# Formal Verification of Ethereum Smart Contracts Using Isabelle/HOL

Maria Ribeiro[1], Pedro Adão[1,2(✉)], and Paulo Mateus[1,2]

[1] Instituto Superior Técnico, Universidade de Lisboa, Lisbon, Portugal
{maria.ribeiro,pedro.adao}@tecnico.ulisboa.pt,
pmat@math.tecnico.ulisboa.pt
[2] Instituto de Telecomunicações, Lisbon, Portugal

**Abstract.** The concept of blockchain was developed with the purpose of decentralizing the trade of assets, suppressing the need for intermediaries during this process, as well as achieving a digital trust between parties. A blockchain consists in a public immutable ledger, constituted by chronologically ordered blocks such that each block contains records of a finite number of transactions.

The Ethereum platform, that this paper builds upon, is implemented using a blockchain architecture and introduces the possibility of storing Turing complete programs. These programs, also known as smart contracts, can then be executed using the Ethereum Virtual Machine. Despite its core language being the EVM bytecode, they can also be implemented using a higher-level language that is later compiled to EVM, being Solidity the most used. Among its applications stand out decentralized information storage, tokenization of assets, and digital identity verification.

In this paper we propose a method for formal verification of Solidity smart contracts in Isabelle/HOL. We start from the imperative language and big-step semantics proposed by Schirmer [23], and adapt it to describe a rich subset of Solidity, implementing it using the Isabelle/HOL proof assistant. Then, we describe the properties about programs using Hoare logic, and present a proof system for the language, for which results on soundness and (relative) completeness are obtained.

Finally, we describe the verification of an electronic voting smart contract, which illustrates the degree of proof complexity that can be achieved using this method. Examples of smart contracts containing overflow and reentrancy vulnerabilities are also presented.

**Keywords:** Formal verification · Isabelle/HOL · Hoare logic · Smart contracts · Solidity · Ethereum

Partially supported by Programa Operacional Competitividade e Internacionalização (COMPETE 2020), Fundo Europeu de Desenvolvimento Regional (FEDER) through Programa Operacional Regional de Lisboa (Lisboa 2020), Project BLOCH - LISBOA-01-0247-FEDER-033823, and Fundação para a Ciência e Tecnologia (FCT) project UID/EEA/50008/2019.


V. Nigam et al. (Eds.): Scedrov Festschrift, LNCS 12300, pp. 71–97, 2020.
https://doi.org/10.1007/978-3-030-62077-6_7

## 1 Introduction

The emergence of the blockchain concept was associated with the appearance of Bitcoin, one of the first decentralized cryptocurrencies, introduced in 2008 by Satoshi Nakamoto [21]. A cryptocurrency is independent of any central administrative entities and uses instead a *peer-to-peer* digital system, managed by a network of nodes. Transactions are stored in a blockchain, an append-only public ledger, through the process of *mining*. Nodes in the network, also known as *miners*, try to solve a difficult computational problem called *proof-of-work*. When a transaction is verified by the network it is incorporated into the blockchain using a cryptographic hash function, which includes data from the previous block's hash and makes the whole chain cryptographically secure and, therefore, immutable.

Our work focuses on the Ethereum platform, proposed by Vitalik Buterin [4] in 2013, which similarly uses a blockchain architecture but also introduces the feature of storing Turing complete programs, known as *smart contracts*. These programs can be executed by the stack-based Ethereum Virtual Machine (EVM), and its formalization was first approached by Gavin Wood [26]. Ethereum also introduces the concept of *gas*, as each operation in the virtual machine has an associated cost in *ether*, the Ethereum currency. When a contract is executed, either by being called by a transaction or by code in another contract, the original transaction initiator needs to pay for the total cost of operations.

Given the valuable assets in these contracts, and the fact that they are immutable, studying the security of these programs becomes of uttermost importance. With that in mind, the main goal of this work is to introduce a formal verification method of Ethereum smart contracts using Isabelle, a higher order logic (HOL) theorem prover. We have chosen to verify smart contracts written in Solidity, a higher level language that compiles to EVM bytecode.

The main reference for our language, and respective semantics and proof system, is the work by Schirmer [23]. We adapt the proposed language for sequential programs to capture a relevant subset of Solidity. Our main additions were the modeling of Solidity calls, both internal and external, Solidity exceptions, and reverting all state modifications. To formalize the meaning of these new operations in terms of execution, the big-step semantics was extended. The verification of programs is done using Hoare logic. Soundness and (relative) completeness results for the proof system are presented.

The concept of weakest precondition [6] is presented and used both for optimizing program verification and for the completeness result. Regarding the first, and following the work by Frade and Pinto [7], we enhance the weakest precondition and verification condition computations with the cases for *Dyncom, Require* and *Init*. The proof of (relative) completeness, based on the proof by Winskel [25], is extended with the *Call, Handle, Revert, Dyncom, Require* and *Init* cases.

To conclude the paper we present some relevant examples of applications such as electronic voting, tokens, and reentrancy, describing and analyzing this way the expressiveness of the language.

**Related Work.** Previous efforts have been made by the research community to formally verify smart contracts. Hirai formalized the EVM semantics in Lem and used Isabelle/HOL to prove safety properties of Ethereum smart contracts [11]. Amani et al. [1] formalized the EVM semantics in Isabelle/HOL and proposed a sound program logic to verify correctness of smart contracts. Grishchenko et al. [9] formalized a complete small-step semantics of EVM bytecode in F*, and defined security properties for smart contracts such as call integrity and atomicity. Hildebrandt et al. [10] formalized the EVM semantics in the $\mathbb{K}$ framework. Bhargavan et al. [3] introduced a framework that translates smart contracts from Solidity to F*, allowing verification of functional correctness and safety, as well as a decompilation from EVM bytecode to F* for analysis of low-level properties.

As for analysis of Solidity code, Bartoletti et al. [2] proposed a calculus for a fragment of Solidity with a single primitive to transfer currency and invoke contract procedures, and Jiao et al. [14] developed a formal semantics for Solidity in the $\mathbb{K}$ framework that allows formal reasoning about high-level contracts. Zakrzewski [27] proposed a semantics for a small fragment of Solidity in Coq.

Some automatic analysis tools for analysing Ethereum smart contracts have also been developed as are the cases of Oyente [17], Maian [22], Mythril [20], and Securify [24]. A survey on these techniques and tools can be found in [8].

**Andre's Influence in This Work.** Scedrov's results on linear logic [15,16] significantly shaped our work. Linear logic encompasses the dynamics of algorithms and resources, and its main impacts have been in computer science rather than traditional mathematics. Linear logic significantly influenced the design of Hoare triples, which are the basis of this work. Moreover, the use of formal methods in Scedrov's work [18,19], namely on process algebras, has also been a significant contribution to the security area in general and inspired this work in particular. Indeed, this paper's primary goal is to present a proof-based method to derive security properties in an imperative language for contracts over a blockchain, which is a very restrictive form of concurrent programming, and for which we do not impose polynomial-time bounding. More importantly, Andre directly impacted the work and scientific career of two of the authors. Pedro Adão and Paulo Mateus were respectively PhD and Postdoc students of Andre.

## 2 The Ethereum Blockchain

Ethereum can be seen as a decentralized computing platform since it uses a blockchain architecture and introduces the feature of storing *smart contracts*.

In this section we present a simplified definition of the Ethereum blockchain. In the following definitions let $\mathbb{N}_x$ the set of non-negative integers with size up to $x$ bits and $\mathbb{B}$ the set of bytes. An account is an object of the Ethereum environment that is identified by a 160-bit string known as the account's address.

**Definition 1 (World state).** *The world state is a mapping $\sigma$ between addresses (160 bit strings) and account states.*

$$\sigma : \{0,1\}^{160} \rightarrow \mathbb{N}_{256} \times \mathbb{N}_{256} \times (\{0,1\}^{256} \rightarrow \{0,1\}^{256}) \times \mathbb{B}^*$$

There are two types of accounts: *externally owned accounts (EOA)* and accounts associated with code (*contract accounts*).

**Definition 2 (Account state).** *Given an address $a$, the account state $\sigma(a)$ is a tuple $\mathcal{A} = \langle nonce, balance, storage, code \rangle$, where*

- ***nonce*** $\in \mathbb{N}_{256}$ *is the nonce of the account. If $a$ is the address of an EOA, corresponds to the number of transactions sent from this address. If $a$ is the address of a contract account, corresponds to the number of contract-creations made by this account;*
- ***balance*** $\in \mathbb{N}_{256}$ *is the value of ether owned by account $a$;*
- ***storage*** *is a mapping between 256-bit values and corresponds to the account's storage;*
- ***code*** $\in \mathbb{B}^*$ *is the EVM code of this account. In case of an EOA corresponds to the empty string.*

There are two types of transactions: *contract creation transactions* and *transactions which result in message calls*. A transaction is triggered by an external actor.

**Definition 3 (Transaction).** *A transaction is a tuple $T = \langle nonce, gasprice, gaslimit, from, to, value, init/data \rangle$, where*

- ***nonce*** $\in \mathbb{N}_{256}$ *is the number of transactions sent by address* from*;*
- ***gasprice*** $\in \mathbb{N}_{256}$ *is equal to the cost per unit of gas, in ether, for all computation costs of this transaction;*
- ***gaslimit*** $\in \mathbb{N}_{256}$ *is equal to the maximum amount of gas that should be used in the execution of this transaction;*
- ***from*** $\in \{0,1\}^{160}$ *is the address of the transaction's sender;*
- ***to*** $\in \{0,1\}^{160}$ *is the address of the transaction's recipient;*
- ***value*** $\in \mathbb{N}_{256}$ *is the value of ether to be transferred to the message call's recipient or, in the case of contract creation, as an endowment to the newly created account.*

*Additionally, in the case of a contract creation transaction*

- ***init*** *is the EVM code for the account initialization procedure;*

*In the case of a message call*

- ***data*** *is the input data of the message call.*

A *message call* is an internal concept which consists of data (a set of bytes) and value (specified as ether) passed from one account to another. It may be triggered by a transaction, where the sender is an EOA, or by EVM code, where the sender is a contract account.

Transactions are grouped and stored in finite blocks.

**Definition 4 (Block).** *A block $B$ is a package of data constituted by*

- *a header, constituted by the block's number, timestamp, nonce, difficulty, beneficiary, state and hash of its parent's block header;*
- *a list of transactions $\mathcal{T} = \{T_1, \ldots, T_m\}$.*

The block's difficulty influences the time that it takes to find a valid nonce for the block and thus solving the proof-of-work mining problem. The beneficiary is the address which receives all the fees from the successful mining of this block. The fact that the hash of this block's header includes its parent's hash, is essential to the blockchain's immutability. The stored state corresponds to the one after all transactions are executed.

Ethereum can be seen as a transaction-based state machine. In such a representation a transaction represents a valid transition between two states $\sigma_t$ and $\sigma_{t+1}$. Since transactions are grouped in finite blocks, a block may also represent a state transition $\sigma'_t$ and $\sigma'_{t+1}$. These transitions between blocks introduce the concept of a chain of blocks, a blockchain.

**Definition 5 (Blockchain).** *A blockchain is defined as an ordered sequence of blocks $\mathcal{B} = \{B_0, B_1, \ldots\}$.*

In this paper we present an approach to the formal verification of Solidity smart contracts. Regarding code structure, a Solidity contract consists, as it follows a object-oriented structure, of a set of *state variables* which are part of the account's storage, and a set of *function declarations*. Functions in a contract can introduce local variables, which are stored in the memory. Solidity also contains a set of globally available variables that can be accessed regarding the current block, transaction, message call and address.

A function can be called by an external user, an EOA, which initiates a transaction, or by another contract. This happens when a called contract contains code that calls another contract, generating a new message call. A function call can be internal or external and an external call can be a *regular call* or a *delegate call*, in which case the code is executed in the context of the calling contract. Details about these methods and respective implementation are presented in Sect. 3.4.

Every contract has a *fallback function*, which is automatically executed whenever a call is made to the contract and none of its other functions match the given function identifier, or in the cases where no data is supplied.

Solidity also allows the usage of *exceptions*. Whenever an exception is thrown, all state changes are reverted. Our approach for modeling exceptions and *state reversion* is described in Sect. 3.4.

## 3 The SOLI Language

In this section we define the core elements of the language and introduce a set of big-step execution rules to describe their semantics. The main reference for our language and respective semantics is the work by Schirmer [23] which we adapt and extend to capture a relevant fragment of Solidity.

### 3.1 Syntax

The syntax of our language is a combination of deep and shallow embeddings. Commands are represented by an inductive, state dependent, datatype whereas some other syntactic elements are defined as abbreviations of their semantics. Boolean expressions, *bexp*, and assertions, *assn*, are defined as state sets.

**Definition 6 (Syntax).** *Let 's be the state space type. The syntax for* boolean expressions and assertions *is defined by the types 's bexp and 's assn, respectively. The syntax for* commands *is defined by the polymorphic datatype 's com, where 's ⇒'s is a state-update function and fname the type of function names.*

$$
\begin{array}{lcl}
's\ bexp & := & 's\ set \\
's\ assn & := & 's\ set \\
's\ com & := & \textbf{Skip} \mid \textbf{Upd}\ 's \Rightarrow 's \mid \textbf{Seq}\ 's\ com\ 's\ com \mid \\
 & & \textbf{If}\ 's\ bexp\ 's\ com\ 's\ com \mid \textbf{While}\ 's\ bexp\ 's\ com \mid \\
 & & \textbf{Dyncom}\ 's \Rightarrow 's\ com \mid \textbf{Call}\ fname \mid \textbf{Revert} \mid \\
 & & \textbf{Handle}\ 's\ com\ 's\ com \mid \textbf{Require}\ 's\ bexp \mid \\
 & & \textbf{Init}\ 's\ com\ 's \Rightarrow 's \Rightarrow 's
\end{array}
$$

Regarding the definition of commands, **Upd** is used to model assignments by executing a state-update function $'s \Rightarrow' s$. Conditional statements and while loops are defined with the usual syntax. The **Skip** command, which does nothing, is also defined.

In order to allow complex operations such as calling other functions and reverting all state changes, the following commands are introduced in `SOLI`. **Dyncom** is a command which receives a state and allows to write statements which are state dependent. This is useful when referring to states in different steps of execution. A general **Call** is introduced, which receives a function name. It corresponds to the simplest form of calling a procedure. The different types of procedure calls and respective execution details are described in detail in Sect. 3.4. **Revert** throws a revert type exception which signals that the state must be reverted, and **Handle** is an auxiliary command to handle state reversion if signaled. **Require** models Solidity exceptions, and **Init** models state reversion whenever a `REVERT` exception is thrown. Both commands are detailed in Sect. 3.4.

### 3.2 Concrete Syntax

To improve the readability of `SOLI` programs, some syntax translations are introduced. $\{|b|\}$ is defined as the set of states for which the predicate $b$ holds. Syntax translations are defined as follows, where $c_1$ and $c_2$ are commands, $b$ a boolean and $s$ a state.

$$
\begin{aligned}
'x \ ::= \ a &\rightharpoonup \textbf{Upd}\ (\lambda s.\ s(\!|x := a|\!)) \\
c_1;;c_2 &\rightharpoonup \textbf{Seq}\ c_1\ c_2 \\
\textbf{IF}\ b\ \textbf{THEN}\ c_1\ \textbf{ELSE}\ c_2 &\rightharpoonup \textbf{If}\ \{|b|\}\ c_1\ c_2 \\
\textbf{IF}\ b\ \textbf{THEN}\ c_1 &\rightharpoonup \textbf{IF}\ b\ \textbf{THEN}\ c_1\ \textbf{ELSE Skip} \\
\textbf{WHILE}\ b\ \textbf{DO}\ c &\rightharpoonup \textbf{While}\ \{|b|\}\ c \\
\textbf{REQUIRE}\ b &\rightharpoonup \textbf{Require}\ \{|b|\}
\end{aligned}
$$

### 3.3 Semantics

To model big-step semantics, the state space $'s$ is augmented, as described by the datatype $'s\ state$, with information about whether exceptions were thrown.

$$'s\ state \ := \ \text{Normal}\ 's \ \mid \ \text{Rev}\ 's$$

To formalize the execution relation, the partial function $\Gamma$ is introduced, which maps function names to the corresponding bodies.

In Isabelle such a function is defined as $'b \Rightarrow 'a\ option$ where $'a\ option = Some\ 'a \mid None$. In the case of $\Gamma$ being defined for $m$, it is selected using $the\ (Some\ m) = m$.

**Definition 7 (Big-step semantics).** *The* big-step semantics *for* `SOLI` *is based on a deterministic evaluation relation formalized by the predicate*

$$\Gamma \vdash \langle c,\ s\rangle \Rightarrow t$$

*where*

$$
\begin{aligned}
\Gamma &:: fname \rightharpoonup 's\ com \\
c &:: 's\ com \\
s,\ t &:: 's\ state
\end{aligned}
$$

*and evaluated accordingly to the set of rules represented in Fig. 1. The meaning for this predicate is as expected, that is, if command c is executed in initial state s, then the execution terminates in state t.*

### 3.4 Additional Language Features

Gas isn't modeled in `SOLI`, the main reason is because it expresses a high level language and would not be accurate to measure gas consumption since it is defined for each opcode. Also the goal is to verify properties which are expressed in a symbolic way, and in most of the cases this measure is not relevant. One could however estimate bounds for `SOLI` commands, with the help of some side tools such as the Remix compiler, and defining the consumption inductively.

$$\frac{}{\Gamma \vdash \langle \textbf{Skip},\ \text{Normal } s\rangle \Rightarrow \text{Normal } s}\ (\textit{Skip}) \qquad \frac{}{\Gamma \vdash \langle \textbf{Upd } f,\ \text{Normal } s\rangle \Rightarrow \text{Normal } (f\ s)}\ (\textit{Upd})$$

$$\frac{\Gamma \vdash \langle c_1,\ \text{Normal } s_1\rangle \Rightarrow s_2 \qquad \Gamma \vdash \langle c_2,\ s_2\rangle \Rightarrow s_3}{\Gamma \vdash \langle \textbf{Seq } c_1\ c_2,\ \text{Normal } s_1\rangle \Rightarrow s_3}\ (\textit{Seq})$$

$$\frac{s \in b \qquad \Gamma \vdash \langle c1, \text{Normal } s\rangle \Rightarrow t}{\Gamma \vdash \langle \textbf{If } b\ c1\ c2,\ \text{Normal } s\rangle \Rightarrow t}\ (\textit{IfTrue}) \qquad \frac{s \notin b \qquad \Gamma \vdash \langle c_2,\ \text{Normal } s\rangle \Rightarrow t}{\Gamma \vdash \langle \textbf{If } b\ c_1\ c_2,\ \text{Normal } s\rangle \Rightarrow t}\ (\textit{IfFalse})$$

$$\frac{s_1 \in b \qquad \Gamma \vdash \langle c,\ s_1\rangle \Rightarrow s_2 \qquad \Gamma \vdash \langle \textbf{While } b\ c,\ s_2\rangle \Rightarrow s_3}{\Gamma \vdash \langle \textbf{While } b\ c,\ \text{Normal } s_1\rangle \Rightarrow s_3}\ (\textit{WhileTrue})$$

$$\frac{s \notin b}{\Gamma \vdash \langle \textbf{While } b\ c,\ \text{Normal } s\rangle \Rightarrow \text{Normal } s}\ (\textit{WhileFalse})$$

$$\frac{\Gamma \vdash \langle c\ s,\ \text{Normal } s\rangle \Rightarrow t}{\Gamma \vdash \langle \textbf{DynCom } c,\ \text{Normal } s\rangle \Rightarrow t}\ (\textit{DynCom}) \qquad \frac{\Gamma \vdash \langle the\ (\Gamma\ f),\ \text{Normal } s\rangle \Rightarrow t}{\Gamma \vdash \langle \textbf{Call } f,\ \text{Normal } s\rangle \Rightarrow t}\ (\textit{Call})$$

$$\frac{\Gamma \vdash \langle c_1, \text{Normal } s\rangle \Rightarrow \text{Normal } t}{\Gamma \vdash \langle \textbf{Handle } c_1\ c_2, \text{Normal } s\rangle \Rightarrow \text{Normal } t}\ (\textit{HandleNormal})$$

$$\frac{\Gamma \vdash \langle c_1, \text{Normal } s\rangle \Rightarrow \text{Rev } r \qquad \Gamma \vdash \langle c_2,\ \text{Normal } r\rangle \Rightarrow t}{\Gamma \vdash \langle \textbf{Handle } c_1\ c_2,\ \text{Normal } s\rangle \Rightarrow t}\ (\textit{HandleRevert})$$

$$\frac{}{\Gamma \vdash \langle \textbf{Revert},\ \text{Normal } s\rangle \Rightarrow \text{Rev } s}\ (\textit{Revert}) \qquad \frac{}{\Gamma \vdash \langle c,\ \text{Rev } s\rangle \Rightarrow \text{Rev } s}\ (\textit{RevState})$$

$$\frac{s \in b \qquad \Gamma \vdash \langle \textbf{Skip},\ \mathit{Normal}\ s\rangle\ \Rightarrow\ \mathit{Normal}\ s}{\Gamma \vdash \langle \textbf{Require } b,\ \mathit{Normal}\ s\rangle\ \Rightarrow\ \mathit{Normal}\ s}\ (\textit{RequireTrue})$$

$$\frac{s \notin b \qquad \Gamma \vdash \langle \textbf{Revert},\ \mathit{Normal}\ s\rangle\ \Rightarrow\ \mathit{Rev}\ s}{\Gamma \vdash \langle \textbf{Require } b,\ \mathit{Normal}\ s\rangle\ \Rightarrow\ \mathit{Rev}\ s}\ (\textit{RequireFalse})$$

$$\frac{\Gamma \vdash \langle \mathit{bdy},\ \mathit{Normal}\ s\rangle\ \Rightarrow\ \mathit{Normal}\ t}{\Gamma \vdash \langle \textbf{Init } \mathit{bdy}\ \mathit{rvrt},\ \mathit{Normal}\ s\rangle\ \Rightarrow\ \mathit{Normal}\ t}\ (\textit{ExecNormal})$$

$$\frac{\Gamma \vdash \langle \mathit{bdy},\ \mathit{Normal}\ s\rangle\ \Rightarrow\ \mathit{Rev}\ t}{\Gamma \vdash \langle \textbf{Init } \mathit{bdy}\ \mathit{rvrt},\ \mathit{Normal}\ s\rangle\ \Rightarrow\ \mathit{Rev}\ (\mathit{rvrt}\ s\ t)}\ (\textit{ExecRev})$$

**Fig. 1.** Big-step semantics rules for `SOLI`

### Exceptions

To deal with exceptions, the EVM has two available opcodes: `REVERT` and `INVALID`. Both undo all state changes, but `REVERT` will also allow to return a value and refund all remaining gas to the caller, whereas `INVALID` will simply consume all remaining gas. Solidity uses these opcodes to handle exceptions using the `revert()`, `require()` and `assert()` functions. The `require()` and `assert()` functions receive a bool and throw the respective exception if the condition is not met while `revert()` simply throws the exception. Both `revert()`

and `require()` use the `REVERT` opcode and can also receive an error message to display to the user; `assert()` uses the `INVALID` opcode.

In `SOLI` the first two are modeled. `revert()` corresponds to the *Revert* command, which modifies the current state type from *Normal* to *Rev*. The `require()` function is defined by the *Require* command, as a conditional statement.

$$\begin{aligned} \mathit{Require} &:: {'s}\ \mathit{bexp} \Rightarrow {'s}\ \mathit{com} \\ \mathit{Require}\ b &:= \textbf{If}\ b\ \textbf{Skip}\ \textbf{Revert} \end{aligned}$$

**Calling a Function**

In order to model the different types of function call, calling must be extended with the following definition, which introduces the modeling of passing arguments, resetting local variables and returning results.

$$\begin{aligned} &\mathit{call} :: [\![ {'s} \Rightarrow' s,\ \mathit{fname},\ {'s} \Rightarrow' s \Rightarrow' s,\ {'s} \Rightarrow' s \Rightarrow' s\ \mathit{com} ]\!] \Rightarrow {'s}\ \mathit{com} \\ &\mathit{call}\ \mathit{pass}\ f\ \mathit{return}\ \mathit{result} := \\ &\qquad \textbf{DynCom}\ (\lambda s.\ (\textbf{Upd}\ \mathit{pass};;\ \textbf{Call}\ f;; \\ &\qquad\qquad (\textbf{DynCom}\ (\lambda t.\ \textbf{Upd}\ (\mathit{return}\ s);;\ \mathit{result}\ s\ t)))) \end{aligned}$$

Here **DynCom** is used to abstract over the state space and refer to certain program states. The initial state $s$ is captured by the first **DynCom** and the state after executing the body of the called procedure, $t$, by the second. The function *pass*, receives the initial state $s$ and is used to pass the arguments of the function to the intended variables in the memory before the body of the function is executed. The *return* function is used to return from the procedure by cleaning the state, that is by restoring the local variables. In the case of a function call with a return value, the *result* function is used to communicate the results to the caller environment by updating the result variable. The control flow of *call* is depicted is in Fig. 2.

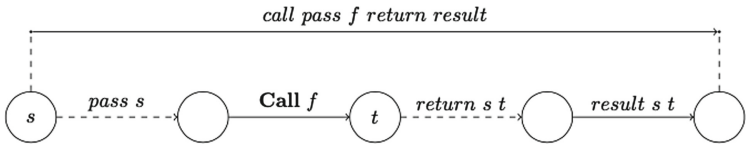


**Fig. 2.** Control flow for *call*

The big-step execution rule for *call*, Fig. 3, follows intuitively from the above description. First the body of the called function is executed after passing the arguments, that is, starting in state *pass* $s$. Then the result command is executed after returning from the call, that is, starting in state *return* $s$ $t$.

$$\frac{(the\ (\Gamma f),\ Normal\ (pass\ s))\ \Rightarrow\ Normal\ t \quad (result\ s\ t,\ Normal\ (return\ s\ t))\ \Rightarrow\ u}{(call\ pass\ f\ return\ result,\ Normal\ s)\ \Rightarrow\ u}\ (ExecCall)$$

**Fig. 3.** Big-step execution rule for *call*

In Solidity, a function call can be internal or external. For *internal calls*, functions are in the same contract and so state variables, memory, and execution context are the same and we only need to model function arguments and results. *External calls*, which call functions from other contracts, are done via message call. All function arguments have to be copied to memory and, after execution, the memory needs to be restored. In addition, some execution environment variables are updated, such as *msg_sender*, *msg_value*, *msg_data* and *address_this*.

**Reverting State Changes**

In Solidity, whenever an error occurs, for instance when some condition is not satisfied, a `REVERT` exception is thrown and all state changes made in the current call must be reverted.

Suppose one wants to execute the `SOLI` statement *bdy* starting in a normal type state $s$. The execution can run without any errors and terminate in a normal state $t$. But, if an exception is thrown, the execution must be stopped with the current *Rev* state $t'$ in order to proceed to the state reversion. This is modeled with **Handle** *bdy c*, where $c$ is the statement which handles the reversion.

Inside $c$ the state is first passed to a normal state in order to allow the regular `SOLI` statements, for instance **Upd** to be executed. The update of the state variables to their original value is made with the *rvrt* function, which receives the initial state $s$ and the current state $t$. Finally the error is propagated by re-throwing **Revert**. The control flow for a statement execution is depicted in Fig. 4.

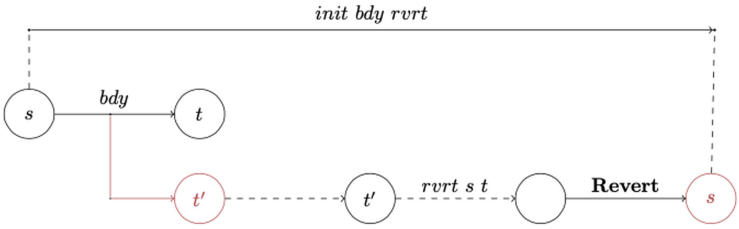


**Fig. 4.** Control flow for *Init*

In order to actually revert the state, first one needs to get hold of the initial state $s$ which can be captured using **DynCom**. Also, while taking care of the state reversion, another **DynCom** is used to refer to the current state $t$ when updating the variables.

Whenever a statement, such as a function, is written in SOLI it is encapsulated in an *Init* command which receives the function body and the *rvrt* function which models the reset of all state variables in case of error.

$$\begin{aligned} &\mathit{Init} :: [\![ {'s}\ \mathit{com}, {'s} \Rightarrow {'s} \Rightarrow {'s} ]\!] \Rightarrow {'s}\ \mathit{com} \\ &\mathit{Init}\ \mathit{bdy}\ \mathit{rvrt} := \mathbf{DynCom}\ (\lambda s.\ (\mathbf{Handle}\ \mathit{bdy};;\ (\mathbf{DynCom}\ (\lambda t. \\ &\qquad\qquad\qquad\qquad\qquad\qquad \mathbf{Upd}\ (\mathit{rvrt}\ s);;\ \mathbf{Revert})))) \end{aligned}$$

The big-step execution rules for *Init* are defined in Fig. 1. For normal execution it is immediate, it is just the regular execution of *bdy*. If an exception is thrown when executing *bdy*, its execution stops in a revert type state $t$ and the full execution will terminate in state $\mathit{rvrt}\ s\ t$.

### 3.5 State Space

Since the goal of this work is to verify properties about specific programs, it was chosen to explicitly state the HOL type for each variable by working with records. Some of the used types are constructed using the HOL type word, which represents a bit.

$$\begin{aligned} \mathit{byte} &:= 8\ \mathit{word} \\ \mathit{address} &:= 160\ \mathit{word} \\ \mathit{uint} &:= 256\ \mathit{word} \end{aligned}$$

A record of Isabelle/HOL is a collection of fields where each has a specified name and type. A record comes with select and update operations for each field. Record types can also be defined by extending other record types. A record *st* represents the storage of a Solidity contract and *loc* the local variables for the functions in that contract. We illustrate this concept for an electronic voting contract in Sect. 5.1.

### 3.6 Environment Variables

Solidity defines a set of *global variables* regarding the execution environment, mainly to provide information about the blockchain. For a block, we need variables that keep track of the current block's hash, miner's address, difficulty, gaslimit, number and timestamp. For a transaction, we need variables that keep track of the current message call: data, gas, sender and signature, and for the whole transaction: gasprice and origin. In SOLI these variables are part of the environment record *env*, which is defined in Fig. 5.

An *account state* is defined as a record *Account* with four fields corresponding to its nonce, balance, storage and code, also represented in Fig. 5. The *world state* is defined as a field of the *env* record *gs* which maps addresses to their account states.

$$
\begin{array}{rcl}
\textit{record account\_state} & := & \\
\textit{nonce} & := & \textit{uint} \\
\textit{balance} & :: & \textit{uint} \\
\textit{storage} & :: & \textit{uint} \Rightarrow \textit{uint} \\
\textit{code} & :: & \textit{byte list}
\end{array}
\qquad
\begin{array}{rcl}
\textit{record env} & := & \\
\textit{block\_coinbase} & :: & \textit{uint} \\
\textit{block\_difficulty} & :: & \textit{address} \\
\textit{block\_gaslimit} & :: & \textit{uint} \\
\textit{block\_number} & :: & \textit{uint} \\
\textit{block\_timestamp} & :: & \textit{uint} \\
\textit{msg\_data} & :: & \textit{byte list} \\
\textit{msg\_sender} & :: & \textit{address} \\
\textit{msg\_value} & :: & \textit{uint} \\
\textit{tx\_origin} & :: & \textit{address} \\
\textit{tx\_gasPrice} & :: & \textit{uint} \\
\textit{address\_this} & :: & \textit{address} \\
\textit{gs} & :: & \textit{address} \Rightarrow \textit{account\_state}
\end{array}
$$

**Fig. 5.** Account state representation and environment variables

## 4 Hoare Logic

In this section we present Hoare logic, a system proposed by Tony Hoare [12,13], and its formalization for `SOLI` regarding partial correctness. We extended the proof system in [23] to a relevant subset of Solidity.

### 4.1 The Proof System

A Hoare logic formula is a triple of the form $P\ c\ Q$, where $c$ is a command and $P$ and $Q$ are assertions, the precondition and postcondition, respectively. In EVM, command execution can result in a $Normal$ or in a $Rev$ state. To model this feature, we split the postcondition in two, $Q$ and $A$, for regular and for exceptional termination respectively.

To reason about recursive procedures, a set of assumptions $\Theta$ is introduced. This set contains function specifications, which will be used as hypothesis when proving the body of a recursive procedure. An *assumption for a function* is a tuple that contains its precondition, name and both postconditions.

$$'s\ assmpt\ :=\ \langle 's\ assn,\ fname,\ 's\ assn,\ 's\ assn\rangle$$

The notation used for a derivable Hoare formula is associated with the procedure body environment $\Gamma$ and with the set of assumptions $\Theta$.

**Definition 8 (Hoare logic).** *A Hoare logic is defined for* `SOLI` *such that a derivable formula is represented by*

$$\Gamma, \Theta \vdash P\ c\ Q, A$$

$$\frac{}{\Gamma,\Theta \vdash Q\ \mathbf{Skip}\ Q, A}\ (Skip) \qquad \frac{}{\Gamma,\Theta \vdash \{s.\ f\ s \in Q\}\ \ (\mathbf{Upd}\ f)\ \ Q, A}\ (Upd)$$

$$\frac{\Gamma,\Theta \vdash P\ c_1\ R, A \qquad \Gamma,\Theta \vdash R\ c_2\ Q, A}{\Gamma,\Theta \vdash P\ (\mathbf{Seq}\ c_1\ c_2)\ Q, A}\ (Seq) \qquad \frac{\Gamma,\Theta \vdash (P \cap\ b)\ c\ P, A}{\Gamma,\Theta \vdash P\ (\mathbf{While}\ b\ c)\ (P \cap -b), A}\ (While)$$

$$\frac{\Gamma,\Theta \vdash (P \cap\ b)\ c_1\ Q, A \qquad \Gamma,\Theta \vdash (P \cap -b)\ c_2\ Q, A}{\Gamma,\Theta \vdash P\ (\mathbf{If}\ b\ c_1\ c_2)\ Q, A}\ (If)$$

$$\frac{\Gamma,\Theta \vdash P\ c_1\ Q, R \qquad \Gamma,\Theta \vdash R\ c_2\ Q, A}{\Gamma,\Theta \vdash P\ (\mathbf{Handle}\ c_1\ c_2)\ Q, A}\ (Handle) \qquad \frac{}{\Gamma,\Theta \vdash A\ \mathbf{Revert}\ Q, A}\ (Revert)$$

$$\frac{\forall s \in P.\ \ \Gamma,\Theta \vdash P\ (c\ s)\ Q, A}{\Gamma,\Theta \vdash P\ (\mathbf{DynCom}\ c)\ Q, A}\ (DynCom)$$

$$\frac{(P, f, Q, A)\ \in\ S \qquad \forall \langle P, f, Q, A\rangle \in S.\ \ f \in dom\ \Gamma\ \wedge\ \Gamma, (\Theta \cup S) \vdash P\ (the\ (\Gamma\ f))\ Q, A}{\Gamma,\Theta \vdash P\ (\mathbf{Call}\ f)\ Q, A}\ (CallRec)$$

$$\frac{\Gamma,\Theta \vdash (P \cap\ b)\ \mathbf{Skip}\ Q, A \qquad \Gamma,\Theta \vdash (P \cap -b)\ \mathbf{Revert}\ Q, A}{\Gamma,\Theta \vdash P\ (\mathbf{Require}\ b)\ Q, A}\ (Require)$$

$$\frac{\forall s \in P.\ \ \Gamma,\Theta \vdash P\ bdy\ Q, \{t.\ rvrt\ s\ t \in A\}}{\Gamma,\Theta \vdash P\ (\mathbf{Init}\ bdy\ rvrt)\ Q, A}\ (Init) \qquad \frac{(P, f, Q, A)\ \in\ \Theta}{\Gamma,\Theta \vdash P\ (\mathbf{Call}\ f)\ Q, A}\ (Asm)$$

$$\frac{\forall s \in P'.\ \exists PQA.\ \Gamma,\Theta \vdash P\ c\ Q, A\ \wedge\ Q \subseteq Q'\ \wedge\ A \subseteq A'\ \wedge\ s \in P}{\Gamma,\Theta \vdash P'\ c\ Q', A'}\ (Conseq)$$

**Fig. 6.** Hoare logic for *SOLI*

*where*

$$\begin{aligned} \Gamma &:: fname\ \Rightarrow\ 's\ com \\ \Theta &:: 's\ assmpt\ set \\ P, Q, A &:: 's\ assn \\ c &:: 's\ com, \end{aligned}$$

*and the proof system is constituted by the rules in Fig. 6.*

There is a rule for each `SOLI` command and, additionally, the *Asm* and *Consequence* rules. To have an intuitive meaning for the rules of this system, one should read it backwards. For instance, for the *Upd* rule, if $Q$ holds (for regular execution) after the update then $P$ is the set of states such that the application of $f$ to them belongs to $Q$. *Skip* and *Revert* have the intuitive meaning of doing nothing, and *Seq* and *Handle* correspond respectively to the cases where $c_1$ $c_2$ are

both executed, and $c_1$ throws an exception and $c_2$ is executed. In the *DynCom* rule, the triple has to hold for every state $s$ that satisfies the precondition as the dynamic command will depend on the initial state. The *CallRec* rule regards a set of function specifications $S$ whose bodies are verified and that is added to $\Theta$. Then, when one of these functions is called, the specification can be assumed using *Asm* rule.

The **Require** command is modelled as a conditional statement, hence both rules follow the same structure. In one of the branches the precondition $b$ holds, and in the other it does not. The **Init** statement corresponds to a regular execution of the body ending in a state for which the regular postcondition $Q$ holds or, in the case of an exception being thrown, the execution ends in a state such that by reverting all state changes the exceptional postcondition $A$ holds. A dependence on the initial state $s$ is introduced in the premise. The *Consequence* rule allows to strengthen the precondition as well as to weaken the postcondition.

### 4.2 Weakest Precondition Calculus

In order to verify properties about programs using Hoare logic, a backward propagation method is followed. In this method, sufficient conditions for a certain result, the postcondition, are determined. The rules are successively applied backwards, starting in the postcondition until the beginning of the program. Some side conditions may be generated.

The weakest precondition is then said to be the most lenient assumption on the initial state such that $Q, A$ will hold after the execution of the command $c$.

Weakest precondition calculus, also know as predicate transformer semantics (Dijkstra [6]), is a reformulation of Hoare logic. It constitutes a strategy to reduce the problem of proving a Hoare formula to the problem of proving an HOL assertion, which is called the verification condition. Since assertions are expressed as sets, reasoning about the conditions is expressed using set operations.

**Definition 9 (Weakest precondition calculus).** *Let $c$ be a command, $Q$ and $A$ assertions, and $wp^{\Gamma,\Theta}(c, Q, A)$ the weakest precondition of $Q, A$ for $c$. The weakest precondition calculus for* `SOLI` *is inductively defined as follows:*

$$wp^{\Gamma,\Theta}\ (\textbf{Skip},\ Q,\ A)\ =\ Q$$
$$wp^{\Gamma,\Theta}\ (\textbf{Revert},\ Q,\ A)\ =\ A$$
$$wp^{\Gamma,\Theta}\ (\textbf{Upd}\ f,\ Q,\ A)\ =\ \{s.\ f\ s \in Q\}$$
$$wp^{\Gamma,\Theta}\ (\textbf{Seq}\ c_1\ c_2,\ Q,\ A)\ =\ wp^{\Gamma,\Theta}\ (c_1,\ wp^{\Gamma,\Theta}\ (c_2,\ Q,\ A),\ A)$$
$$wp^{\Gamma,\Theta}\ (\textbf{If}\ b\ c_1\ c_2,\ Q,\ A)\ =\ \{s.\ (s \in b \longrightarrow s \in wp^{\Gamma,\Theta}\ (c_1,\ Q,\ A)) \wedge (s \notin b\ \longrightarrow\ s \in wp^{\Gamma,\Theta}\ (c_2,\ Q,\ A))\}$$
$$wp^{\Gamma,\Theta}\ (\textbf{While}\ I\ b\ c,\ Q,\ A)\ = \{s.\ (s \in b\ \longrightarrow\ s \in wp^{\Gamma,\Theta}\ (\textbf{Seq}\ c\ (\textbf{While}\ I\ b\ c),\ Q,\ A)) \wedge (s \notin b\ \longrightarrow\ s \in wp^{\Gamma,\Theta}\ (c_2,\ Q,\ A))\}$$
$$wp^{\Gamma,\Theta}\ (\textbf{Call}\ f,\ Q,\ A)\ =\ P_f,\ \textit{such that}\ \ f \in \textit{dom}\ \Gamma\ \wedge\ (P_f,\ f,\ Q_f,\ A_f) \in \Theta$$
$$wp^{\Gamma,\Theta}\ (\textbf{DynCom}\ c,\ Q,\ A)\ =\ \bigcap_s\ wp^{\Gamma,\Theta}\ (c\ s,\ Q,\ A)$$
$$wp^{\Gamma,\Theta}\ (\textbf{Handle}\ c_1\ c_2,\ Q,\ A)\ =\ wp^{\Gamma,\Theta}\ (c_1,\ Q,\ wp^{\Gamma,\Theta}\ (c_2,\ Q,\ A))$$
$$wp^{\Gamma,\Theta}\ (\textbf{Require}\ b,\ Q,\ A)\ =\ \{s.\ (s \in b \longrightarrow s \in Q\ \wedge\ (s \notin b\ \longrightarrow\ s \in A)\}$$
$$wp^{\Gamma,\Theta}\ (\textbf{Init}\ bdy\ rvrt,\ Q,\ A)\ =\ \bigcap_s\ wp^{\Gamma,\Theta}\ (bdy,\ Q,\ \{t.\ rvrt\ s\ t \in A\})$$

The weakest precondition for the call of procedure $f$ corresponds to the precondition for its specification, present in the set of assumptions.

Since both **Dyncom** and **Init** have to consider every preceding state $s$, their weakest precondition corresponds to the intersection of certain weakest preconditions: in the former, to the *wp* of the command applied to each one of the states; in the latter, to the *wp* of *bdy* such that in case of exception the state reversion is applied to $s$. From the definition of **Require** it follows immediately that its weakest precondition is $Q$ if $b$ holds, and $A$ otherwise.

A strategy to generate verification conditions based on this calculus is described in Sect. 4.5.

### 4.3 Soundness

To prove soundness of our proof system, we follow the same technique as [23]. The formal definition for validity regarding partial correctness is defined as follows:

**Definition 10 (Validity—Partial Correctness)**

$$\Gamma \vDash P\ c\ Q, A \quad \textit{if}$$
$$\forall s\ t.\ \Gamma \vdash \langle c, s\rangle \Rightarrow t\ \wedge s \in \textit{Normal}\ 'P\ \longrightarrow\ t \in \textit{Normal}\ 'Q\ \cup\ \textit{Rev}\ 'A.$$

The goal is to prove that if a formula is derivable in the Hoare Logic (Fig. 6) then it also valid according to Definition 10. In the case of recursive calls, we need

to take into account the set of assumptions $\Theta$ and also the depth of recursion. However, the definitions of validity and big-step semantics are not rich enough to approach these properties. The notion of validity is thus extended with the set of assumptions.

**Definition 11 (Validity with context)**

$$\Gamma, \Theta \models P\ c\ Q, A \quad \textit{if}$$
$$\forall \langle P, f, Q, A\rangle \in \Theta.\ \Gamma \models P\ (\textbf{Call}\ f)\ Q, A \longrightarrow \Gamma \models P\ c\ Q, A.$$

Also, an additional set of big-step rules to deal with the depth of recursion are defined, where $n$ is the limit on nested procedure calls.

$$\Gamma \vdash \langle c,\ s\rangle \overset{n}{\Rightarrow} t$$

These rules are similar to the normal big-step rules (Fig. 1) except for the *Call* statement where the limit $n$ is decremented in each step to account for the depth of the recursion.

$$\frac{\Gamma \vdash \langle the\ (\Gamma\ f),\ \text{Normal}\ s\rangle \overset{n}{\Rightarrow} t}{\Gamma \vdash \langle \textbf{Call}\ f,\ \text{Normal}\ s\rangle \overset{n+1}{\Longrightarrow} t}\ (Call)$$

We can show that this new set of rules is monotonic with respect to the limit $n$.

**Lemma 1 (Monotonicity)**

$$\Gamma \vdash \langle c, s\rangle \overset{n}{\Rightarrow}\ t\ \wedge\ n \leq m\ \longrightarrow\ \Gamma \vdash \langle c, s\rangle \overset{m}{\Rightarrow}\ t$$

Validity can now be established regarding the limit on nested recursive calls.

**Definition 12 (Validity with limit)**

$$\Gamma \Vdash^{n} P\ c\ Q, A \quad \textit{if}$$
$$\forall s\ t.\ \Gamma \vdash \langle c, s\rangle \overset{n}{\Rightarrow} t\ \wedge\ s \in Normal\ 'P \longrightarrow\ t \in Normal\ 'Q\ \cup\ Rev\ 'A.$$

Finally the notions of validity with context and limit can be joined, leading to a definition which suits the needs to reason about recursive procedure calls.

**Definition 13 (Validity with limit and context)**

$$\Gamma, \Theta \Vdash^{n} P\ c\ Q, A \quad \textit{if}$$
$$\forall \langle P, f, Q, A\rangle \in \Theta. \Gamma \Vdash^{n} P\ (\textbf{Call}\ f)\ Q, A \longrightarrow\ \Gamma \Vdash^{n} P\ c\ Q, A.$$

The required conditions to show that Hoare rules preserve validity are now established.

**Lemma 2**

$$(\forall n.\ \Gamma, \Theta \models^{n} P\ c\ Q, A) \longrightarrow \Gamma, \Theta \models P\ c\ Q, A$$

**Lemma 3 (Soundness with limit and context).** *Let $\Gamma$ be the mapping between function names and their bodies, $\Theta$ the set of assumptions, c a* `SOLI` *command, $P$ the precondition and $Q, A$ the postconditions.*

$$If\ \Gamma, \Theta \vdash P\ c\ Q, A\ then\ (\forall n.\ \Gamma, \Theta \models^{n} P\ c\ Q, A).$$

The intended result follows directly from Lemmas 3 and 2.

**Theorem 1 (Soundness).** *Let $\Gamma$ be the mapping between function names and their bodies, $\Theta$ the set of assumptions, c a* `SOLI` *command, $P$ the precondition and $Q, A$ the postconditions.*

$$If\ \Gamma, \Theta \vdash P\ c\ Q, A\ then\ \Gamma, \Theta \models P\ c\ Q, A.$$

### 4.4 Completeness

Due to its inheritance from HOL, used to state assertions, Hoare logic is not complete. However, Cook [5] introduced the notion of relative completeness by separating incompleteness of the assertion language from incompleteness due to inadequacies in the axioms and rules for the programming language constructs. It is assumed that there is an oracle which can be inquired about the validity of an HOL assertion. The proof follows the method by Winskel [25] and relies on the concept of weakest precondition, Definition 9.

**Lemma 4**

$$\Gamma, \Theta \models P\ c\ Q, A \longrightarrow (s \in P \longrightarrow s \in wp(c,\ Q,\ A))$$

An auxiliary weakest precondition property regarding the derivation of a formula and its precondition is proven.

**Lemma 5**

$$\Gamma, \Theta \vdash\ wp(c, Q, A)\ c\ Q, A$$

Using Lemma 5, the (relative) completeness Theorem can now be proven.

**Theorem 2 ((Relative) Completeness).** *Let $\Gamma$ be the mapping between function names and their bodies, $\Theta$ the set of assumptions, c a* `SOLI` *command, $P$ the precondition and $Q, A$ the postconditions.*

$$If\ \Gamma, \Theta \models P\ c\ Q, A\ then\ \Gamma, \Theta \vdash P\ c\ Q, A.$$

### 4.5 Computation of Verification Conditions

One of the goals of this work is to develop a proof technique for the verification of properties about smart contracts. In this section, we extend previous work by Frade and Pinto [7] and present a method to compute the verification conditions of a program, which follows a backwards propagation through the weakest precondition.

To achieve this, the invariants for while loops must be supplied explicitly. The concept of annotated command is, therefore, introduced.

**Definition 14 (Annotated commands).** *The syntax for* annotated commands *is defined by the polymorphic datatype* $'s\ acom$.

$$
\begin{aligned}
's\ acom\ :=\ & \textbf{Skip} \mid \textbf{Upd}\ 's \Rightarrow 's \mid \textbf{Seq}\ 's\ acom\ 's\ acom \\
& \mid \textbf{If}\ 's\ bexp\ 's\ acom\ 's\ acom \mid \textbf{While}\ 's\ assn\ 's\ bexp\ 's\ acom \\
& \mid \textbf{Dyncom}\ 's \Rightarrow 's\ acom \mid \textbf{Call}\ fname \mid \textbf{Revert} \\
& \mid \textbf{Handle}\ 's\ acom\ 's\ acom \mid \textbf{Require}\ 's\ bexp \\
& \mid \textbf{Init}\ 's\ acom\ 's \Rightarrow 's \Rightarrow 's
\end{aligned}
$$

The weakest precondition calculus for annotated commands is the same as for normal commands except for **While** where it becomes the loop invariant, since it is a condition that must be met before each loop execution, or even if it isn't executed in the first place.

**Definition 15 (Weakest precondition calculus for annotated commands).** *The weakest precondition calculus for annotated commands is inductively defined as follows:*

$$
\begin{aligned}
& wp^{\Gamma,\Theta}\ (\textbf{Skip},\ Q,\ A) = Q \\
& wp^{\Gamma,\Theta}\ (\textbf{Revert},\ Q,\ A) = A \\
& wp^{\Gamma,\Theta}\ (\textbf{Upd}\ f,\ Q,\ A) = \{s.\ f\ s \in Q\} \\
& wp^{\Gamma,\Theta}\ (\textbf{Seq}\ c_1\ c_2,\ Q,\ A) = wp^{\Gamma,\Theta}\ (c_1,\ wp^{\Gamma,\Theta}\ (c_2,\ Q,\ A),\ A) \\
& wp^{\Gamma,\Theta}\ (\textbf{If}\ b\ c_1\ c_2,\ Q,\ A) = \{s.\ (s \in b \longrightarrow s \in wp^{\Gamma,\Theta}\ (c_1,\ Q,\ A)) \wedge \\
& \qquad\qquad (s \notin b \longrightarrow s \in wp^{\Gamma,\Theta}\ (c_2,\ Q,\ A))\} \\
& wp^{\Gamma,\Theta}\ (\textbf{While}\ I\ b\ c,\ Q,\ A) = I \\
& wp^{\Gamma,\Theta}\ (\textbf{Call}\ f,\ Q,\ A) = P_f \text{ such that } f \in dom\ \Gamma \wedge (P_f,\ f,\ Q_f,\ A_f) \in \Theta \\
& wp^{\Gamma,\Theta}\ (\textbf{DynCom}\ c,\ Q,\ A) = \bigcap_s wp^{\Gamma,\Theta}\ (c\ s,\ Q,\ A) \\
& wp^{\Gamma,\Theta}\ (\textbf{Handle}\ c_1\ c_2,\ Q,\ A) = wp^{\Gamma,\Theta}\ (c_1,\ Q,\ wp^{\Gamma,\Theta}\ (c_2,\ Q,\ A)) \\
& wp^{\Gamma,\Theta}\ (\textbf{Require}\ b,\ Q,\ A) = \{s.\ (s \in b \longrightarrow s \in Q \wedge (s \notin b \longrightarrow s \in A)\} \\
& wp^{\Gamma,\Theta}\ (\textbf{Init}\ bdy\ rvrt,\ Q,\ A) = \bigcap_s wp^{\Gamma,\Theta}\ (bdy,\ Q,\ \{t.\ rvrt\ s\ t \in A\})
\end{aligned}
$$

The verification condition computation for a command can be obtained using the structure of each rule in the proof system. An important property about these is that the verification conditions are computed independently from preconditions, leaving only the need to check their inclusion in the propagated weakest precondition. This prevents the generation of unnecessary verification conditions.

**Definition 16 (Verification condition I).** *The* verification condition *function vc is defined as follows:*

$$vc\ (\Gamma, \Theta \vdash\ P\ c\ Q, A)\ =\ P \subseteq wp^{\Gamma,\Theta}\ (c,\ Q,\ A)\ \cup\ vc^{\Gamma,\Theta}_{aux}\ (c,\ Q,\ A),$$

*where the auxiliary verification condition* $vc^{\Gamma,\Theta}_{aux}$ *is inductively defined as follows:*

$$vc^{\Gamma,\Theta}_{aux}\ (\textbf{Skip},\ Q,\ A)\ =\ \emptyset$$
$$vc^{\Gamma,\Theta}_{aux}\ (\textbf{Revert},\ Q,\ A)\ =\ \emptyset$$
$$vc^{\Gamma,\Theta}_{aux}\ (\textbf{Upd}\ f,\ Q,\ A)\ =\ \emptyset$$
$$vc^{\Gamma,\Theta}_{aux}\ (\textbf{Seq}\ c_1\ c_2,\ Q,\ A)\ =\ vc^{\Gamma,\Theta}_{aux}\ (c_1,\ wp^{\Gamma,\Theta}(c_2,\ Q,\ A),\ A)\ \cup\ vc^{\Gamma,\Theta}_{aux}\ (c_2,\ Q,\ A)$$
$$vc^{\Gamma,\Theta}_{aux}\ (\textbf{If}\ b\ c_1\ c_2,\ Q,\ A)\ =\ vc^{\Gamma,\Theta}_{aux}\ (c_1,\ Q,\ A)\ \cup\ vc^{\Gamma,\Theta}_{aux}\ (c_2,\ Q,\ A)$$
$$vc^{\Gamma,\Theta}_{aux}\ (\textbf{While}\ I\ b\ c,\ Q,\ A)\ =\ (I \cap b) \subseteq wp^{\Gamma,\Theta}\ (c,\ I,\ A)\ \cup\ vc^{\Gamma,\Theta}_{aux}\ (c,\ I,\ A)\ \cup\ (I \cap -b) \subseteq Q$$
$$vc^{\Gamma,\Theta}_{aux}\ (\textbf{Call}\ f,\ Q,\ A)\ =\ Q_f \subseteq Q$$
$$vc^{\Gamma,\Theta}_{aux}\ (\textbf{DynCom}\ c,\ Q,\ A)\ =\ \bigcap_s\ vc^{\Gamma,\Theta}_{aux}\ (c\ s,\ Q,\ A)$$
$$vc^{\Gamma,\Theta}_{aux}\ (\textbf{Handle}\ c_1\ c_2,\ Q,\ A)\ =\ vc^{\Gamma,\Theta}_{aux}\ (c_1,\ Q,\ wp^{\Gamma,\Theta}(c_2,\ Q,\ A))\ \cup vc^{\Gamma,\Theta}_{aux}\ (c_2,\ Q,\ A)$$
$$vc^{\Gamma,\Theta}_{aux}\ (\textbf{Require}\ b,\ Q,\ A)\ =\ \emptyset$$
$$vc^{\Gamma,\Theta}_{aux}\ (\textbf{Init}\ bdy\ rvrt,\ Q,\ A)\ =\ \bigcap_s\ vc^{\Gamma,\Theta}_{aux}\ (bdy,\ Q,\ \{t.\ rvrt\ s\ t \in A\})$$

However, upon the verification of a program with any number of function calls, their specification must have been verified and added to the set of assumptions. A verification condition suitable for every program is then formalized.

**Definition 17 (Verification condition II).** *Let S be the set of specifications for every function whose call is generated by the execution of c. The verification condition function VC for c is defined as*

$$VC\ (\Gamma, \Theta \vdash\ P\ c\ Q, A)\ =\ P \subseteq wp^{\Gamma,\Theta}\ (c,\ Q,\ A)\ \cup$$
$$vc^{\Gamma,\Theta}_{aux}\ (c,\ Q,\ A)\ \cup \bigcup_{\langle P,f,Q,A\rangle \in S} vc\ (\Gamma, (\Theta \cup S) \vdash\ P\ (the\ (\Gamma\ f))\ Q, A).$$

**An Alternative Formulation of Rules**

The verification condition computations explicitly separate the main verification condition (the inclusion of precondition in the weakest precondition of the program) from auxiliary conditions (generated from the structure of the rules). In order to construct a proof which follows this backwards propagation method,

some Hoare rules are modified to a structure that will be referred as weakest precondition style. Following the method above, we were able to obtain the same rules as in [23] together with a new rule for the Solidity command *Require*. The set of rules is presented in Fig. 7.

$$\frac{P \subseteq Q}{\Gamma, \Theta \vdash P\ \mathbf{Skip}\ Q, A}\ (Skip') \qquad \frac{P \subseteq A}{\Gamma, \Theta \vdash P\ \mathbf{Revert}\ Q, A}\ (Revert')$$

$$\frac{\begin{array}{c} P \subseteq \{s.\ (s \in b \longrightarrow s \in P_1) \wedge (s \notin b \longrightarrow s \in P_2)\} \\ \Gamma, \Theta \vdash P_1\ c_1\ Q, A \qquad \Gamma, \Theta \vdash P_2\ c_2\ Q, A \end{array}}{\Gamma, \Theta \vdash P\ (\mathbf{If}\ b\ c_1\ c_2)\ Q, A}\ (If')$$

$$\frac{P \subseteq \{s.\ f\ s\ \in Q\}}{\Gamma, \Theta \vdash P\ (\mathbf{Upd}\ f)\ Q, A}\ (Upd')$$

$$\frac{P \subseteq I \qquad \Gamma, \Theta \vdash (I \cap b)\ c\ I, A \qquad (I \cap -b)\ \subseteq Q}{\Gamma, \Theta \vdash P\ (\mathbf{While}\ b\ c)\ Q, A}\ (While')$$

$$\frac{P \subseteq \{s.\ (s \in b \longrightarrow s \in Q) \wedge (s \notin b \longrightarrow s \in A)\}}{\Gamma, \Theta \vdash P\ (Require\ b)\ Q, A}\ (Require')$$

$$\frac{\begin{array}{c} P \subseteq \{s.\ \exists Z.\ pass\ s \in P'\ Z \wedge (\forall t.\ t \in Q'\ Z \longrightarrow return\ s\ t \in R\ s\ t)\} \\ \forall Z.\ \Gamma, \Theta \vdash (P'\ Z)\ (Call\ f)\ (Q'\ Z), A \qquad \forall st.\ \Gamma, \Theta \vdash (R\ s\ t)\ (result\ s\ t)\ Q, A \end{array}}{\Gamma, \Theta \vdash P\ (call\ pass\ f\ return\ result)\ Q, A}\ (CallRec')$$

**Fig. 7.** Weakest precondition style rules

## 5 Application to Real-World Smart Contracts

In this section we illustrate the usage of our method for proving properties about smart contracts.

### 5.1 Electronic Voting

In this example an electronic voting contract, *Ballot*[1], which features automatic and transparent vote counting and delegate voting, is presented. This is an example of a successful contract verification that has some complex properties originated by the loop invariant, and that introduces the need to prove additional lemmas, defined generally.

The *Ballot* contract contains a *Voter* struct constituted by the weight of the voter (accumulated by delegation), a boolean that states whether the person already voted, the delegate's address (in case of vote delegation) and the index of the voted proposal. It also contains a *Proposal* struct constituted by the proposal name and corresponding vote count. As global variables the contract contains

[1] https://solidity.readthedocs.io/en/v0.5.12/solidity-by-example.html.

an address *chairperson*, a mapping *voters* between addresses and *Voter* structs, and a list of proposals *proposals*, which are stored in the *st* record.

$$\begin{array}{rcl} \textbf{record}\ st & = & env + \\ chairperson & :: & address \\ voters & :: & address \Rightarrow Voter \\ proposals & :: & Proposal\ list \end{array} \qquad \begin{array}{rcl} \textbf{record}\ loc & = & st + \\ winningVoteCount & :: & int \\ p & :: & int \\ winningProposal\_out & :: & int \\ r & :: & int \\ winningVoteCount\_out & :: & int \end{array}$$

This example is focused on the verification of the *winnerName* function (Fig. 8), which returns the name of the winning proposal by calling the *winningProposal* function which returns the corresponding index. This function finds the maximum value of *voteCount* in the list of proposals using a loop. It introduces the necessity of supplying an invariant and to verify that, while the list is gone through, the current maximum is correctly computed. The verification requires a definition of the maximum of a list and additional lemmas on the matter to be introduced.

```
winningProposal_com :: loc com
winningProposal_com ≡ INIT(
    'winningProposal_out ::= 0;;
    'winningVoteCount ::= voteCount 'proposals[0];;
    'p ::= 1;;
    WHILE ('p < length 'proposals) DO
        IF ('winningVoteCount < voteCount 'proposals['p])
        THEN 'winningVoteCount ::= voteCount 'proposals['p];;
             'winningProposal_out ::= 'p;;
        'p ::=' p + 1)

winnerName_com :: loc com
winnerName_com ≡ INIT(
    call_wp;;
    'winnerName_out ::= name 'proposals['r])

call_wp :: st com
call_wp ≡ call (λs. s) winningProposal (λst. t)
        (λst. Upd (λu. u(|r := winningProposal_out t|)))
```

**Fig. 8.** *winningProposal* and *winnerName* functions

*INIT* is defined as an *init* statement to revert all state changes in case of exception, that is, resetting the global variables to their initial values. In order to internally call the *winningProposal* function, *call_wp* is defined as a *call* statement.

The verification consists in showing that the return value $r$ from the *winningProposal* function corresponds to the maximum vote count of the list and that the output of the *winnerName* function is the corresponding name. The initial values for global variables are stored in the auxiliary variables *chair*, *vtrs* and *prop*.

$$
\begin{aligned}
&\Gamma, \Theta \vdash \ \{|\ chair =' chairperson \ \wedge \ vtrs =' voters \ \wedge \ prop =' proposals \ |\} \\
&\quad winnerName\_com \\
&\qquad \{|\ max'\ (map\ voteCount\ prop) \ = \ (map\ voteCount\ prop)['r] \wedge \\
&\qquad\ 'winnerName\_out \ = \ name\ prop['r] \ |\}, \\
&\qquad \{|\ 'chairperson = chair \ \wedge \ 'voters = vtrs \ \wedge \ 'proposals = prop \ |\}
\end{aligned}
$$

$$
\begin{aligned}
I = \{|\ & 1 \ \leq \ 'p \ \leq length\ prop \wedge \\
& 'winningVoteCount = max'(take\ 'p(map\ voteCount\ prop)) \wedge \\
& 'winningVoteCount = (map\ voteCount\ prop)['winningProposal\_out] \ |\}
\end{aligned}
$$

The $max'$ function was defined to retrieve the maximum of a list of natural numbers. Invariant $I$ states the limits that should be verified on the value of $p$ and that the current maximum is correctly computed.

The application of the verification method results, after simplification, in two conditions, which are solved through the use of the auxiliary lemmas.

1. $'proposals \neq \{\} \implies 1 \leq length\ 'proposals \wedge$
   $voteCount\ 'proposals[0] = \ max'(take\ 1\ (map\ voteCount\ 'proposals))$
2. $'p < length\ \ 'proposals \implies$
   $(max'(take\ 'p\ (map\ voteCount\ 'proposals)) < voteCount\ 'proposals['p] \longrightarrow$
   $max'(take('p+1)(map\ voteCount\ 'proposals)) = voteCount\ 'proposals['p]) \wedge$
   $(\neg max'(take\ 'p\ (map\ voteCount\ 'proposals)) < \ voteCount\ 'proposals['p] \ \longrightarrow$
   $max'\ (take\ ('p+1)\ (map\ voteCount\ 'proposals)) =$
   $max'\ (take\ 'p\ (map\ voteCount\ 'proposals)))$

The first condition results from the precondition inclusion and is proved using Lemma 6, together with the fact that

$$voteCount\ 'proposals[0] = (map\ voteCount\ 'proposals)[0]$$

**Lemma 6**

$$l \neq \{\} \implies l[0] = max'\ (take\ 1\ l).$$

The second, which results from the invariant verification conditions, is proven using the $max'$ definition and Lemma 7, that follows by induction on the structure of the list.

**Lemma 7**

$$\{x_1, \ldots, x_n\} \neq \{\} \implies max'\ (\{x_1, \ldots, x_n\}) = max\ (max'\ \{x_1, \ldots, x_{n-1}\})\ \ x_n$$

### 5.2 Ethereum Tokens

Solidity is prone to underflows and overflows since the EVM works with 256-bit unsigned integers and, therefore, all operations are performed modulo $2^{256}$. As an example of a vulnerable implementation of an ERC20 token, the Hexagon (*HXG*) token[2] is taken into account. This example illustrates that some proofs on the alleged specification of a contract may not terminate but give us important insights about the source of its vulnerability. Amongst its global variables it contains a mapping *balanceOf*, a mapping *allowances* and a *uint burnPerTransaction*, which is set to 2. In this example we analyze the *transfer* function (Fig. 9). According to its specification it should be the case that, after the function is executed, the balance of address *from* decreases by $val + 2$, the balance of address *to* increases by *val*, and the balance of address *adr0* increases by 2.

Note that the conditions in the postcondition are stated using the *uint* Isabelle function which allows to check that an operation does not underflow or overflow. The *uint_arith* Isabelle tactic is used in the proof to unfold this definition, which can take some time to run. It gets stuck with a verification condition which depends on the fact that $uint\ (val + 2) = uint\ val + 2$.

```
transfer :: loc com
transfer ≡ INIT (
 REQUIRE ('to ≠ 'adr0);;
 REQUIRE ('balanceOf 'frm ≥ 'val + 'burnPerTransaction);;
 REQUIRE ('balanceOf 'to + 'val ≥ 'balanceOf 'to);;
 'balanceOf ::= 'balanceOf ('frm := 'balanceOf 'frm  - ('val + 'burnPerTransaction));;
 'balanceOf ::= 'balanceOf ('to := 'balanceOf 'to + 'val);;
 'balanceOf ::= 'balanceOf ('adr0 := 'balanceOf 'adr0  + 'burnPerTransaction);;
 'currentSupply ::= 'currentSupply  -  'burnPerTransaction)
```

**Fig. 9.** *transfer* function of *Hexagon* contract

[2] https://etherscan.io/address/0xB5335e24d0aB29C190AB8C2B459238Da1153cEBA#code.

$$
\begin{aligned}
\Gamma, \Theta \vdash\ & \{|\ {'burnPerTransaction} = 2\ \wedge\ from = {'frm}\ \wedge\ t = {'to}\ \wedge\ a = {'adr0}\ \wedge \\
& \quad bal\_from = {'balanceOf}\ from\ \wedge\ bal\_to = {'balanceOf}\ t\ \wedge \\
& \quad bal\_a = {'balanceOf}\ a\ \wedge\ supply = {'currentSupply}\ \wedge \\
& \quad from \neq a\ \wedge\ from \neq t\ \wedge\ a \neq t\ |\} \\
& transfer \\
& \{|\ uint({'balanceOf}\ from) = uint\ bal\_from - (uint\ {'val} + 2)\ \wedge \\
& \quad uint({'balanceOf}\ t) = uint\ bal\_to + uint\ {'val}\ \wedge \\
& \quad uint({'balanceOf}\ a) = uint\ bal\_a + 2\ |\}, \\
& \{|\ {'balanceOf}\ from = bal\_from\ \wedge\ {'balanceOf}\ t = bal\_to\ \wedge \\
& \quad {'balanceOf}\ a = bal\_a\ |\}
\end{aligned}
$$

Looking at *transfer* function there is no condition that ensures this and prevents overflow during the addition of $'val$ and $'burnPerTransaction$. Therefore, one is not able to prove the specification since there is no way to prove that

$$
\begin{aligned}
uint\ (balanceOf\ from) &= uint\ (bal\_from - (val + 2)) \\
&= uint\ bal\_from - (uint\ val + 2)
\end{aligned}
$$

This vulnerability can be exploited. Suppose the *transfer* function is called by an attacker with *val* equal to $2^{256} - 2$. It follows that $val + burnPerTransaction = 2^{256} - 2 + 2 = 0$ and therefore the second **REQUIRE** statement's guard will become $balanceOf\ 'frm \geq 0$ which is always true. The balance of $'frm$ is then decreased by 0 and the balance of $'to$ increased by $2^{256} - 2$.

To solve this issue a require statement can be added to the *transfer* function which checks if $'val\ +'\ burnPerTransaction < 2^{256}$.

### 5.3 Reentrancy

This example shows how the defined recursive features can be used to model reentrancy. A *DAO* is a Decentralized Autonomous Organization built using the Ethereum blockchain. In 2016, an hacker exploited a bug in the *DAO* contract which resulted in the loss of approximately $50 million in ether. This was the first reentrancy attack which consisted in draining funds using the attacker's fallback function.

A fallback function is a contract's function, with no arguments or return values, which is automatically executed whenever a call is made to the contract and none of its other functions match the given function identifier or when no data is supplied. This is the case when the contract receives ether, with no data specified. The vulnerability consisted in the fact that *DAO*'s *withdraw* function uses *call.value*() to send ether to the caller's account. Now, this triggers its fallback function, which contains arbitrary code defined by the owner.

To explain the technical aspects of this attack a simplified version, *babyDao*, is presented. The contract contains, as state variable, a mapping *credit* between addresses and their respective balances. The vulnerability is present in the

$withdraw\_com \;::\; loc\ com$
$withdraw\_com \;\equiv\; INIT($
**IF** $('credit\ user > 0)$
**THEN** $call\_value;;$
$'credit ::= 'credit(user := 0))$

$fallback \;::\; loc\ com$
$fallback \;\equiv\; INIT($
**IF** $(balance\ ('gs\ babyDao) - 'credit\ user \geq 0)$
**THEN** $call\_withdraw)$

**Fig. 10.** *withdraw* and malicious *fallback* function

*withdraw* function (Fig. 10) and the attacker's goal is to drain all caller's funds in the contract and send this value to his account. To perform this transference the function uses *call.value()* which triggers its fallback function, containing arbitrary code defined by the owner. This is modelled as the *call* statement *call_value*, which is defined so that the values of some environment variables are updated, the balance of *user* is increased by *msg_value* and the balance of *babyDao* is decreased by the same amount. The fallback function's code is then executed.

To write the specification for *withdraw*, the auxiliary variables $c$, $b$ and *bdao* are introduced.

$$\begin{aligned}\forall c\ b\ bdao\ .\ \Gamma, \Theta \vdash \{|\ & c = 'credit\ user\ \wedge\ b = balance\ ('gs\ user)\ \wedge \\ & bdao = balance\ ('gs\ babyDao)\ |\} \\ & withdraw \\ & \{|\ 'credit\ user = 0\ \wedge\ balance\ ('gs\ user) = b + c\ \wedge \\ & balance\ ('gs\ babyDao) = bdao - c\ |\}, \{\}\end{aligned}$$

In the case of a so called friendly fallback function, the specification for *withdraw* holds. However, suppose an attacker writes a *fallback* function which besides increasing the attacker's balance and decreasing the balance of *babyDao*, contains code that checks whether the balance of *babyDao* will remain bigger than or equal to 0 after another possible withdraw, and if so, calls *withdraw*.

Suppose the attacker has some credit $c$ and *bdao* is the total balance of *babyDao*. When the attacker calls the *withdraw* function, *call_value* transfers ether to the attacker, triggering its *fallback* function which may create another call to *withdraw*. This causes the attacker to receive the same amount of ether again and enter a recursive loop until all possible ether has been drained from *babyDao* without causing the function to fail, that is, the guard of the conditional statement in the *fallback* function never evaluates to false. The attacker's credit is only set to 0 after *babyDao*, and therefore, after all these recursive calls. The withdraw function ends up being called $\lfloor \frac{bdao}{c} \rfloor$ times and the attacker increases its value by $\lfloor \frac{bdao}{c} \rfloor \times c$.

In this case, the proof for the specification no longer holds but a proof for the attack can be established using the rules for multiple procedure recursive calls.

## 6 Conclusions

The main contribution from this work is the development of an imperative language and respective semantics system regarding a relevant subset of Solidity, based on a set of existent imperative languages in Isabelle/HOL, in particular the language proposed by Schirmer [23]. The main additions were the modelling of Solidity calls, both internal and external, Solidity exceptions, and reverting all state modifications. The relative completeness proof, based on the proof by Winskel [25], uses an auxiliary lemma that involves the concept of weakest precondition. After the addition of the *Dyncom, Require* and *Init* cases to the *wp* and *vc* computations, following the work by Frade and Pinto [7], we extend the proof of the lemma with the *Call, Handle, Revert, Dyncom, Require* and *Init* cases.

The main advantage of using a proof assistant is the richness with which properties about programs can be expressed as we saw in Sect. 5. From the *Ballot* example, it can be seen how invariants increase the complexity of a proof, but also how that complexity can be tackled using auxiliary properties. Also, the example of *Ethereum tokens* was analyzed and in most cases, upon a correct specification, the tactic *uint_arith* is able to find, or at least give a hint of, overflows and underflows. Finally, the possibility of recursion allows to model reentrancy vulnerabilities and fallback function attacks.

## References

1. Amani, S., Bégel, M., Bortin, M., Staples, M.: Towards verifying Ethereum smart contract bytecode in Isabelle/HOL. In: CPP 2018, pp. 66–77. ACM (2018)
2. Bartoletti, M., Galletta, L., Murgia, M.: A minimal core calculus for solidity contracts. In: Pérez-Solà, C., Navarro-Arribas, G., Biryukov, A., Garcia-Alfaro, J. (eds.) DPM/CBT 2019. LNCS, vol. 11737, pp. 233–243. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-31500-9_15
3. Bhargavan, K., et al.: Formal verification of smart contracts: short paper. In: PLAS 2016, pp. 91–96. ACM (2016)
4. Buterin, V.: Ethereum: a next-generation cryptocurrency and decentralized application platform
5. Cook, S.A.: Soundness and completeness of an axiom system for program verification. SIAM J. Comput. **7**, 70–90 (1978)
6. Dijkstra, E.W., Scholten, C.S.: Predicate Calculus and Program Semantics. Texts and Monographs in Computer Science. Springer, Heidelberg (1990). https://doi.org/10.1007/978-1-4612-3228-5
7. Frade, M.J., Pinto, J.S.: Verification conditions for source-level imperative programs. Comput. Sci. Rev. **5**(3), 252–277 (2011)
8. Grishchenko, I., Maffei, M., Schneidewind, C.: Foundations and tools for the static analysis of Ethereum smart contracts. In: Chockler, H., Weissenbacher, G. (eds.) CAV 2018. LNCS, vol. 10981, pp. 51–78. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-96145-3_4

9. Grishchenko, I., Maffei, M., Schneidewind, C.: A semantic framework for the security analysis of Ethereum smart contracts. In: Bauer, L., Küsters, R. (eds.) POST 2018. LNCS, vol. 10804, pp. 243–269. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-89722-6_10
10. Hildenbrandt, E., et al.: KEVM: a complete formal semantics of the Ethereum virtual machine. In: CSF 2018, pp. 204–217. IEEE Computer Society (2018)
11. Hirai, Y.: Defining the Ethereum virtual machine for interactive theorem provers. In: Brenner, M., et al. (eds.) FC 2017. LNCS, vol. 10323, pp. 520–535. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-70278-0_33
12. Hoare, C.A.R.: An axiomatic basis for computer programming. Commun. ACM **12**(10), 576–580 (1969)
13. Hoare, C.A.R.: Procedures and parameters: an axiomatic approach. In: Engeler, E. (ed.) Symposium on Semantics of Algorithmic Languages. LNM, vol. 188, pp. 102–116. Springer, Heidelberg (1971). https://doi.org/10.1007/BFb0059696
14. Jiao, J., Kan, S., Lin, S., Sanán, D., Liu, Y., Sun, J.: Semantic understanding of smart contracts: executable operational semantics of solidity. In: SP 2020, pp. 1265–1282. IEEE Computer Society (2020)
15. Lincoln, P., Mitchell, J., Scedrov, A., Shankar, N.: Decision problems for propositional linear logic. Ann. Pure Appl. Logic **56**(1), 239–311 (1992)
16. Lincoln, P.D., Mitchell, J.C., Scedrov, A.: Linear logic proof games and optimization. Bull. Symbolic Logic **2**(3), 322–338 (1996)
17. Luu, L., Chu, D., Olickel, H., Saxena, P., Hobor, A.: Making smart contracts smarter. In: ACM CCS 2016, pp. 254–269. ACM (2016)
18. Mateus, P., Mitchell, J., Scedrov, A.: Composition of cryptographic protocols in a probabilistic polynomial-time process calculus. In: Amadio, R., Lugiez, D. (eds.) CONCUR 2003. LNCS, vol. 2761, pp. 327–349. Springer, Heidelberg (2003). https://doi.org/10.1007/978-3-540-45187-7_22
19. Mitchell, J.C., Ramanathan, A., Scedrov, A., Teague, V.: A probabilistic polynomial-time process calculus for the analysis of cryptographic protocols. Theor. Comput. Sci. **353**(1), 118–164 (2006)
20. Mythril. https://github.com/ConsenSys/mythril
21. Nakamoto, S.: Bitcoin: a peer-to-peer electronic cash system (2009)
22. Nikolic, I., Kolluri, A., Sergey, I., Saxena, P., Hobor, A.: Finding the greedy, prodigal, and suicidal contracts at scale. In: ACSAC 2018, pp. 653–663. ACM (2018)
23. Schirmer, N.: Verification of sequential imperative programs in Isabelle/HOL. Ph.D. thesis, Technical University Munich, Germany (2006)
24. Tsankov, P., Dan, A.M., Drachsler-Cohen, D., Gervais, A., Bünzli, F., Vechev, M.T.: Securify: practical security analysis of smart contracts. In: ACM CCS 2018, pp. 67–82. ACM (2018)
25. Winskel, G.: The Formal Semantics of Programming Languages: An Introduction. MIT Press, Cambridge (1993)
26. Wood, G.: Ethereum: a secure decentralised generalised transaction ledger. Ethereum Project Yellow Paper (2019)
27. 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