Isabelle/Solidity: A Tool for the Verification of Solidity Smart Contracts (tool paper)

Asad Ahmed
University of Exeter, Exeter EX4 4PY, UK

Diego Marmsoler
University of Exeter, Exeter EX4 4PY, UK

Abstract

Smart contracts are an important innovation in Blockchain which allow to automate financial transactions. Every day, hundreds of thousands of new contracts are deployed managing millions of dollars' worth of transactions. Thus, bugs in smart contracts may lead to high financial losses and it is important to get them right before deploying them to the Blockchain. To address this problem we developed Isabelle/Solidity, a tool for the verification of smart contracts in Isabelle. The tool is implemented as a definitional extension for the Isabelle proof assistant and thus complements existing tools in this area which are mostly based on axiomatic approaches. In this paper we describe Isabelle/Solidity and demonstrate it by verifying a casino contract from the VerifyThis long term verification challenge.

2012 ACM Subject Classification Security and privacy → Logic and verification

Keywords and phrases Program Verification, Smart Contracts, Isabelle, Solidity

Digital Object Identifier 10.4230/OASIcs.CVIT.2016.23

Funding This work was supported by the Engineering and Physical Sciences Research Council [grant number EP/X027619/1]

1 Introduction

One important innovation which comes with Blockchain are so-called *smart contracts*. These are digital contracts which are automatically executed once certain conditions are met and which are used to automate transactions on the Blockchain. For instance, a payment for an item might be released instantly once the buyer and seller have met all specified parameters for a deal. Every day, hundreds of thousands of new contracts are deployed [9] managing millions of dollars' worth of transactions [16].

Technically, a smart contract is *code which is deployed to a Blockchain* and which can be executed by sending special transactions to it. Thus, as for 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. In general, it is estimated that since 2019, more than $5B was stolen due to vulnerabilities in smart contracts [5].

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 tool for the deductive verification of Solidity smart contracts implemented as a definitional extension for the Isabelle [14] proof assistant. The main contributions of this paper are:

- A practical and user-friendly tool equipped with concrete syntax to support formal specification of Solidity smart contracts. Moreover, invariant and verification keywords allow users to specify and verify the invariant and post-conditions, respectively, for smart contracts
- We provide an overview of the technical architecture of Isabelle/Solidity.

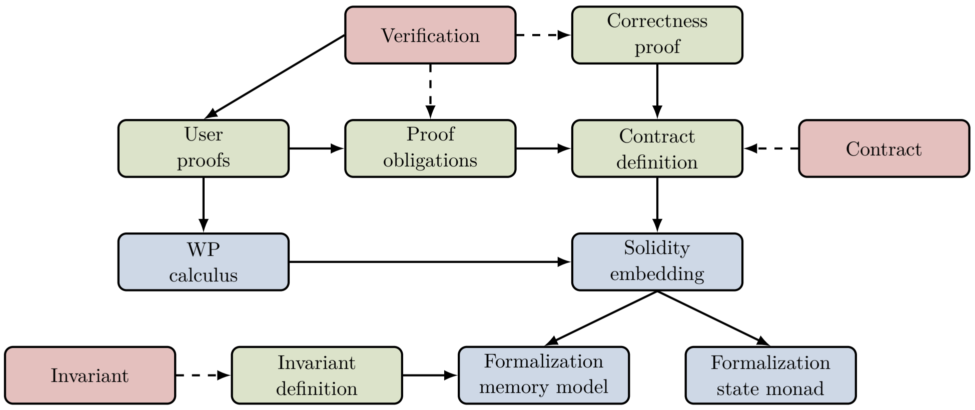

Figure 1 Isabelle/Solidity: theories (blue), commands (red), and generated artefacts (green).

- We use it to verify the casino contract from the VerifyThis verification challenge.

2 Isabelle/Solidity

Isabelle/Solidity is available as a definitional package for the Isabelle proof assistant. It is available online and can be downloaded from `www.marmsoler.com/assets/isabelle-solidity.zip`

Figure 1 shows the architecture of Isabelle/Solidity. Isabelle/Solidity is based on four Isabelle theories which are represented by the blue rectangles:

Formalization state monad We model Solidity programs as functions which manipulate states. Such functions are usually described in terms of state monads [15] and this theory contains our formalization of it based on [6].

Formalization memory model Solidity has a quite particular memory model with different types of stores which support different types of data structures (See Table 1). This theory contains our formalization of the Solidity memory model. It also defines the notion of a Solidity state as a collection of different stores.

Solidity embedding A Solidity statement is defined as a particular state monad manipulating a Solidity state. This theory contains definitions for all of our Solidity statements.

WP calculus To support a user in the verification of Solidity programs, Isabelle/Solidity comes with a verification condition generator (VCG). The VCG is based on a weakest precondition calculus [8] for our Solidity statements and which is formalized in this theory.

In addition to the theories described above, Isabelle/Solidity extends the Isabelle proof assistant by means of three new definitional commands represented by the red rectangles in Figure 1. Each of the commands generates lower-level definitions and theorems for Isabelle/HOL which are represented as green rectangles.

Contract This command allows a user to specify a new contract. It requires them to specify a list of member variables, a constructor, and a list of methods. For each method the user can specify a list of parameters as well as a method body. The body is provided

Table 1 Data Locations in Solidity

	Stack	**Storage**	**Memory**	**Calldata**
Persistence	Temporary	Permanent	Temporary	Temporary
Mutable	No	Yes	No	No
Scope	Local to function	Global to contract	Local to contract	local to contract

as a state monad using the monads defined in the corresponding Isabelle theory. The command then generates definitions for the *contract's methods*. To this end each method is mapped to a corresponding partial function definition [11].

Invariant This command supports a user in the specification of an invariant for the contract. An invariant is specified as a HOL formula over the store and balance of the contract. The command then uses the typing information of member variables to generate a corresponding *invariant definition*.

Verification This command triggers the verification of a contract. It requires a user to provide an invariant as well as postconditions for the methods. Then it presents the user with a list of *proof obligations* for each of the contract's methods. The users then need to discharge these proof obligations by providing a corresponding *proof*. To this end they can use the WP calculus provided by our framework. After discharging these obligations, Isabelle/Solidity proves an overall *correctness theorem* which guarantees that the invariant as well as postconditions are not violated.

3 Case Study

To demonstrate Isabelle/Solidity in action, we use it to verify the casino contract from the VerifyThis long-term verification challenge [1]. The version of the contract used here is provided in Appendix A. The casino contract implements a betting game based on guessing the outcome of coin-tossing. At any time, the game is characterized by three states, i.e., `IDLE`, `GAME_AVAILABLE` and `BET_PLACED`. Initially, the game is in the `IDLE` state. The game has been implemented in Solidity using the following functions.

- The operator can create a new game by calling the `creatGame` function and provide the hash value of a secret number. The function stores the hash value and changes the state to `GAME_AVAILABLE`.
- Once in state `GAME_AVAILABLE`, a player can place a bet by invoking the `placeBet` function and provide a guess (`HEADS` or `TAILS`). This function stores the player's address, its guess, and the amount of the bet, and changes the state to `BET_PLACED`.
- Now, the operator can decide the bet anytime by calling the `decideBet` function and providing the secret number. The secret number is then used to decide the outcome of the coin tossing (`HEADS` or `TAILS`). If the player's guess is correct, the double of the amount of the bet is transferred to their address. Otherwise, the amount equal to the bet is added to the pot. The function also changes the state of the game to `IDLE`.
- The operator may add money to the bet, at anytime, using `addToPot` but can only remove money if the game is not in state `BET_PLACED` by calling `removeFromPot`.

The contract logic is aimed for fairness, security and integrity. The unique hash value corresponding to the secrete number serves as a safeguard from the potential manipulation of game owner when revealing the bet. Also a positive-integer type secret number introduces a degree of randomness due to a large domain. This results in a significant challenge for player to predict the hash value and hence game outcome.

4 Specification

To verify a smart contract in Isabelle/Solidity we first need to specify it. To this end Isabelle/Solidity provides the `contract` command which supports a user in this task. The command requires a user to specify a list of storage variables and corresponding types, followed by a specification of the constructor and the contract's methods.

Listing 1 shows the specification of storage variables for the casino contract in Isabelle/Solidity.

Listing 1: Isabelle/Solidity data types for Casino

```
contract Casino
  for state: StateT
  and operator: AddressT
  and player: AddressT
  and pot: IntT
  and hashedNumber: BytesT
  and bet: IntT
  and guess: CoinT
```

In general, Isabelle/Solidity supports most of the basic Solidity data types, such as bit-sized integers, bytes, and addresses. Enums can be encoded as integers with corresponding abbreviations.

Methods

We do not show the specification for all of the methods of the casino contract here but we only discuss one. The others can be similarly translated from the original Solidity contract.

The specification of a new method can be done using the keyword `emethod`. To allow a method to receive funds we can set the payable flag by using the corresponding `payable` keyword. What follows is a specification of stack (`param`), memory (`memory`), and calldata (`calldata`) parameters and corresponding types. Finally, one can provide the body of the function using the `where` keyword followed by a corresponding monad specification (using `"do {...}"`, notation).

Listing 2 shows the specification of the function `decideBet`. It is declared to be payable and accepts one parameter `secretNumber`, of integer type. Lines 5-7, implement preconditions for deciding a bet, i.e., only the operator can call the function, the game should be in state `BET_PLACED` and `secretNumber` should be equal to the `hashedNumber`. For this purpose, Isabelle/Solidity employs `assert` which models the Solidity `require` command.

Isabelle/Solidity also supports other Solidity statements, such as control structures and assignment operators. For example, in Line 9, `IF`...`THEN`...`ELSE` reveals `HEADS` or `TAILS` by taking the modulus ($\langle\%\rangle$) of `secretNumber`. For assignment operators, Isabelle/Solidity distinguishes between stack (`::=`) and storage (`::=`$_s$) assignments along with corresponding lookup operators ($\sim$ and $\sim_s$).

Listing 2: Isabelle/Solidity method for Casino

```
emethod decideBet payable
  param secretNumber: IntT
where
  do {
    byOperator;
    inState (Sint BET_PLACED);
    ⟨assert⟩ (hashedNumber ∼s [] ⟨=⟩ (⟨keccak256⟩ (secretNumber ∼ [])));
    decl TSint secret;
    secret [] ::= IF ((secretNumber ∼ []) ⟨%⟩ ⟨sint⟩ 2) ⟨=⟩ (⟨sint⟩ 0)
                  THEN ⟨sint⟩ HEADS ELSE ⟨sint⟩ TAILS;
    IF (secret ∼ []) ⟨=⟩ (guess ∼s []) THEN
      do {
        pot [] ::=s ((pot ∼s []) ⟨−⟩ (bet ∼s []));
        bet [] ::=s ⟨sint⟩ 0;
        ⟨transfer⟩ (player ∼s []) ((bet ∼s []) ⟨*⟩ (⟨sint⟩ 2))
      }
    ELSE
      do {
        pot [] ::=s pot ∼s [] ⟨+⟩ bet ∼s [];
        bet [] ::=s ⟨sint⟩ 0
      };
    state [] ::=s ⟨sint⟩ IDLE
  },
```

5 Verification

Isabelle/Solidity facilitates the specification of invariants using the `invariant` command. This command requires a user to provide the name of the invariant, followed by its specification in terms of a predicate formulated over the contract's store and balance. It then generates a definition for the invariant which, in addition to the predicate specified by the user, also requires the value of member variables to adhere to their types. The command also proves corresponding introduction and elimination rules which can be invoked during verification.

Assume that we want to ensure our contract has always enough funds to cover the payout of players. More formally, we want to ensure that, whenever the game is in the BET_PLACED state, the contract's internal balance satisfies:

$$pot\_balance(s, b) = b \geq s(\text{``}pot\text{''}) + s(\text{``}bet\text{''}) \ \wedge\ s(\text{``}bet\text{''}) \leq s(\text{``}pot\text{''}) \qquad (1)$$

and if it is not in BET_PLACED, then

$$pot\_balance(s, b) = b \geq pot \qquad (2)$$

The corresponding specification in Isabelle/Solidity is given in Listing 3.

Listing 3: Invariant in Isabelle/Solidity

```
invariant pot_balance sb where
  (fst sb state = Value (Sint BET_PLACED)
    ⟶ snd sb ≥ unat (sint (vt (fst sb pot)))
               + unat (sint (vt (fst sb bet)))
       ∧ sint (vt (fst sb bet)) ≤ sint (vt (fst sb pot))) ∧
  (fst sb state ≠ Value (Sint BET_PLACED)
    ⟶ snd sb ≥ unat (sint (vt (fst sb pot))))
  for "casino"
```

To formally verify the invariant, Isabelle/Solidity provides the verification command. It requires the user to provide a name followed by an invariant specified using the invariant keyword. Moreover, a user can provide postconditions for the constructor and each of the contract's methods. The corresponding specification for our casino contract is shown in Listing 4.

Listing 4: Verification in Isabelle/Solidity

```
verification pot_balance:
  pot_balance
 "K (K (K True))"
 "createGame" "createGame_post" and
 "placeBet" "placeBet_post" and
 "decideBet" "decideBet_post" and
 "addToPot" "K (K (K True))" and
 "removeFromPot" "K (K (K (K True)))"
 for "casino"
```

The postconditions require a method to update the corresponding state of the game properly. For example, placeBet_post imposes state-transition safety for the placeBet method transaction, i.e., successful method call changes the end-state of the contract to BET_PLACED. The postcondition is expressed using abbreviation in Isabelle/Solidity as Listing 5.

Listing 5: Post-condition in Isabelle/Solidity

```
abbreviation(in Contract) placeBet_post where
"placeBet_post hn start_state return_value end_state ≡
 state.Storage end_state this state = BET_PLACED"
```

The verification command tries to prove a general correctness theorem for our contract. To this end, it provides the user with a set of proof obligations which are required to be discharged to verify the contract. For our example, the verification command provides us with six different proof obligations (one for each of the contract's functions). The one for decideBet is shown in Listing 6.

Listing 6: Verifying casino contract in Isabelle/Solidity

```
 show ⋀call secretNumber. (⋀x h r. effect (call x) h r ⟹vcond x) h r)
 ⟹ effect (decideBet call secretNumber) s r
 ⟹ inv_state pot_balance s
 ⟹ post s r pot_balance (K True) (decideBet_post secretNumber)
unfolding decideBet_def
apply (erule post_exc_true, erule_tac post_wp)
unfolding inv_state_def
apply (vcg | solve<auto simp add: wpsimps>)+
...
```

The goal basically requires to show that the invariant is preserved by the function. To discharge it, a user can use our verification condition generator and general Isabelle reasoning infrastructure.

6 Related Work

Early approaches to the verification of smart contracts are mostly based on automatic verification techniques. A popular example of research in this area is solc-verify [10] which is based on the boogie verifier [7]. While tools in this category are fully automated, they are often limited in expressiveness compared to interactive verification approaches.

One line of research in this area focusses on the development of tools to verify smart contracts independent of the programming language. One example in this area is the work of Cassez et al. [3, 4] about deductive verification of smart contracts with Dafny. While these tools can also be used to verify Solidity smart contracts, they may reach their limits when it comes to contracts involving more specialized language features.

Other tools focus on the verification of contracts written in a particular language, such as Solidity. One example here is SolidiKeY [2] which allows to formally specify and verify Solidity smart contracts using the KeY prover. SolidiKeY is based on an axiomatic semantics of Solidity which is the main difference to Isabelle/Solidity, which is based on a denotational semantics [13] and implemented in a definitional approach.

Isabelle/Solidity tool, proposed in this paper, is based on the shallow embedding of subset of Solidity in Isabelle/HOL [12]. The approach allows for better automation and therefore has been employed in this paper to develop concrete syntax and support for the formal specification and verification of Solidity smart contracts.

7 Conclusion

In this paper, we present Isabelle/Solidity, a tool for the formal specification and verification of Solidity smart contracts. The tool is implemented as a definitional extension of the Isabelle proof assistant and provides features for the specification of contracts and invariants, and the verification of invariants and postconditions. To support the user in the verification it also provides a verification condition generator based on a weakest precondition calculus.

Isabelle/Solidity currently supports many features of Solidity, including domain specific expressions and advanced data types such as maps and arrays and their representation in different types of stores (storage, memory, calldata). There are, however, more advanced features of the language, such as inheritance, which is currently not supported and a task for future work.

Another limitation is the lack of a verified compiler for Isabelle/Solidity. Thus, it cannot be guaranteed that the verified properties hold on the level of EVM bytecode. Thus, another direction for future work is the development of a verified compiler for Isabelle/Solidity.

References

1 Wolfgang Ahrendt. Welcome to Fabulous Las Contract Blockchain, 12 2024. URL: `https://web.archive.org/web/20241209135636/https://verifythis.github.io/02casino/`.

2 Wolfgang Ahrendt and Richard Bubel. Functional Verification of Smart Contracts via Strong Data Integrity. In *Leveraging Applications of Formal Methods, Verification and Validation: Applications: 9th International Symposium on Leveraging Applications of Formal Methods, ISoLA 2020, Rhodes, Greece, October 20–30, 2020, Proceedings, Part III 9*, pages 9–24. Springer, 2020.

3 Franck Cassez, Joanne Fuller, and Horacio Mijail Anton Quiles. Deductive Verification of Smart Contracts with Dafny. In Jan Friso Groote and Marieke Huisman, editors, *Formal Methods for Industrial Critical Systems - 27th International Conference, FMICS 2022, Warsaw, Poland, September 14-15, 2022, Proceedings*, volume 13487 of *Lecture Notes in Computer Science*, pages 50–66. Springer, 2022. `doi:10.1007/978-3-031-15008-1\_5`.

4 Franck Cassez, Joanne Fuller, and Horacio Mijail Anton Quiles. Deductive Verification of Smart Contracts with Dafny. *Int. J. Softw. Tools Technol. Transf.*, 26(2):131–145, 2024. URL: `https://doi.org/10.1007/s10009-024-00738-1`, `doi:10.1007/S10009-024-00738-1`.

5 CipherTrace. Cryptocurrency Crime and Anti-money Laundering Report. Technical report, 2021.

6 David Cock, Gerwin Klein, and Thomas Sewell. Secure Microkernels, State Monads and Scalable Refinement. In Otmane Ait Mohamed, César Muñoz, and Sofiène Tahar, editors, *Theorem Proving in Higher Order Logics*, pages 167–182. Springer, 2008.

7 Robert DeLine and K Rustan M Leino. Boogiepl: A Typed Procedural Language for Checking Object-oriented Programs. Technical report, Citeseer, 2005.

8 Edsger W. Dijkstra. Guarded Commands, Nondeterminacy and Formal Derivation of Programs. *Commun. ACM*, 18(8):453–457, aug 1975.

9 Etherscan. Ethereum Daily Deployed Contracts Chart, 02 2025. URL: `https://etherscan.io/chart/deployed-contracts`.

10 Ákos Hajdu and Dejan Jovanović. solc-verify: A Modular Verifier for Solidity Smart Contracts. In *Verified Software. Theories, Tools, and Experiments: 11th International Conference, VSTTE 2019, New York City, NY, USA, July 13–14, 2019, Revised Selected Papers 11*, pages 161–179. Springer, 2020.

11 Alexander Krauss. Recursive Definitions of Monadic Functions. *Electronic Proceedings in Theoretical Computer Science*, 43:1–13, 2010. URL: `http://dx.doi.org/10.4204/EPTCS.43.1`, `doi:10.4204/eptcs.43.1`.

12 Diego Marmsoler, Asad Ahmed, and Achim D Brucker. Secure Smart Contracts with Isabelle/Solidity. In *International Conference on Software Engineering and Formal Methods*, pages 162–181. Springer, 2024.

13 Diego Marmsoler and Achim D Brucker. Isabelle/solidity: A Deep Embedding of Solidity in Isabelle/HOL. *Formal Aspects of Computing*, 2022.

14 T. Nipkow, L. C. Paulson, and M. Wenzel. Isabelle/HOL: A Proof Assistant for Higher-Order Logic, 2002.

15 Philip Wadler. Monads for Functional Programming. In Manfred Broy, editor, *Program Design Calculi*, pages 233–264. Springer, 1993.

16 Ycharts. Ethereum Transactions Per Day, 02 2025. URL: `https://ycharts.com/indicators/ethereum_transactions_per_day`.

A Casino: Solidity Smart Contract

Listing 7: Solidity source code for the Casino

```
contract Casino {
  enum Coin { HEADS, TAILS } ;
  enum State { IDLE, GAME_AVAILABLE, BET_PLACED }
  State private state;
  address public operator, player;
  uint public pot;
  bytes32 public hashedNumber;
  uint public bet;
  Coin guess;

  function createGame(bytes32 hashNum)
    public byOperator, inState(IDLE) {
    hashedNumber = hashNum;
    state = GAME_AVAILABLE;
  }

  function placeBet(Coin _guess) public payable inState(GAME_AVAILABLE) {
    require (msg.sender != operator);
    require (msg.value <= pot);
    state = BET_PLACED;
    player = msg.sender;
    bet = msg.value;
    guess = _guess;
  }

  function decideBet(uint secretNumber)
  public byOperator, inState(BET_PLACED) {
    require (hashedNumber == keccak256(secretNumber));
    Coin secret = (secretNumber % 2 == 0)? HEADS : TAILS;
    if (secret == guess) {
      pot = pot - bet;
      bet = 0;
      player.transfer(bet*2);
    } else {
      pot = pot + bet;
      bet = 0;
    }
    state = IDLE;
  }

  function addToPot() public payable byOperator {
    pot = pot + msg.value;
  }

  function removeFromPot(uint amount) public byOperator, noActiveBet {
    operator.transfer(amount);
    pot = pot - amount;
  }
}
```