Secure Smart Contracts with Isabelle/Solidity

Diego Marmsoler[0000−0003−2859−7673], Asad Ahmed, and Achim D. Brucker[0000−0002−6355−1200]

University of Exeter, Exeter EX4 4PY, UK
{d.marmsoler, a.ahmed6, a.brucker}@exeter.ac.uk

Abstract. Smart contracts are programs stored on the blockchain, often developed in a high-level programming language, the most popular of which is Solidity. Smart contracts are used to automate financial transactions and thus bugs can lead to large financial losses. With this paper, we address this problem by describing a verification environment for Solidity in Isabelle/HOL. The framework consists of a novel formalization of the Solidity storage model, a shallow embedding of Solidity expressions and statements, an implementation of Isabelle commands to support a user in specifying Solidity smart contracts, and a verification condition generator to support a user in the verification. Compliance of the semantics is tested by a set of unit tests and the approach is evaluated by means of three case studies. Our results show that the framework can be used to verify even complex contracts with reasonable effort. Moreover, they show that the shallow embedding significantly reduces the verification effort compared to an alternative approach based on a deep embedding.

Keywords: Program Verification · Smart Contracts · Isabelle · Solidity

1 Introduction

Blockchain [33] is a novel technology used to store data in a decentralized manner, thereby providing *transparency*, *security*, and *trust*. Although the technology was originally invented to enable cryptocurrencies, it quickly found applications in several other domains, such as *finance* [23], *healthcare* [4], *land management* [10], and *identity management* [46].

One important innovation which comes with blockchains are so-called *smart contracts*. These are digital contracts which are automatically executed once certain conditions are met and are typically used to automate transactions on the blockchain. For instance, a payment for an item might be released instantly once the buyer and seller meet all the specified parameters of a deal. Every day, hundreds of thousands of new contracts are deployed managing millions of dollars' worth of transactions [45].

Technically, a smart contract is code which is deployed to a blockchain, and which can be executed by sending special transactions to it. Smart contracts are usually developed in a high-level programming language, the most popular

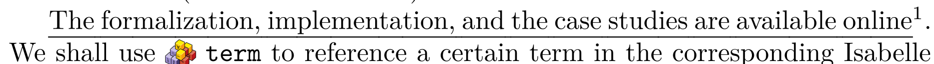
of which is *Solidity*. Solidity can be compiled to run on the Ethereum Virtual Machine (EVM), and thus, it works on all EVM-based smart contract platforms, such as Ethereum, Avalanche, Moonbeam, Polygon, BSC, and more. Currently, more than 90% of all smart contracts are developed using Solidity [27].

As with every computer program, *smart contracts may contain bugs which can be exploited*. However, since smart contracts are often used to automate financial transactions, such exploits may result in huge economic losses. For example, in 2016, a vulnerability in an Ethereum smart contract was exploited, resulting in a loss of approximately $60M [5]. More recently, hackers exploited a vulnerability in the DeFi-platform Poly Network to steal $600M [34]. Overall, it is estimated that since 2019, more than $5B have been stolen due to vulnerabilities in smart contracts [11].

The high impact of vulnerabilities in smart contracts, together with the fact that once deployed to the blockchain, they cannot be updated or removed easily, makes it important to "get them right" before they are deployed. To address this problem, we developed Isabelle/Solidity, a shallow embedding of Solidity in Isabelle/HOL. To this end, the major contributions of this paper are as follows:

1. *We provide a novel formalization of Solidity's storage model in Isabelle/HOL* (sec. 3.1). The formalization provides definitions for all different stores used in Solidity as well as functions to manipulate data structures stored on them.
2. *We provide a shallow embedding of Solidity expressions and statements in Isabelle/HOL.* The embedding consists of a formalization of a state monad (sec. 3.2) as well as definitions of a subset of expressions (sec. 3.3) and statements (sec. 3.4) in terms of the state monad.
3. *We implemented two new commands for Isabelle to support the specification and verification of Solidity smart contracts* (sec. 3.5). The commands are implemented in Isabelle/ML and generate definitions, theorems, and proofs for a contract specification given by a user.
4. *We provide a formalization of a weakest precondition calculus and corresponding verification condition generator for our embedding* (sec. 3.6). It can be used to reduce lemmas involving Solidity statements to HOL formulas which can then be discharged using standard Isabelle infrastructure.
5. *We provide a test suite to validate Solidity semantics* (sec. 4.1). The test suite includes set of test cases to test the semantics of expressions and statements.
6. *We formalize and verify three smart contracts in Isabelle/Solidity* (sec. 4.2). Each contract is specified using our framework, and then we verify a key invariant for each contract.

Our results suggest that the shallow embedding can significantly reduce the effort to verify concrete smart contracts. In particular, verifying the invariant of one of the case studies required 3 250 lines of Isabelle/Isar code in a deep embedding. In our shallow embedding, the same invariant was verified in less than 100 lines (a reduction of 97%).

The formalization, implementation, and the case studies are available online[1]. We shall use  `term` to reference a certain term in the corresponding Isabelle

[1] https://github.com/dmarmsoler/isabelle-solidity-shallow

source file. This will then be a clickable link which opens a theory file and highlights the corresponding formalization (for example `skip_monad`). The paper's artefact contains a virtual machine with an Isabelle installation and a copy of the same source files. It is also available online at https://zenodo.org/records/12770521.

2 Solidity

To provide a first impression of the language, lst. 1 shows a simple smart contract in Solidity which allows clients to deposit and withdraw funds. To this end, the contract keeps an internal record of funds transferred by its customers. This record is increased whenever a customer transfers additional funds via the `deposit` method. Once the customer calls the `withdraw` method, all its recorded funds are returned and its internal record reset to 0.

Listing 1: A simple banking contract in Solidity

```
1  contract Bank {
2    mapping(address => uint256) balances;
3    function deposit() public payable {
4      balances[msg.sender] = balances[msg.sender] + msg.value;
5    }
6    function withdraw() public {
7      uint256 bal = balances[msg.sender];
8      balances[msg.sender] = 0;
9      msg.sender.transfer(bal);
10   }
11 }
```

Although simple, the example already demonstrates several specialties of the Solidity programming language. In particular, every contract has access to an *internal* balance (not to confuse with mapping `balances` in line 2 which is simply a member variable of the contract). Funds can be transferred to and from this internal balance either explicitly (using function `transfer`) or implicitly (via an external method call). The address of the client (the one calling a contract's method) and the amount of funds transferred with a call can be accessed using keywords `msg.sender` and `msg.value`, respectively.

3 Isabelle/Solidity

Our semantics is based on the official documentation[2] of Solidity v0.8.25. It is formalized in higher-order logic (HOL) using inductive data types [7]. To this end, we use **bold** font for types and roman font for type constructors.

3.1 Storage Model

Solidity programs can make use of different types of stores for data: storage (for persistent data), memory (for volatile data), calldata (for volatile and read-

[2] https://docs.soliditylang.org/en/v0.8.25/

only data), and stack. Our storage model is formalized in theory `State.thy` and consists of a formalization of all of these different types of stores.

Value Types. The basic data types stored in Solidity are called value types and defined as follows:

$$\textbf{valtype} ::= \text{Bool}(\textbf{bool}) \mid \text{Sint}(\textbf{256 word}) \mid \text{Address}(\textbf{address}) \mid \text{Bytes}(\textbf{bytes}) \qquad (\texttt{valtype})$$

where **bool** is the type of boolean values, **256 word** the type of finite bit strings[3] of length 256, **address** the type of addresses, and **bytes** the type of bytes.

Storage. The persistent store can keep basic value types as well as complex types, such as arrays and hash-maps.

$$\textbf{sdata} ::= \text{Value}(\textbf{valtype}) \mid \text{Array}(\textbf{sdata list}) \mid \text{Map}(\textbf{valtype} \Rightarrow \textbf{sdata}) \qquad (\texttt{sdata})$$

where a **list** represents the type of lists over another type a.

Memory. The volatile store supports two types of data: basic value types and arrays.

$$\textbf{mdata} ::= \text{Value}(\textbf{valtype}) \mid \text{Array}(\textbf{location list}) \qquad (\texttt{mdata})$$

where **location** represents memory locations which is just a synonym for the type of natural numbers.

Note that, unlike storage, memory arrays are organized as pointer structures.

> *Example 3.1 (Memory array).* A simple array of 3 integers 5, 6, and 7, is represented as follows:
>
> $$[\text{Array}([1, 2, 3]), \text{Value}(\text{Sint}(5)), \text{Value}(\text{Sint}(6)), \text{Value}(\text{Sint}(7))]$$

Calldata. The read-only store can keep two types of data: basic value types and arrays.

$$\textbf{cdata} ::= \text{Value}(\textbf{valtype}) \mid \text{Array}(\textbf{cdata list}) \qquad (\texttt{cdata})$$

Stack. In Solidity, the stack can keep four different types of data: basic value types and pointers to storage, memory, or calldata, respectively.

$$\textbf{kdata} ::= \text{Value}(\textbf{valtype}) \mid \text{Storage}(\textbf{identifier}, \textbf{valtype list}) \mid \text{Memory}(\textbf{location}) \mid \text{Calldata}(\textbf{identifier}, \textbf{valtype list}) \qquad (\texttt{kdata})$$

Note that memory pointers are given in terms of memory locations while pointers to storage and calldata are given in terms of an identifier and a list of valtypes which represent the sequence of indices to reach the referenced element.

[3] Bit strings are modelled using Isabelle's word library [14]

State. A state of a Solidity program consists of configurations of each of its stores. To define such a state, we first introduce the following type abbreviations:

$$\mathbf{memory} ::= \mathbf{mdata\ list} \quad (\texttt{memory})$$
$$\mathbf{stack} ::= \mathbf{identifier} \rightharpoonup \mathbf{kdata} \quad (\texttt{stack})$$
$$\mathbf{storage} ::= \mathbf{address} \Rightarrow \mathbf{identifier} \Rightarrow \mathbf{sdata} \quad (\texttt{storage})$$
$$\mathbf{calldata} ::= \mathbf{identifier} \Rightarrow \mathbf{cdata} \quad (\texttt{calldata})$$
$$\mathbf{balances} ::= \mathbf{address} \Rightarrow \mathbf{nat} \quad (\texttt{balances})$$

where $A \rightharpoonup B$ is the set of finite maps from A to B. Note that the formalization of storage assigns a private store to each contract (identified by its address which is known during execution).

A state is now defined as a record over the above types:

$$\mathbf{state} \quad ::= \quad \mathbf{memory} \times \mathbf{calldata} \times \mathbf{storage} \times \mathbf{stack} \times \mathbf{balances} \quad (\texttt{state})$$

For a given state s, we denote with $\mathrm{Memory}(s)$, $\mathrm{Calldata}(s)$, $\mathrm{Storage}(s)$, $\mathrm{Stack}(s)$, $\mathrm{Balances}(s)$ its memory, calldata, storage, stack, and balances, respectively.

Auxiliary definitions. Our model provides definitions for different functions to lookup and update data for each type of store. In addition, we provide definitions for functions to copy data from memory to storage (`copy_memory_storage`), memory to calldata (`copy_memory_calldata`), storage to memory (`copy_storage_memory`), calldata to memory (`copy_calldata_memory`), and calldata to storage (`copy_calldata_storage`). These functions need to convert between the different representations of the data for each type of store.

3.2 State Monads

Statements and expressions are formalized using a state monad [12,42] with exception and non-termination. State monads are formalized in `State_Monad.thy`. To this end, we first define a result type as follows:

$$\mathbf{result}(\mathbf{n}, \mathbf{e}) \quad ::= \quad \mathrm{Normal}(\mathbf{n}) \mid \mathrm{Exception}(\mathbf{e}) \mid \mathrm{NT} \quad (\texttt{result})$$

The type **result** is defined over two type parameters, **n** and **e**, to denote the type of normal and exceptional return values, respectively. $\mathrm{Normal}(x)$ represents the result x of a normal execution, $\mathrm{Exception}(e)$ represents an exceptional return with exception e, and NT represents a non-terminating execution.

We can then define our state monad as follows:

$$\mathbf{state_monad}(\mathbf{a}, \mathbf{e}, \mathbf{s}) \stackrel{\mathrm{def}}{=} \mathbf{s} \Rightarrow \mathbf{result}(\mathbf{a} \times \mathbf{s}, \mathbf{e} \times \mathbf{s}) \quad (\texttt{state_monad})$$

The definition requires three type parameters: a type **a** for return values, a type **s** for states, and a type **e** for exceptions. The monad models an execution which starts in a certain state and either terminates in a new state with corresponding return value/exception, or does not terminate. We also define corresponding functions to create and execute monads, respectively (`create`, `execute`).

Monad operations. To support the construction of monads, we defined several basic operations on monads. For example, the following constants denote normal and erroneous returns, respectively:

$$\text{return}(a) \overset{\text{def}}{=} \lambda s.\ \text{Normal}(a, s) \qquad (\texttt{return})$$

$$\text{throw}(e) \overset{\text{def}}{=} \lambda s.\ \text{Exception}(e, s) \qquad (\texttt{throw})$$

The bind operator $f \gg\!= g$ combines a monad f: $\mathbf{state_monad}(\mathbf{a}, \mathbf{e}, \mathbf{s})$ with a function g: $\mathbf{a} \Rightarrow \mathbf{state_monad}(\mathbf{b}, \mathbf{e}, \mathbf{s})$:

$$f \gg\!= g\ (s) \overset{\text{def}}{=} \begin{cases} g(a)(s') & \textit{if } f(s) = \text{Normal}(a, s') \\ \text{Exception}(e, s) & \textit{if } f(s) = \text{Exception}(e, s) \\ \text{NT} & \textit{if } f(s) = \text{NT} \end{cases} \qquad (\texttt{bind})$$

To combine multiple monads we will often use the familiar do-notation. Then,

$$\mathbf{do} \left[\begin{array}{l} a \leftarrow \text{return}(1) \\ \text{return}(a) \end{array} \right.$$

denotes the monad given by $\text{return}(1) \gg\!= (\lambda a.\ \text{return}(a))$.

Partial functions. To model loops and recursive functions, we use Isabelle's partial function package [25]. This package resorts to domain theory to express general recursions over complete partial orders [44]. The package includes a setup for the option monad but can be extended to work with custom monads as well.

We implemented a new mode `sm` for the partial function package to work with our state monad. To this end, we provided a definition for a bottom element (`empty_result`), a corresponding ordering (`result_ord`), and a least upper bound (`result_lub`) for our state monad. Then, we needed to prove the following statement about admissible properties.

Lemma 3.1 (`admissible_sm`): For each proposition P, the following proposition is admissible w.r.t. `result_ord` and `result_lub`:

$$\lambda f.\ \forall x, s, r.\ \text{effect}(f(x), s, r) \implies P(x, h, r)$$

where $\text{effect}(m, s, r)$ defines that a state monad m, executed in state s either executes normally with result r or throws an exception r.

To support the partial function package in proving the existence of a unique least fixed-point, we also needed to prove monotonicity for some of our monads (`throw_monad_mono`, `return_monad_mono`, `bind_mono`, `option_monad_mono`).

3.3 Expressions

Solidity expressions and statements are formalized in terms of our state monad in `Solidity.thy`. To this end, we first define the result type of an expression as

follows:

$$\begin{aligned}\textbf{rvalue} \quad ::= \quad & \text{Storage}(\textbf{identifier}, \textbf{valtype list}) \mid \text{Memory}(\textbf{location}) \\ \mid\ & \text{Calldata}(\textbf{identifier}, \textbf{valtype list}) \mid \text{Value}(\textbf{valtype}) \mid \text{Empty} \quad (\texttt{rvalue})\end{aligned}$$

Thus, an expression can return a basic value type or a pointer to storage, memory, or calldata. In addition, an expression may also return nothing which is modelled by the constructor Empty. Now, we can define the expression monad as an instance of our state monad:

$$\textbf{expression_monad} \quad ::= \quad \textbf{state_monad}(\textbf{rvalue}, \textbf{ex}, \textbf{state})$$

where **ex** is a datatype for capturing the different types of errors.

Our formalization contains definitions for all the basic expressions in terms of the expression monad: basic constants (`bool_monad`, `sint_monad`, `address_monad`), arithmetic operators (`plus_monad`, `minus_monad`, `mult_monad`, `mod_monad`) as well as corresponding safe versions to check for overflows, comparison operators (`equals_monad`, `less_monad`), and boolean operators (`not_monad`, `and_monad`, `or_monad`).

In addition, we also provide definitions for Solidity specific operators, such as an operator to compute hash values (`keccak256_monad`), an operator to retrieve the address of the currently executing contract (`this_monad`), an operator to obtain the value which was sent by the sender (`value_monad`), and an operator to obtain the address of the message sender (`sender_monad`).

Finally, we provide monads to obtain the value of a storage or stack variable (`storeLookup`, `stackLookup`) and monads to compute the length of memory/storage arrays (`arrayLength`, `storeArrayLength`). Note that all the lookup functions take an optional list of expression monads to navigate complex data types such as arrays and hash-maps. In particular, the operators for stack variables need to check the location of the datatype (memory, storage, or calldata) and then navigate the structure in the corresponding store.

Example 3.2 (Stack lookup). Assuming st is a state with memory as defined in ex. 3.1 and $\text{Stack}(st)(\text{“}x\text{”}) = \text{Memory}(0)$. Then

$$\text{stackLookup}(\text{“}x\text{”}, [\text{sint_monad}(0)])(s) = \text{Normal}(\text{Value}(\text{Sint}(5)), s)$$

3.4 Statements

Again, we formalize all basic statements. Assignments, for example, are formalized by two monads to assign to stack (`assign_stack_monad`) or storage variables (`assign_storage_monad`). In Solidity, the behavior of assigning complex data types, such as arrays, depends on the location of the array. tab. 1 summarizes the different cases which need to be considered. A complex data type can be located in any of the three types of stores: memory (M), storage (S), and calldata (C). In addition, we can have pointers to storage elements on the stack (P). Depending on whether we copy the complete array or just modify a pointer,

we distinguish between deep copy (D) and shallow copies (S). For example, the first entry in tab. 1 states that assigning a memory array to a memory array actually just assigns a pointer instead of copying the whole array.

Solidity also supports dynamic storage arrays which can be expanded at runtime. To this end, we provide also a monad to allocate new storage for an additional array element (`allocate`).

Conditionals and loops are formalized by two monads (`cond_monad` and `while_monad`). As mentioned in sec. 3.2, loops are formalized using a partial recursive function. To this end, we first needed to verify some additional monotonicity results (`cond_mono`, `cond_K`) used by the partial function package to verify the existence of a corresponding fixed-point. The package then provides us with a fixed-point induction theorem which can be used to reason about loops.

Table 1. Assignments for complex data types

LHS	RHS	Type
M	M	S
M	C	D
M	P	D
M	S	D
S	M	D
S	C	D
S	S	D
S	P	D
P	S	S
P	M	N/A
P	C	N/A
P	P	N/A
C	_	N/A

Finally, we provide a formalization of the transfer statement which can be used to transfer funds to other accounts (`transfer_monad`). Note that in Solidity, a transfer implicitly triggers the execution of a so-called fallback method and thus we postulate the existence of such a method for each contract address. In addition, we assume that such methods preserve invariants over the currently executing contract's storage and balance, given that we are able to show that a callback preserves invariants as well:

$$\frac{\forall s, r.\, P(s) \wedge \text{effect}(\mathit{call}, s, r) \Longrightarrow \text{inv}(r, P, E) \qquad P(s)}{\forall a, r.\, \text{effect}(\text{fallback}(a), s, r) \Longrightarrow \text{inv}(r, P'(s), E'(s))} \; (\texttt{fallback_invariant})$$

where $\text{inv}(r, P, E)$ requires that predicate P holds for a successful termination r and E for an exceptional one. Moreover,

$$P'(s) = \lambda s'.\, \text{Stack}(s) = \text{Stack}(s') \wedge \text{Memory}(s) = \text{Memory}(s') \\ \wedge \text{Calldata}(s) = \text{Calldata}(s') \wedge P(s')$$

and E' is defined similarly as E.

3.5 Automation: Specification

To support a user in the specification and verification of a contract, we implemented a new definitional package for Isabelle using Isabelle/ML [43]. Our package is implemented in file `Contract.thy`.

Specification. To support a user with the specification of a contract, our package implements a new Isabelle command `contract`. This command requires a user to provide a *name* for the contract, followed by a list of declarations of storage variables. Then, the user can specify a constructor using keyword

constructor, followed by an implementation of the constructor in terms of an expression monad. Finally, the user can specify a list of methods using the keyword cmethod. Each method can have an optional list of formal parameters followed by an implementation in terms of an expression monad.

Example 3.3 (Banking contract). The formalization of our example contract from lst. 1 in Isabelle/Solidity looks as follows:

```
14 contract Bank
15   for "STR ''balances''": "sdata.Map (mapping (sdata.Value sint))"
16
17 constructor where
18   "skip_monad"
19
20 cmethod deposit where
21   "assign_storage_monad (STR ''balances'') [sender_monad]
22     (plus_monad_safe (storeLookup (STR ''balances'') [sender_monad]) (value_monad))" ,
23
24 cmethod withdraw where
25   "do {
26     decl (STR ''bal'') (storeLookup (STR ''balances'') [sender_monad]);
27     assign_storage_monad (STR ''balances'') [sender_monad] (sint_monad 0);
28     transfer_monad (sender_monad) (stackLookup (STR ''bal'') [])
29   }"
```

Our package then generates definitions for the constructor and the contract's methods. To this end, we first initialize the contract's balance, storage variables, and stack. Then, we can just execute the monad provided by the user.

Example 3.4 (Constructor). Our package would generate the following definition for the constructor from the specification given in ex. 3.3:

$$\text{bank_constructor_def} =$$

$$\textbf{do} \left[\begin{array}{l} \text{init_balance} \\ \text{initStorage}(\text{“}\textit{balances}\text{”}, \text{Map}(\text{mapping}(\text{Value}(\text{Sint})))) \\ \text{emptyStack} \\ \text{skip_monad} \end{array} \right.$$

The definition of the methods is a bit more elaborate since we want to allow for mutual recursive definitions. Again, we first initialize the balance and the stack, before we execute the monadic definition provided by the user. Note, however, that a method has a special parameter *call* which can be used in its definition. This is a construct which can be used to call internal or external methods. Note also that for each method we need to generate and prove a monotonicity lemma for the partial function package.

Example 3.5 (Methods). Our package would generate the following definition for the deposit method from the specification given in ex. 3.3:

$$\text{deposit}(\textit{call}) =$$

$$\textbf{do} \left[\begin{array}{l} \text{init_balance} \\ \text{emptyStack} \\ \text{assign_storage_monad}(\text{“}\textit{balances}\text{”}, [\text{sender_monad}], \\ \text{plus_safe}(\text{storeLookup}(\text{“}\textit{balances}\text{”}, [\text{sender_monad}], \text{sender_monad}))) \end{array} \right.$$

Our package would also generate a proof for the following lemma:

$$\text{sm.mono_body}(\text{deposit}) \qquad (\texttt{deposit_def_mono})$$

where $\text{sm.mono_body}(x)$ requires definition x to be monotonic w.r.t. the state monad ordering described in sec. 3.2.

Finally, we generate a wrapper datatype for all methods of a contract and a corresponding definition for a function call over this data type. Since we allow for the specification of recursive methods, the definition of call uses the partial function package. The partial function package then provides us with a proof method to verify properties over (mutually recursive) functions using fixed-point induction (`call.raw_induct`).

Example 3.6 (Wrapper). Our package would generate the following wrapper for the specification given in ex. 3.3.

$$\textbf{bank} \ ::= \ \text{Deposit_m} \mid \text{Withdraw_m} \qquad (\texttt{bank})$$

Moreover, the definition of call looks as follows:

$$\text{bank_call}(m) = \begin{cases} \text{deposit}(\text{bank_call}) & \text{for } m = \text{Deposit_m} \\ \text{withdraw}(\text{bank_call}) & \text{for } m = \text{Withdraw_m} \end{cases}$$

Verification. To support a user with the verification of contract invariants, our package implements a new Isabelle command `invariant`. This command requires a user to provide the name of the contract, followed by a *name* for the invariant and two predicates formulated over the contracts private state and internal balance (one for normal executions and the other one for exceptional ones). The command then provides the user with a series of proof obligations which they need to discharge to verify the invariant.

Example 3.7 (Invariant). Assume that we want to verify that the sum of all registered deposits of our banking contract is always less or equal to the contract's internal balance:

$$\text{sum_bal}(s, b) = b \geq \sum_{ad} (s(\text{``}balances\text{''})(ad)) \qquad (1)$$

The corresponding specification in Isabelle/Solidity would then look as follows:

```
215 invariant sum_balances: sum_balances sum_balances for Bank
216   apply -
217   using constructor_sum_balances apply blast
218   using deposit_sum_balances apply blast
219   using withdraw_sum_balances apply blast
220   done

proof (prove)
goal (1 subgoal):
 1. (effect bank_constructor s r ⟹ local.inv r (inv_state sum_balances) (inv_state sum_balances)) &&&
    ((⋀x. ∀s r. inv_state sum_balances s ∧ effect (callx x) s r ⟶
                  local.inv r (inv_state sum_balances) (inv_state sum_balances)) ⟹
     effect (deposit callx) s r ⟹
     inv_state sum_balances s ⟶ local.inv r (inv_state sum_balances) (inv_state sum_balances)) &&&
```

In the output window we can also see the proof obligations for the constructor and the deposit method which are then discharged by dedicated lemmas.

The invariant command requires the user to verify that the constructor establishes the invariant and that the methods preserve it. After discharging the proof obligations, our package then derives a general theorem which shows that the contract indeed preserves the invariant for all possible environments.

Example 3.8 (Proof obligations). The proof obligation for the deposit method generated from the specification provided in ex. 3.7 looks as follows:

$$\text{effect}(\text{deposit}(\mathit{call}), s, s', r) \land \text{inv}(\text{sum_bal}, s) \implies \text{inv}(\text{sum_bal}, s')$$

Once the user discharges the proof obligations, our package then generates a proof for the following theorem:

Theorem 1 (`bank.sum_bal`).

$$\text{effect}(\text{call}(x'), s, s', r, e) \land \text{inv}(\text{sum_bal}, s) \implies \text{inv}(\text{sum_bal}, s')$$

The proof of the final theorem generated by our package uses the fixed-point induction theorem provided by the partial function package (`call.raw_induct`) and uses the lemmas provided by the user to discharge the different proof obligations required by the induction theorem.

3.6 Automation: Verification

To support a user in discharging the proof obligations generated by our package, we implemented a weakest precondition calculus [15] and corresponding verification condition generator for our monads in theory `WP.thy`.

The definition of the weakest precondition for our state monad is as follows:

$$\text{wp}(f, P, E, s) = \begin{cases} P(r, s') & \text{if execute}(f, s) = \text{Normal}(r, s') \\ E(e, s') & \text{if execute}(f, s) = \text{Exception}(e, s') \\ \text{True} & \text{if execute}(f, s) = \text{NT} \end{cases} \qquad (\texttt{wp})$$

Note that we are only interested in partial correctness here (since smart contracts are always guaranteed to terminate). Thus, the weakest precondition of a non-terminating program is just True. We then prove corresponding lemmas for all our monads (in total ca. 100 lemmas).

We use Isabelle/Eisbach [32] to define the verification condition generator. Its purpose is to reduce the proof obligations to propositions in plain Isabelle/HOL which can then be proven using the proof methods available for Isabelle/HOL.

Example 3.9 (Verifying deposit). The proof for the deposit method (discussed in ex. 3.8) using our VCG is shown below:

```
lemma deposit_sum_balances:
   fixes call:: "bank ⇒ (unit, ex, state) state_monad"
  assumes "effect (deposit call) s s' () e"
      and "inv_state sum_balances s"
    shows "inv_state sum_balances s'"
  using assms
  apply (erule_tac effect_wp)
  unfolding deposit_def inv_state_def
  by (vcg | auto simp add:1 | rule bal_msg_sender, assumption)+
```

We first unfold some definitions, and then we just combine the VCG with the standard method *auto* which is able to finish the proof.

4 Evaluation

4.1 Compliance and Unit Testing

An important problem in any formalization of a real-world (programming) language is to ensure that the formal semantics is a faithful representation of the real-world implementation, in our case, Solidity on the Etherum Blockchain. Furthermore, when developing the formal semantics further, for instance, adding new features, it is important to ensure that no unwanted changes are introduced.

We address this issue by translating, currently manually, the test cases from the reference test suite of Solidity [16] to our formalization (the test cases are implemented in file `Unit_Tests.thy`). Our semantics is executable for a finite address space. Thus, if we instantiate the generic address type (recall sec. 3.1) with a finite address space (e.g., $1, \ldots, 100$), we can use Isabelle's code generator [18] to efficiently evaluate (ground) test cases. This allows us to explore the semantics using Isabelle's `value` statement as well as turn the test cases from [16] into formal lemmas.

Example 4.1 (Unit test). The following shows a simple unit test for the *assign_stack* monad.

```
lemma "is_Normal (execute (do {
        init (STR ''x'') (kdata.Value (Sint 0));
        init (STR ''y'') (kdata.Value (Sint 0));
        contract_assign_stack_monad (STR ''y'') [] (sint_monad 1);
        assert_monad (equals_monad (contract_stackLookup (STR ''x'') []) (sint_monad 0));
        assert_monad (equals_monad (contract_stackLookup (STR ''y'') []) (sint_monad 1))
      }) emptyState)"
  by (normalization, simp?)
```

All such "test case"-lemmas are proven by the `normalization` method, potentially followed by simplification. The `normalization` method uses a combination of simplification and code generation for a small, trusted subset of HOL [18]. We formalized over **50** test cases; their checking takes less than **30** seconds.

4.2 Case Studies

To evaluate the potential of the approach for verifying Solidity smart contracts we used it in three case studies.

Banking. For the first case study, we chose the banking contract shown in lst. 1 since it is also verified in the deep embedding [28,30,31] and thus allows us to compare the two approaches. We specified the contract in our framework and then verified the invariant discussed in ex. 3.7 (`sum_bal`).

The formalization is available in theory `Token.thy` and the effort of specifying and verifying the contract in Isabelle/Solidity is summarized in fig. 1. The complete specification of the banking contract is just 11 lines of Isar code and consists of a specification of the constructor as well as the deposit and withdraw methods. It includes several basic features of Solidity, such as mappings and transfer statements. The verification of the invariant required to verify 70 lines of Isar code. One particular challenge in the verification was the fact that the invariant is a property formulated over the complete address space, which is a finite but arbitrarily large set.

Compared to previous verification attempts we can see a large reduction in effort. For example, the original verification of the very same invariant for the same contract was 3 000 lines of Isar [28,30]. Even using a weakest precondition calculus still required 700 lines of proof code [31].

Casino. For the second case study we chose the casino contract used in the previous "VerifyThis long-term verification challenge". The contract is available online[4]. The contract allows an operator to create a new game by placing a hidden secret in the form of a hash of a secret number inside the contract (using method `createGame`). A player can then place a bet (HEAD or TAIL) by transferring funds to the contract via the `placeBet` method. The operator can then invoke `decideBet` to start the game. To this end, the operator reveals the secret number to decide if HEAD (even) or TAIL (odd) wins. In the case the player wins, they get twice their bet back and the pot is deduced accordingly. In the case the player loses, their original bet is added to the pot. The operator can always increase the pot or deduce some amount (if no game is active).

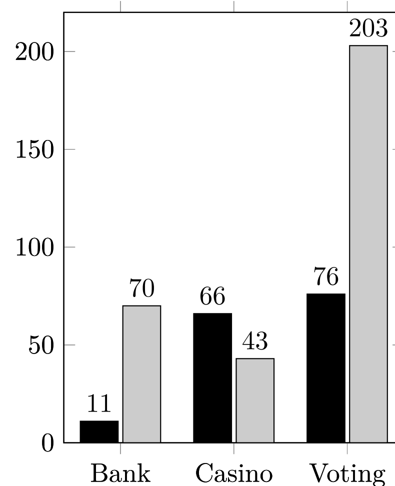


Fig. 1. Specification (black) and verification (grey) effort in terms of lines of proof code.

One important property, from the player's perspective, is to ensure that the casino has always enough funds to cover the pot. Thus, for our case study, we

[4] https://verifythis.github.io/02casino/

verified the following invariant:

$$\text{pot_balance}(s, b) = b \geq s(\text{``}pot\text{''}) \qquad (\texttt{pot_balance})$$

The formalization of the casino contract is available in `Casino.thy` and the specification and verification effort is summarized in fig. 1. Its specification in Isabelle/Solidity required 66 lines of code in total and consists of 7 member variables and 6 methods. It contains some advanced features of Solidity, such as enums, modifiers and hash functions.

The verification of the invariant was just 43 lines of code. The invariant is trivial for method `createGame`, since it does not modify the balance of the contract or the pot. However, all other methods modify either the balance or the value of the pot (or both), and thus require to show that the invariant is preserved.

Voting. For the third case study we chose a voting contract from the official Solidity documentation which states that "The following contract is quite complex, but showcases a lot of Solidity's features."

The contract is available online[5]. The idea is to create one contract per ballot, providing a short name for each option. Then the creator of the contract who serves as chairperson will give the right to vote to each address individually by calling the `giveRightToVote` method. The persons behind the addresses can then choose to either vote themselves or to delegate their vote to a person they trust. To vote themselves, they can invoke method `vote` by providing the number of the proposal. To delegate their vote, they can invoke method `delegate` and provide the address of the person they want to delegate to. At the end of the voting time, method `winningProposal` can be called to compute the proposal with the largest number of votes and declare it the winning proposal.

We then verified an invariant which guarantees that the number of votes is always bound by the number of eligible voters:

$$\text{sum_votes}(s, b) = \sum_{i \leq \#s(\text{``}proposals\text{''})} (s(\text{``}proposals\text{''})[i].\text{``}voteCount\text{''}) \leq \sum_{ad \in \{ad \mid s(\text{``}voters\text{''})(ad).\text{``}voted\text{''}\}} (s(\text{``}voters\text{''})(ad).\text{``}weight\text{''}) \qquad (\texttt{sum_votes})$$

The formalization is available in `Voting.thy` and the specification and verification effort is summarized in fig. 1. Specification of the voting contract required 76 lines of Isabelle/Isar in total split across 5 methods. It showcases many advanced features of Solidity, such as hash maps, dynamic storage arrays, memory arrays, storage pointers and loops. Verifying the invariant required 200 lines of proof code. One particular challenge in the verification of this invariant was that the invariant was formulated over complex data structures which required advanced reasoning capabilities.

[5] https://docs.soliditylang.org/en/v0.8.25/solidity-by-example.html#voting

5 Related Work

The work presented in this paper is about the verification of Solidity smart contracts using Isabelle. Thus, related work can be found in two different areas: First, work related to the verification of Solidity and work related to general program verification using Isabelle. We discuss other works formalizing Solidity in Isabelle at the end of this section.

As outlined by Almakhour et al. [3] and Tolmach et al. [40], there is a growing amount of research about verification of smart contracts. Early work in this area was done by Bhargavan et al. [8] which describe an approach to map a Solidity contract to F* where it can then be verified. TinySol [6] and Featherweight Solidity[13], on the other hand, are two calculi formalizing some core features of Solidity. Ahrendt and Bubel describe SolidiKeY [2], a formalization of a subset of Solidity in the KeY tool [1] to verify data integrity for smart contracts. Hajdu and Jovanovic [19,20], provide a formalization of Solidity in terms of a simple SMT-based intermediate language and Tai et al. [35] provide an encoding of Solidity for Z3. Finally, and Jiao et al. [21,22], provide a formalization of Solidity in $\mathbb{K}$. All of these works use an axiomatic approach for defining the formal semantics of Solidity. In contrast, we use a conservative embedding into Isabelle/HOL, which ensures the consistency of our semantics "by construction".

Formalizing programming languages or specification languages in Isabelle is by no means a new technique. Over time, several languages and tools have been developed along this line, such as Isabelle/SIMPL [39], IMP [36,26] and the seL4 verification project [24]. Also, using a monadic representation of stateful computations is common. For example, the AutoCorres [17,9] (used in the seL4 verification project [24]) provides an abstraction of C code using monads, and also Clean [41], does use monads for modelling stateful computations. While work in this area provides support for many general features of program verification, they do not support specific features for Solidity.

To the best of our knowledge there are only two other formalizations of Solidity in Isabelle. Closely related is our own work on a deep embedding of Solidity in Isabelle [28]. In our deep embedding, Solidity statements are formalized as a dedicated datatype which allows to verify properties about the language itself. In general, it is also possible to use the deep embedding for the verification of concrete contracts. However, this requires quite some effort even for rather small contracts. This paper, on the other hand, provides a shallow embedding of Solidity in which statements are directly mapped to corresponding HOL definitions. This allows for better automation and can significantly reduce the effort to verify concrete contracts as shown in our first case study (recall sec. 4.2). Thus, both approaches complement each other, and it depends on the concrete verification task which approach is better.

Ribeiro et al. [37]. adapt the Simpl language [38] to formalize a subset of Solidity in Isabelle/HOL. Compared to our work, SOLI is quite low-level and rather an intermediate language than a direct formalization of Solidity. Indeed, while it supports several features which are not provided by Solidity (Upd, Dyncom), it does not provide explicit support for Solidity-specific language features, such

as different types of stores, a notion of Gas, fallback methods, external vs. internal functions, etc. Another difference to our work is that their semantics seems to be not executable and therefore difficult to evaluate. On the other hand, we considered it important to have an executable semantics that can be evaluated against the reference implementation.

6 Conclusion

With this paper, we present a verification environment for Solidity smart contracts based on Isabelle/HOL. The approach comes with a novel formalization of the Solidity storage model and a corresponding shallow embedding of Solidity expressions and statements. In addition, we implemented two new Isabelle commands to support the specification and verification of Solidity contracts in Isabelle. Finally, we formalized a weakest precondition calculus and implemented a corresponding verification condition generator to support a user in the verification. To validate our semantics, we implemented a test suite consisting of over 50 unit tests. Finally, we evaluated the approach by means of three case studies.

Our results show that the approach can be used to verify even complex contracts with reasonable effort. In addition, it significantly reduces the verification effort compared to a corresponding deep embedding. However, the case studies also revealed some limitations of the approach which lead to future work.

One limitation regards compliance of our semantics with the official Solidity documentation. Even though the unit tests provide some evidence that our semantics indeed matches the specification, we cannot guarantee full compliance. While it will never be possible to guarantee complete compliance, future work should provide further evidence in that regard. A promising approach would be to use semantic fuzzing as for example described in [29].

Another limitation concerns proof automation. Although the weakest precondition calculus and corresponding verification condition generator automate large parts of the proofs, there is still area for improvement. For example, some of our rules are rather general and split a goal into several subgoals where only some are indeed sensible for a given context. Thus, future work should focus on improving such rules by inspecting the context and apply more specialized rules for the given situation.

Acknowledgments. We thank the anonymous reviewers of SEFM 2024 for their careful reading and constructive comments to improve earlier versions of this paper. This work was supported by the Engineering and Physical Sciences Research Council [grant number EP/X027619/1].

References

1. Ahrendt, W., Beckert, B., Bubel, R., Hähnle, R., Schmitt, P.H., Ulbrich, M.: Deductive software verification–the KeY book, vol. LNCS 10001. Springer (2016)
2. Ahrendt, W., Bubel, R.: Functional verification of smart contracts via strong data integrity. In: Margaria, T., Steffen, B. (eds.) Leveraging Applications of Formal Methods, Verification and Validation: Applications. pp. 9–24. Springer (2020)
3. Almakhour, M., Sliman, L., Samhat, A.E., Mellouk, A.: Verification of smart contracts: A survey. Pervasive and Mobile Computing **67**, 101227 (2020)
4. Azaria, A., Ekblaw, A., Vieira, T., Lippman, A.: MedRec: using blockchain for medical data access and permission management. In: 2016 2nd International Conference on Open and Big Data (OBD). pp. 25–30 (2016)
5. Bahrynovska, T.: History of Ethereum Security Vulnerabilities, Hacks and Their Fixes (2017)
6. 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.) Data Privacy Management, Cryptocurrencies and Blockchain Technology. pp. 233–243. Springer (2019)
7. Berghofer, S., Wenzel, M.: Inductive datatypes in HOL — lessons learned in formal-logic engineering. In: Bertot, Y., Dowek, G., Théry, L., Hirschowitz, A., Paulin, C. (eds.) TPHOLs. pp. 19–36. Springer (1999)
8. Bhargavan, K., Delignat-Lavaud, A., Fournet, C., Gollamudi, A., Gonthier, G., Kobeissi, N., Kulatova, N., Rastogi, A., Sibut-Pinote, T., Swamy, N., Zanella-Béguelin, S.: Formal verification of smart contracts: Short paper. In: Programming Languages and Analysis for Security. pp. 91–96. PLAS, ACM (2016)
9. Brecknell, M., Greenaway, D., Hölzl, J., Immler, F., Klein, G., Kolanski, R., Lim, J., Norrish, M., Schirmer, N., Sickert, S., Sewell, T., Tuch, H., Wimmer, S.: Autocorres2. Archive of Formal Proofs (April 2024)
10. Chavez-Dreyfuss, G.: Sweden tests blockchain technology for land registry (2016)
11. CipherTrace: Cryptocurrency crime and anti-money laundering report. Tech. rep., Mastercard (2021)
12. Cock, D., Klein, G., Sewell, T.: Secure microkernels, state monads and scalable refinement. In: Mohamed, O.A., Muñoz, C., Tahar, S. (eds.) TPHOLs. pp. 167–182. Springer (2008)
13. Crafa, S., Di Pirro, M., Zucca, E.: Is Solidity solid enough? In: Bracciali, A., Clark, J., Pintore, F., Rønne, P.B., Sala, M. (eds.) Financial Cryptography and Data Security. pp. 138–153. Springer (2020)
14. Dawson, J.: Isabelle theories for machine words. ENTCS **250**(1), 55–70 (2009), int. Workshop on Automated Verification of Critical Systems (AVoCS 2007)
15. Dijkstra, E.W.: Guarded commands, nondeterminacy and formal derivation of programs. Commun. ACM **18**(8), 453–457 (aug 1975)
16. Foundation, E.: Solidity semantic reference test suite (version 0.8.25). https://github.com/ethereum/solidity/tree/v0.8.25/test/libsolidity/semanticTests (2023)
17. Greenaway, D., Andronick, J., Klein, G.: Bridging the gap: Automatic verified abstraction of C. In: Beringer, L., Felty, A. (eds.) ITP. pp. 99–115. Springer (2012)
18. Haftmann, F., Bulwahn, L.: Code generation from Isabelle/HOL theories (2023), http://isabelle.in.tum.de/doc/codegen.pdf
19. Hajdu, Á., Jovanovic, D.: solc-verify: A modular verifier for Solidity smart contracts. In: Chakraborty, S., Navas, J.A. (eds.) VSTTE. LNCS, vol. 12031, pp. 161–179. Springer (2019)
20. Hajdu, Á., Jovanovic, D.: SMT-friendly formalization of the Solidity memory model. In: Müller, P. (ed.) ESOP. LNCS, vol. 12075, pp. 224–250. Springer (2020)

21. Jiao, J., Kan, S., Lin, S.W., Sanan, D., Liu, Y., Sun, J.: Semantic understanding of smart contracts: executable operational semantics of Solidity. In: SP. pp. 1695–1712. IEEE (2020)
22. Jiao, J., Lin, S.W., Sun, J.: A generalized formal semantic framework for smart contracts. In: Wehrheim, H., Cabot, J. (eds.) FASE. pp. 75–96. Springer (2020)
23. Kelly, J.: Banks adopting blockchain 'dramatically faster' than expected: IBM (2016)
24. Klein, G., Andronick, J., Elphinstone, K., Heiser, G., Cock, D., Derrin, P., Elkaduwe, D., Engelhardt, K., Kolanski, R., Norrish, M., Sewell, T., Tuch, H., Winwood, S.: seL4: formal verification of an operating-system kernel. Commun. ACM **53**(6), 107–115 (2010)
25. Krauss, A.: Recursive definitions of monadic functions. ENTCS **43**, 1–13 (2010)
26. Lammich, P., Wimmer, S.: IMP2 – simple program verification in Isabelle/HOL. Archive of Formal Proofs (2019)
27. Llama, D.: TVL breakdown by smart contract language (2024), https://defillama.com/languages
28. Marmsoler, D., Brucker, A.D.: A denotational semantics of Solidity in Isabelle/HOL. In: Calinescu, R., Pasareanu, C. (eds.) SEFM. LNCS 13085, Springer (2021)
29. Marmsoler, D., Brucker, A.D.: Conformance testing of formal semantics using grammar-based fuzzing. In: Kovacs, L., Meinke, K. (eds.) Tests And Proofs. LNCS 13361, Springer-Verlag (2022)
30. Marmsoler, D., Brucker, A.D.: Isabelle/Solidity: a deep embedding of solidity in Isabelle/HOL. Archive of Formal Proofs (July 2022)
31. Marmsoler, D., Thornton, B.: SSCalc: a calculus for solidity smart contracts. In: Ferreira, C., Willemse, T.A.C. (eds.) Software Engineering and Formal Methods. pp. 184–204. Springer, Cham (2023)
32. Matichuk, D., Murray, T., Wenzel, M.: Eisbach: A proof method language for isabelle. Journal of Automated Reasoning **56**, 261–282 (2016)
33. Nakamoto, S.: Bitcoin: A Peer-to-Peer Electronic Cash System (2008)
34. News, B.: Hackers steal $600m in major cryptocurrency heist (2021)
35. Nguyen, T.D., Pham, L.H., Sun, J., Le, Q.L.: An idealist's approach for smart contract correctness. In: Li, Y., Tahar, S. (eds.) Formal Methods and Software Engineering. pp. 11–28. Springer (2023)
36. Nipkow, T.: Winskel is (almost) right: Towards a mechanized semantics textbook. Formal Aspects of Computing **10**(2), 171–186 (1998)
37. Ribeiro, M., Adão, P., Mateus, P.: Formal Verification of Ethereum Smart Contracts Using Isabelle/HOL, pp. 71–97. LNCS 12300, Springer (2020)
38. Schirmer, N.: Verification of Sequential Imperative Programs in Isabelle/HOL. Ph.D. thesis, Technische Universität München (2006)
39. Schirmer, N.: A sequential imperative programming language syntax, semantics, hoare logics and verification environment. Archive of Formal Proofs (Feb 2008)
40. Tolmach, P., Li, Y., Lin, S.W., Liu, Y., Li, Z.: A survey of smart contract formal specification and verification. ACM Comput. Surv. **54**(7) (jul 2021)
41. Tuong, F., Wolff, B.: Clean – an abstract imperative programming language and its theory. Archive of Formal Proofs (Oct 2019)
42. Wadler, P.: Monads for functional programming. In: Broy, M. (ed.) Program Design Calculi. pp. 233–264. Springer (1993)
43. Wenzel, M.: The Isabelle/Isar implementation (2013)
44. Winskel, G.: The formal semantics of programming languages: an introduction. MIT press (1993)
45. YCharts.com: Ethereum transactions per day (2024)
46. Yurcan, B.: How blockchain fits into the future of digital identity (2016)