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 deductive 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]

Acknowledgements I want to thank . . .

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 [15].

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 this end, we developed Isabelle/Solidity, a tool for the verification of Solidity smart contracts. The main aim of this paper is to introduce the tool and demonstrate it in terms of an example. To this end its contribution is two-fold:

- We provide an overview of the technical architecture of Isabelle/Solidity.
- We use it to verify the casino contract from the VerifyThis verification challenge.

42nd Conference on Very Important Topics (CVIT 2016).
Editors: John Q. Open and Joan R. Access; Article No. 23; pp. 23:1–23:8
OpenAccess Series in Informatics
OASICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

2 Overview

Figure 1 depicts an overview of our approach to verify Solidity smart contract. Isabelle/Solidity is based on four Isabelle theories which are represented by the blue rectangles:

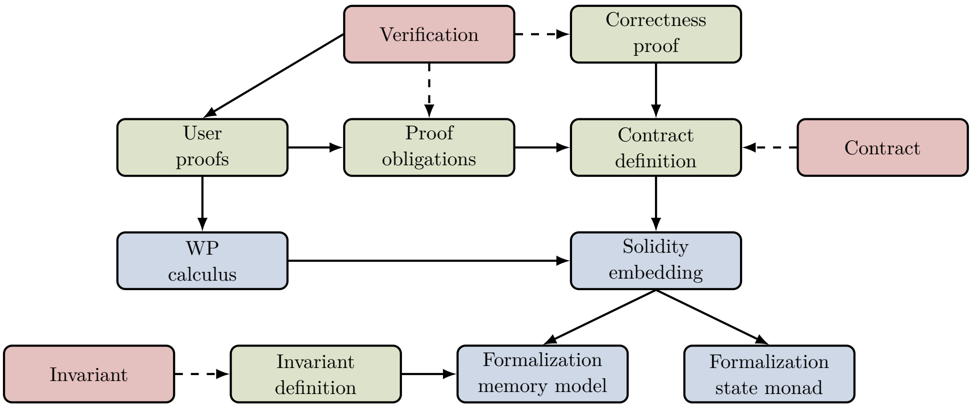

Figure 1 Isabelle/Solidity: theories (blue), commands (red), and generated artefacts (green).

Formalization state monad We model Solidity programs as functions which manipulate states. Such functions are usually described in terms of state monads [14] and this theory contains our formalization of it based on [7].

Formalization memory model Solidity has a quite particular memory model with different types of stores which support different types of data structures. 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 our Solidity statements.

WP calculus To support a user in the verification of properties for our Solidity programms we provide a weakest precondition calculus [8] for our Solidity statements. This theory formalizes the calculus.

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 a user 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 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 [10].

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 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 secret number 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 in not in state `BET_PLACED` by calling `removeFromPot`.

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 member variables and corresponding types, followed by a specification of the constructor and the contract's methods.

Storage Variables

Listing 1 shows the specification of storage variables for the casino smart 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
```

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`.

Listing 2: Isabelle/Solidity method for Casino

```
1  emethod decideBet payable
2    param secretNumber: IntT
3  where
4    do {
5      byOperator;
6      inState (Sint BET_PLACED);
7      ⟨assert⟩ (hashedNumber ∼s [] ⟨=⟩ (⟨keccak256⟩ (secretNumber ∼ [])));
8      decl TSint secret;
9      secret [] ::= IF ((secretNumber ∼ []) ⟨%⟩ ⟨sint⟩ 2) ⟨=⟩ (⟨sint⟩ 0)
10                     THEN ⟨sint⟩ HEADS ELSE ⟨sint⟩ TAILS;
11     IF (secret ∼ []) ⟨=⟩(guess ∼s []) THEN
12     do {
13       decl TSint bet_old;
14       bet_old [] ::= bet ∼s [];
15       bet [] ::=s ⟨sint⟩ 0;
16       pot [] ::=s ((pot ∼s []) ⟨-⟩ (bet_old ∼ []));
17       ⟨transfer⟩ (player ∼s []) ((bet ∼s []) ⟨*⟩ (⟨sint⟩ 2))
18     }
19     ELSE
20     do {
21       pot [] ::=s pot ∼s [] ⟨+⟩ bet ∼s [];
22       bet [] ::=s ⟨sint⟩ 0
23     };
24     state [] ::=s ⟨sint⟩ IDLE
25   },
```

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 (⟨%⟩) of `secretNumber`. For assignment operators, Isabelle/Solidity distinguishes between stack (`::=`) and storage (`::=`$_s$) assignments along with corresponding lookup operators (∼ and ∼$_s$).

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 the invariant as predicates 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 for automated verification of the invariants.

Example 5.1 (Invariant). 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:

$$pot\_balance(s, b) = b \geq s(\text{“}pot\text{”}) + s(\text{“}bet\text{”}) \ \wedge \ s(\text{“}bet\text{”}) \leq s(\text{“}pot\text{”}) \tag{1}$$

and if it is not in `BET_PLACED`, then

$$pot\_balance(s, b) = b \geq pot \tag{2}$$

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 = sdata.Value (Sint BET_PLACED)
    ⟶ snd sb ≥ unat (valtype.sint (sdata.vt pot))
                + unat (valtype.sint (sdata.vt bet))
        ∧ valtype.sint (sdata.vt bet) ≤ valtype.sint (sdata.vt pot)) ∧
  (fst sb state ≠ sdata.Value (Sint BET_PLACED)
    ⟶ snd sb ≥ unat (valtype.sint (sdata.vt 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. The postconditions require a method to update the corresponding state of the game properly.

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"
```

In order to automate the verification task, Isabelle/Solidity provides proof automation support in terms of a weakest precondition calculus. Verification of the Casino smart contract using Isabelle/Solidity resulted in the exploration of a major vulnerability, i.e., Reenetrancy. While verifying the invariant in Example 5.1, it was not possible to verify unless changing the order of the Line 32 and 33 (see Appendix A) which enabled the completion of the verification task. It is due to possibility of calling the function `decideBet` without setting `bet=0` which may lead to unintended behaviour.

6 Related Work

Considering the mission-critical nature of the smart contracts in blockchain technology, formal methods techniques, e.g., Model Checking [3], theorem proving [4], have been employed to formally specify and verify Solidity smart contracts. Model checking [6] is a push-button automatic formal methods technique, however, has limited expressiveness due to finite-state machine modelling and is prone to state-space explosion. On the other hand, theorem proving is highly-expressive and has also been utilized for the verification of Solidity smart contracts.

In theorem proving literature, deep and shallow embedding are two common approaches for the verification of Solidity language. A deep embedding of operational semantic of Solidity in SolidiKeY [2] allows to formally specify and verify smart contracts using KeY theorem prover. Similarly, Jakab [16] formalizes a Coq interpreter, utilizing deep embedding of operational semantic of Solidity, for the smart contract verification. However, aforementioned works rely on the axiomatic verification approach therefore may lead to miss corner cases in verifying smart contracts. In this direction, Diego et al. [12] developed a deep embedding framework for denotational semantic of Solidity in Isabelle/HOL which support correct-by-construction approach for the formal verification of smart contracts. On the other hand, there are two notable efforts for shallow embedding of Solidity in Isabelle theorem prover. Ribeiro et al. [13] utilize mix of deep and shallow embedding to develop an intermediary low-level specification language, `SOLI`, for the specification and verification of Solidity smart contracts in Isabelle/HOL. Whereas [11] provides a shallow embedding of operational semantic of Solidity in Isabelle/HOL. The framework is equipped with complex data types, such as mappings and arrays, and verification of invariants as compared to SOLI.

Isabelle/Solidity tool, proposed in this paper, is based on the shallow embedding approach [11]. This is mainly due to the scope of the proposed tool. The objective of the proposed tool is to formally specify and verify the correctness of the Solidity smart contracts which is best served, in terms of effort and time, by shallow embedding as compare to deep embedding. This has been empirically shown in [11].

7 Conclusion

In this paper, we present Isabelle/Solidity tool for the formal specification and verification of Solidity smart contracts. The tool facilitates Solidity data-types, functions, modifiers, statements, expressions and post/pre-condition specifiers in Isabelle/HOL. The formal specification relies upon the underlying shallow embedding of Solidity expressions and statements as state-monads and storage models for different types of stores in Solidity. For the verification purpose, tool supports invariant specification for the contract that in turn is supported by verification condition generator to automate the verification process. Finally, we evaluated the approach by means of the Casino case study. The use of the Isabelle/Solidity resulted in the exploration of re-entrancy vulnerability in the original version of Casino smart contract. However, there are few challenges in order to fully cover the advanced features of Solidity.

In this regard, inheritance is one of the most notable Solidity feature which can be introduced in the proposed tool for verifying interesting properties of the smart contracts. Moreover, from the verification point-of-view, automation can be further improved by providing specialized rules w.r.t the context of the smart contracts.

References

1 Wolfgang Ahrendt. Ethereum transactions per day, 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 Pedro Antonino and AW Roscoe. Formalising and verifying smart contracts with solidifier: a bounded model checker for solidity. *arXiv preprint arXiv:2002.02710*, 2020.

4 Karthikeyan Bhargavan, Antoine Delignat-Lavaud, Cédric Fournet, Anitha Gollamudi, Georges Gonthier, Nadim Kobeissi, Natalia Kulatova, Aseem Rastogi, Thomas Sibut-Pinote, Nikhil Swamy, et al. Formal verification of smart contracts: Short paper. In *Proceedings of the 2016 ACM workshop on programming languages and analysis for security*, pages 91–96, 2016.

5 CipherTrace. Cryptocurrency crime and anti-money laundering report. Technical report, 2021.

6 Edmund M Clarke. Model checking. In *Foundations of Software Technology and Theoretical Computer Science: 17th Conference Kharagpur, India, December 18–20, 1997 Proceedings 17*, pages 54–56. Springer, 1997.

7 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.

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 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`.

11 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.

12 Diego Marmsoler and Achim D Brucker. Isabelle/solidity: a deep embedding of solidity in isabelle/hol. *Formal Aspects of Computing*, 2022.

13 Maria Ribeiro, Pedro Adão, and Paulo Mateus. Formal verification of ethereum smart contracts using isabelle/hol. In *Logic, Language, and Security: Essays Dedicated to Andre Scedrov on the Occasion of His 65th Birthday*, pages 71–97. Springer, 2020.

14 Philip Wadler. Monads for functional programming. In Manfred Broy, editor, *Program Design Calculi*, pages 233–264. Springer, 1993.

15 Ycharts. Ethereum transactions per day, 02 2025. URL: `https://ycharts.com/indicators/ethereum_transactions_per_day`.

16 Jakub Zakrzewski. Towards verification of ethereum smart contracts: a formalization of core of solidity. In *Verified Software. Theories, Tools, and Experiments: 10th International Conference, VSTTE 2018, Oxford, UK, July 18–19, 2018, Revised Selected Papers 10*, pages 229–247. Springer, 2018.

A Casino: Solidity Smart Contract

Listing 5: 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;
      player.transfer(bet*2);
      bet = 0;}
    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;}
  }
```