地，IELE [24] 旨在设计一种比 EVM 字节码更易读的语言，并将其作为包括 Solidity、Vyper 和 Plutus 在内的高级语言的目标语言。IELE 使用 $\mathbb{K}$ 框架定义，并以 LLVM 工具（编译器）作为后端。

EVM 有多种使用不同语言实现的版本，也有不同的客户端，例如 Geth[8]、Besu[9]，，以及较新的 *Python* 中的执行规范 [3]。Geth 和 Besu 的实现分别使用 Go 和 Java 编写，无法用于推理字节码。Python 实现依赖于 Python 特有的命令式语言特性（可变性、异常），既未提供指令语义的函数式定义，也未明确规范异常情况：例如，Python 代码没有提供前置条件或对异常的显式处理，而且异常可能在调用栈深处发生，从而影响可读性。

有若干工具 Oyente [4], EtherIR [1], eThor [25], Rattle [27], 和 Certora [14] 可用于对 EVM 字节码执行静态分析。此外，还有专门用于分析 Gas 消耗的扩展工具，例如 GASTAP [2], GasReducer [8]。这些工具会构建字节码的抽象表示，但目前尚不清楚该抽象是否保持语义。

与上述引用的形式化、实现和工具不同，我们的形式语义与语言无关（将状态转移函数定义为函数），易于阅读且对开发者友好，并为指令语义提供数学化且经过验证的前置条件和后置条件

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

此外，我们的语义可用于使用标准不变量对字节码进行演绎推理，包括分析 Gas 消耗。

6 结论

我们提出了一种以太坊虚拟机的形式语义，并使用 Dafny 的纯函数式子集实现。我们的语义易于阅读，经过机器验证 且可执行，为对字节码进行形式化推理提供了可靠的框架。

这为若干直接应用打开了大门：[10]

– 完整的智能合约验证：在实践中，这可能是一项代价高昂的工作，并且可能需要专门的验证技能，或熟悉 Dafny。– 编译器优化的正确性：如今可以验证多种 Gas 优化模式，例如，将一串操作码 SWAP1 POP POP 优化为 POP POP。– 下溢/溢出检测的正确性：要检测模运算 ADD(x, y) 中的溢出，通常先计算结果 r = ADD(x, y)，然后检查 r >= x。我们可以形式化地证明这种方法是可靠的。– 合成经过验证的字节码：我们设计了一种方法 [6] ，可直接在 Dafny 中对智能合约进行规约和验证。我们正在探索细化证明技术 ，以便从合约经过验证的 Dafny 代码中合成字节码。最终，我们可能会开发出一种 Dafny-到以太坊虚拟机的经验证的编译器。

尽管我们的方法在形式化方法社区中的优势显而易见，但在以太坊生态系统中采用这些技术仍然面临挑战。虽然使用 Solidity 编写合约、使用 Python 编写规范等成熟技术或许值得商榷，[11],但它们在以太坊社区中仍然十分普遍。要让我们的方法得到广泛采用，主要可能需要克服两方面的障碍：(*i*) 提供便于开发者编写合约的工具； Dafny 围绕这些工具的支持（例如验证性能改进、反例生成 [7], 以及 VSCode 集成）已部分解决了这一问题；以及(*ii*) 教育以太坊社区，使其了解形式化验证对以太坊生态系统的长期益处。

参考文献

1. Albert, E., Gordillo, P., Livshits, B., Rubio, A., Sergey, I.：Ethir：以太坊字节码高级分析框架。载于：Lahiri, S.K., Wang, C.（编）《自动化验证与分析技术》，第513–520页。Springer International Publishing，Cham（2018）2. Albert, E., Gordillo, P., Rubio, A., Sergey, I.：在燃料耗尽的边缘运行——使用静态资源分析防止以太坊智能合约中的燃料耗尽漏洞。载于：Ganty, P., Kaˆaniche, M.（编）《计算机与通信系统验证与评估——第13届国际会议，VECoS 2019，葡萄牙波尔图，2019年10月9日，会议录》。《计算机科学讲义》，第11847卷，第63–78页。Springer（2019）。https://doi.org/10.1007/978-3-030-35092-5 53. Amani, S., B´egel, M., Bortin, M., Staples, M.：在Isabelle/HOL中验证以太坊智能合约字节码。载于：Andronick, J., Felty, A.P.（编）《第7届ACM SIGPLAN认证程序与证明国际会议论文集》，CPP 2018，美国加利福尼亚州洛杉矶，2018年1月8日至9日，第66–77页。ACM（2018）。https://doi.org/10.1145/31670844. Badruddoja, S., Dantu, R., He, Y., Upadhayay, K., Thompson, M.：让智能合约变得更智能。载于：2021 IEEE区块链与加密货币国际会议（ICBC），第1–3页（2021）。https://doi.org/10.1109/ICBC51069.2021.94611485. Bradley, A.R., Manna, Z.：计算演算——用于验证的判定过程。Springer（2007）

[10]示例见https://github.com/ConsenSys/evm-dafny。

6. Cassez, F., Fuller, J., Anton Quiles, H.M.：使用Dafny对智能合约进行演绎验证。载于：Groote, J.F., Huisman, M.（编）《工业关键系统形式化方法——第27届国际会议，FMICS 2022，波兰华沙，2022年9月14日至15日，会议录》。《计算机科学讲义》，第13487卷，第50–66页。Springer（2022）。https://doi.org/10.1007/978-3-031-15008- 1 5 7. Chakarov, A., Fedchin, A., Rakamaric, Z., Rungta, N.：更好的Dafny反例。载于：Fisman, D., Rosu, G.（编）《系统构造与分析工具和算法——第28届国际会议，TACAS 2022，作为欧洲软件理论与实践联合会议ETAPS 2022的一部分举行，德国慕尼黑，2022年4月2日至7日，会议录，第I部分》。《计算机科学讲义》，第13243卷，第404–411页。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.：在使用智能合约时节省资金。载于：Zisman, A., Apel, S.（编）《第40届国际软件工程会议：新思想与新兴成果论文集》，ICSE (NIER) 2018，瑞典哥德堡，2018年5月27日至6月3日 3183420 9 81 84 2018 10 1145 3183399。第–页。ACM（）。https://doi.org/ . / . . Genet, T., Jensen, T.P., Sauvage, J.：以太坊智能合约的终止性。载于：Samarati, P., di Vimercati, S.D.C., Obaidat, M.S., Ben-Othman, J.（编）《第17届电子商务与电信联合国际会议论文集》，ICETE 2020——第2卷：SECRYPT，法国利尤桑、巴黎，2020年7月8日至10日。第39–51页。ScitePress（2020）。https://doi.org/10 0009564100390051 10 5220. / . Grishchenko, I., Maffei, M., Schneidewind, C.：以太坊智能合约安全分析的语义框架。载于：Bauer, L., K¨usters, R.（编）《安全与信任原理——第7届国际会议，POST 2018，作为欧洲软件理论与实践联合会议ETAPS 2018的一部分举行，希腊塞萨洛尼基，2018年4月14日至20日，修订精选论文集》。《计算机科学讲义》，第10804卷，第243–269页。Springer（2018）。https://doi.org/10.1007/978-3-319-89722-6 10 11. Guido, D.：《第6集：区块链人到底在做什么，为什么这不是一场垃圾火灾？》（2021），https://galois.com/blog/2020/11/introducing-the-building-better-systems-podcast/，《构建更好的系统》（播客），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：以太坊虚拟机的完整形式语义。载于：第31届IEEE计算机安全基础研讨会，CSF 2018，英国牛津，2018年7月9日至12日，第204–217页。IEEE计算机学会（2018）。https://doi.org/10 00022 13 1109 2018。/CSF。。Hirai, Y.：为交互式定理证明器定义以太坊虚拟机。载于：Brenner, M., Rohloff, K., Bonneau, J., Miller, A., Ryan, P.Y.A., Teague, V., Bracciali, A., Sala, M., Pintore, F., Jakobsson, M.（编）《金融密码学与数据安全——FC 2017国际研讨会，WAHC、BITCOIN、VOTING、WTSC和TA，马耳他斯利马，2017年4月7日，修订精选论文集》。《计算机科学讲义》，第10323卷，第520–535页。Springer（2017）。https://doi.org/10 0 33 14 1007 978 3 319 70278。/ - - - -。Jackson, D., Nandi, C., Sagiv, M.：Certora技术白皮书。Medium博文（2022），https://medium.com/certora/certora-technology-white-paper-cae5 1 15 0ab bdf。Kroening, D., Strichman, O.：《判定过程——算法视角，第二版》。Springer-Verlag（2016）15. Lameed, N., Hendren, L.J.：在MATLAB即时编译器中高效实现数组复制语义的分阶段静态技术。载于：《编译器构造会议（CC）论文集》，第22–41页（2011）16. Leroy, X.：形式化验证的编译器后端。《自动化推理杂志》43(4)，第363–446页（2009）。https://doi.org/10 4 18 1007 10817 009 9155。/s - - -。Mulligan, D.P., Owens, S., Gray, K.E., Ridge, T., Sewell, P.：Lem：现实世界语义的可复用工程。载于：Jeuring, J., Chakravarty, M.M.T.（编）《第19届ACM SIGPLAN函数式编程国际会议论文集》，瑞典哥德堡，2014年9月1日至3日，第175 2628143 19 188 2014 10 1145 2628136–页。ACM（）。https://doi.org/ . / . . Odersky, M.：如何让破坏性更新更少具有破坏性。载于：《ACM编程语言原理研讨会（POPL）论文集》，第25–36页（1991）20. Park, D., Zhang, Y., Rosu, G.：以太坊2.0存款智能合约的端到端形式化验证。载于：Lahiri, S.K., Wang, C.（编）《计算机辅助验证——第32届国际会议，CAV 2020，美国加利福尼亚州洛杉矶，2020年7月21日至24日，会议录，第I部分》。《计算机科学讲义》，第12224卷，第151–164页。Springer（2020）。https://doi.org/10.1007/978-3-030-53288-8 8

21. Pearce, D.J., Groves, L.：设计验证编译器：开发 Whiley 的经验教训。《计算机编程科学》第 191–220 页（2015） 22. Racordon, D., Shabalin, D., Zheng, D., Abrahams, D., Saeta, B.：可变值语义的实现策略。《对象技术期刊》**21**（2）（2022） 23. Rosu, G： 𝕂：编程语言与形式化分析工具的语义框架。载于：Pretschner, A., Peled, D., Hutzelmann, T.（编），《可信软件系统工程》，北约和平与安全科学系列 D辑：信息与通信安全，第 50 卷，第 186186 24206201710323397816149981052022– . IOS出版社（）。https://doi.org/./-----. Runtime Verification：iele虚拟机。博文（），https://runtimeverification.com/the- iele-virtual-machine/ 25. Schneidewind, C., Grishchenko, I., Scherer, M., Maffei, M.：ethor：以太坊智能合约的实用且可证明健全的静态分析。载于：Ligatti, J., Ou, X., Katz, J., Vigna, G.（编），CCS '20：2020年ACM SIGSAC计算机与通信安全会议，线上会议，美国，2020年11月93417250 2613, 20206216402020101145337229 7- . 第 – 页。ACM（）。https://doi.org/./.. Shankar, N.：函数式语言中安全破坏性更新的静态分析。载于：基于逻辑的程序综合与转换研讨会（LOPSTR）论文集。第 1–24 页（2001） 27. Trail of Bits：Rattle——以太坊EVM二进制分析框架。Medium博文（2018），https: //blog.trailofbits.com/2018/09/06/rattle-an-ethereum-evm-binary-analysis-framework/ 28. Wood, G.：以太坊：安全的去中心化通用化交易账本。以太坊项目黄皮书（2022），https://ethereum.github.io/yellowpaper/paper.pdf，柏林版本 d77a387 - 2022年4月26日