Isabelle/Solidity: A Tool for the Verification of Solidity Smart Contracts

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

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

Abstract

2012 ACM Subject Classification Replace ccsdesc macro with valid one

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

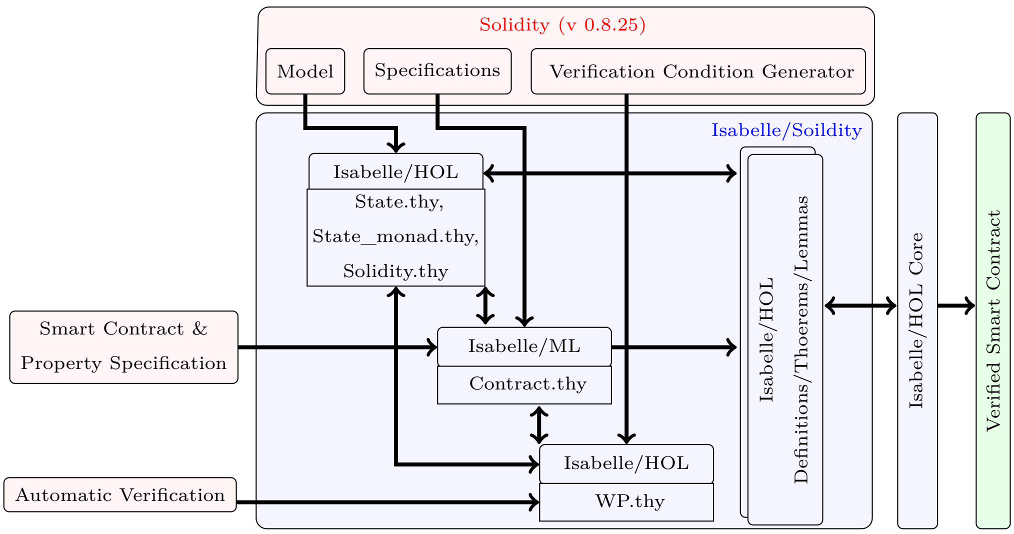
Digital Object Identifier 10.4230/OASIcs.CVIT.2016.23

Funding *Asad Ahmed*: (Optional) author-specific funding acknowledgements
Diego Marmsoler: [funding]

Acknowledgements I want to thank ...

1 Introduction

2 Overview



42nd Conference on Very Important Topics (CVIT 2016).
Editors: John Q. Open and Joan R. Access; Article No. 23; pp. 23:1–23:7
OpenAccess Series in Informatics
OASICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

3 Case Study

We use a case study, Casino, from VerifyThis long-term challenge. Casino is a betting game based on guessing the outcome of coin-tossing and is implemented using Solidity syntax (see Appendix A). Anytime, the game is characterized by three states, i.e., `IDLE`, `GAME_AVAILABLE` and `BET_PLACED`. Initially, game is in `IDLE` state. The game has been implemented using following functions in Solidity:

- Only operator can create a game by calling `creatGame` function which changes the state to `GAME_AVAILABLE`, and also requires a secret number from the player to ensure unbiased and verifiable bet.
- Once in `GAME_AVAILABLE` state, a player can place a bet by invoking `placeBet` fucntion. This function saves the bet (`HEAD` or `TAIL`), amount of the bet and changes the state to `BET_PLACED`.
- Now, operator can decide the bet anytime by calling `decideBet` function by providing secret number generated when game was created. It allows the function to verify the bet and also decide the outcome of the coin tossing (`HEAD` or `TAIL`). If player wins then double the amount of the bet is transferred to the player else amount equal to the bet is transferred to the operator. It also changes the state of the game to `IDLE` state.
- Operator may add money to the bet, at anytime, using `addToPot` but can only remove if game is in not in `BET_PLACED` state by calling `decideBet`.

The Casino smart contract is selected for two accounts: One, it has been implemented using sophisticated and advanced features of Solidity syntax including data types, global and local variables, functions, modifiers and precondition specifiers. Thus, requires to develop powerful equivalent syntax support in the proposed tool to embed the program logic. Two, from verification aspect, Casino has been employed as a case study in literature and hence allows to compare and report the result to gauge the performance of the proposed tool, comparatively.

4 Specification

In this section, we present a Solidity equivalent specification of smart contracts in Isabelle/Solidity. We primarily focused on specifications of state or local variables, data types, functions, modifiers, precondition specifiers and statements in Isabelle/Solidity.

Storage Variables

In Listing 1, Casino smart contract is defined using Isabelle/Solidity command `contract` followed by *name* and list of storage variables. The tool, also, allows data-type annotations to specify the types of variables (Listing 1).

Listing 1: Isabelle/Solidity data types for Casino

```
contract Casino
  for state: "SType.TValue TSint"
  and operator: "SType.TValue TAddress"
  and player: "SType.TValue TAddress"
  and pot: "SType.TValue TSint"
  and hashedNumber: "SType.TValue TBytes"
  and bet: "SType.TValue TSint"
  and guess: "SType.TValue TSint"

constructor payable
where
  "skip"
```

Methods

A function, `decideBet`, in Listing 2 showcases Isabelle/Solidity features to specify Solidity functions. The keyword `emethod` defines `decideBet` which has a `payable` modifier. A memory (local) variable, `secretNumber`, of integer type is declared using keyword `param`. Isabelle/Solidity allows to specify the body of the function using `where "do {...}"`, structure.

Listing 2: Isabelle/Solidity method for Casino

```
1  emethod decideBet payable
2    param secretNumber: "SType.TValue TSint"
3  where
4    "do {
5      byOperator;
6      inState (valtype.Sint 2);
7      ⟨assert⟩ ((hashedNumber ∼s []) ⟨=⟩ (⟨keccak256⟩ (secretNumber ∼ [])));
8      decl TSint secret;
9      secret [] ::= IF ((secretNumber ∼ []) ⟨%⟩ ⟨sint⟩ 2) ⟨=⟩( ⟨sint⟩ 0)
10               THEN ⟨sint⟩ 0 ELSE ⟨sint⟩ 1;
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⟩ 0
25   }",
```

Lines 5-7, implement preconditions for Casino, i.e., only operator can call the function, game should be in `IDLE` state and `secretNumber` should be equal to the `hashedNumber`. For this purpose, Isabelle/Solidity employs `assert` which models the Solidity `require` command. That is, if preconditions are not met then `assert` throws an exception.

Isabelle/Solidity also supports Solidity statements such as control structures and assignment operators. For example, in Line 9, `IF`...`THEN`...`ELSE` reveals `HEAD` or `TAIL` by taking the modulus ($\langle\%\rangle$)of `secretNumber`. For assignment operators, Isabelle/Solidity employs storage ($::=$) and stack ($::=_s$) assignment operators along with search operators ($\sim$ and $\sim_s$) for respective stores.

5 Verification

Isabelle/Solidity facilitates invariance specification using `invariant` command over the contract balance and storage. 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. The command then generates introduction and elimination rules which can be invoked for automated verification of the invariants.

Example 5.1 (Invariant). Assume that we want to verify that when game is in `BET_PLACED` state, contract's internal balance satisfies:

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

and if not in `BET_PLACED`, then

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

Above invariant ensures payout safety for players by explicitly placing upper bound on the contract balance w.r.t the amount of bet and pot in `BET_PLACED` state and otherwise. The corresponding specification in Isabelle/Solidity is given in Listing 3:

Listing 3: Invariant in Isabelle/Solidity

```
invariant pot_balance sb where
   "(fst sb STR ''state'' = sdata.Value (Sint 2)
     ⟶ snd sb ≥ unat (valtype.sint (sdata.vt (fst sb STR ''pot'')))
   + unat (valtype.sint (sdata.vt (fst sb STR ''bet'')))
       ∧ valtype.sint (sdata.vt (fst sb STR ''bet'')) ≤
   valtype.sint (sdata.vt (fst sb STR ''pot''))) ∧
   (fst sb STR ''state'' ≠ sdata.Value (Sint 2)
   ⟶ snd sb ≥ unat (valtype.sint (sdata.vt (fst sb STR ''pot''))))"
 for "casino"
```

To formally verify the invariant, Isabelle/Solidity allows to specify the verification of invariants using `verificaiton` command. It accepts the name of the invariant, invariant and postconditions on the constructor and methods to generate proof obligations. For example, in Listing 4, `creatGame_post` is a postcondition that ensures that state of the game will be `creatGame` after the execution of `creatGame` fucntion. In order to automate the verification task, the tool has weakest precondition calculus based verification condition generator to discharge the proof obligations.

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"
```

Discussions

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 [2], theorem proving [3], have been employed to formally specify and verify Solidity smart contracts. Model checking [4]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 [1] allows to formally specify and verify smart contracts using KeY theorem prover. Similarly,Jakab [8] 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. [6] 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. [7] 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 [5] 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 [5]. 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 [5].

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 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.

2 Pedro Antonino and AW Roscoe. Formalising and verifying smart contracts with solidifier: a bounded model checker for solidity. *arXiv preprint arXiv:2002.02710*, 2020.

3 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.

4 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.

5 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.

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

7 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.

8 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;}
  }
```