Isabelle/Solidity for Smart Contracts

Jane Open Access
Dummy University Computing Laboratory, [optional: Address], Country
My second affiliation, Country

Joan R. Public[1]
Department of Informatics, Dummy College, [optional: Address], Country

Abstract

2012 ACM Subject Classification Replace ccsdesc macro with valid one

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

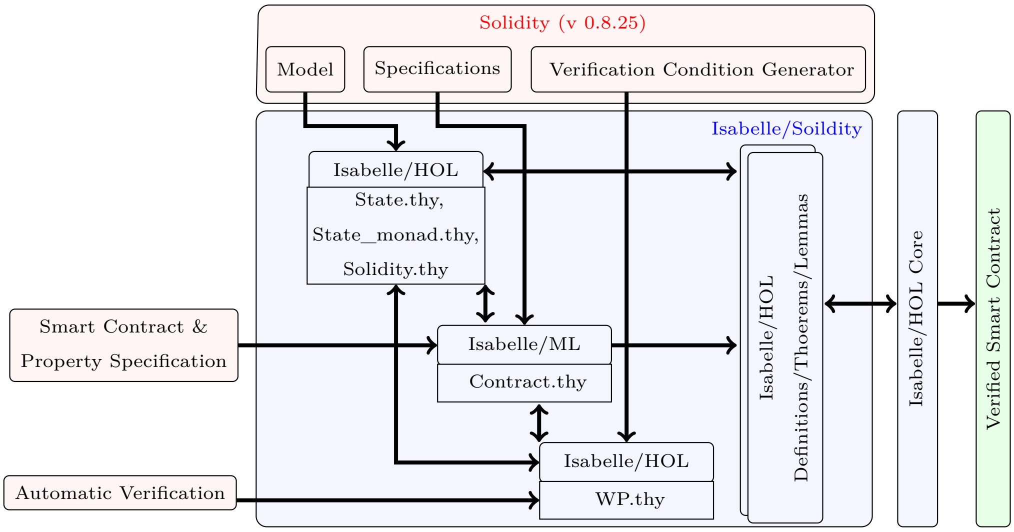
Digital Object Identifier 10.4230/OASIcs.CVIT.2016.23

Funding *Jane Open Access*: (Optional) author-specific funding acknowledgements
Joan R. Public: [funding]

Acknowledgements I want to thank ...

1 Introduction

2 Overview



[1] Optional footnote, e.g. to mark corresponding author

42nd Conference on Very Important Topics (CVIT 2016).
Editors: John Q. Open and Joan R. Access; Article No. 23; pp. 23:1–23:5
OpenAccess Series in Informatics
OASICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

3 Case Study

Listing 1: Solidity source code for the Casino

```
1   contract Casino {
2     enum Coin { HEADS, TAILS } ;
3     enum State { IDLE, GAME_AVAILABLE, BET_PLACED }
4     State private state;
5     address public operator, player;
6     uint public pot;
7     bytes32 public hashedNumber;
8     uint public bet;
9     Coin guess;
10
11    function createGame(bytes32 hashNum)
12      public byOperator, inState(IDLE) {
13      hashedNumber = hashNum;
14      state = GAME_AVAILABLE;
15    }
16
17    function placeBet(Coin _guess) public payable inState(GAME_AVAILABLE) {
18      require (msg.sender != operator);
19      require (msg.value <= pot);
20      state = BET_PLACED;
21      player = msg.sender;
22      bet = msg.value;
23      guess = _guess;
24    }
25
26    function decideBet(uint secretNumber)
27    public byOperator, inState(BET_PLACED) {
28      require (hashedNumber == keccak256(secretNumber));
29      Coin secret = (secretNumber \% 2 == 0)? HEADS : TAILS;
30      if (secret == guess) {
31        pot = pot - bet;
32        player.transfer(bet*2); bet = 0;}
33      else {
34        pot = pot + bet; bet = 0;
35        }
36      state = IDLE;}
37    function addToPot() public payable byOperator { pot = pot + msg.value;}
38
39    function removeFromPot(uint amount) public byOperator, noActiveBet {
40      operator.transfer(amount);
41      pot = pot - amount;}
42    }
```

Casino (Listing 1) implements a bet game based on the flip-coin (Line 2) using Solidity syntex. This game has three explicit state: `IDLE`, `GAME_AVAILABLE`, `BET_PLACED` (Line 3). An operator may create a new game by calling `creatGame` function (Line 11-15). The operator provides a `hashNum` (Line 11) to ensure unbiased and verifiable bet (Line 26-30). The function, `creatGame`, uses two modifiers (`byOperator` and `inState(s)`) to implement only operator access and state-flow control, i.e., new game can only be created in `IDLE` state. The `creatGame` function changes the state to `BET_PLACED` which allows players to place bet on `HEADS` or `TAILS`, as `_guess`, by calling `betPlaced` function. The `betPlaced` function uses require to safe guard possible manipulation from operator by restricting its access (Line 18) and payout safety by capping the maximum bet amount with pot balance (Line 19) in the game.

Once the game is in BET_PLACED, the operator may decide the bet (decideBet) by passing secretNumber (Line 26). The secret number is used to verify the bet against hashNumber (Line 28) to ensure fairness. Then, secret number is used to reveal HEADS (even) or TAILS (odd) (Line 29) which is further utilized to resolve the bet (Line 30-33). In case, player wins, then double amount of the original bet is transferred to the player's account (Line 30), otherwise, amount equivalent to original bet is added to the pot (Line 32).

Finally, in IDLE and GAME_AVAILABLE states, an operator may add or remove any amount of money from pot by invoking addToPot or removeFromPot function. However, in BET_PLACED state, an operator is allowed to only remove the money which is ensured by the modifier noActiveBet.

4 Specification

In this section, we present a Solidity equivalent specification of smart contracts in Isabelle/Solidity. We primarly focused on specifications of state or local variables, data types, functions, modifiers, precondition specifiers and statments 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 "STR state": TSint
        and "STR operator": TAddress
        and "STR player": TAddress
        and "STR pot": TSint
        and "STR hashedNumber": TBytes
        and "STR bet": TSint
        and "STR guess": TSint
```

Methods

A function, removeFromPot, in Listing 2 showcases Isabelle/Solidity features to specify Solidity functions. The keyword emethod defines removeFromPot which has a payable modifier. A memory (local) variable, amount, of integer type for the function 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 removeFromPot payable
2    param "STR amount": TSint
3  where
4    "do {
5      byOperator;
6      noActiveBet;
7      assign_storage_monad (STR ''pot'') []
8          (minus_monad_safe (storeLookup (STR ''pot'') [])
9                  (stackLookup (STR ''amount'') []));
10     transfer_monad (storeLookup (STR ''operator'') [])
11         (stackLookup (STR ''amount'') [])
12   }"
```

Preconditions on access control, `byOperator`, and state-flow, `noActiveBet`, are specified as constants which are based on a higher-order logic funciton, i.e., `assert`, given in Listing 3. The function `assert` returns normal exection if `msg.sender` is the operator else it throws an exception. In this way, Isabelle/Solidity ensures the preconditions which ensures intended behavior of the method.

Listing 3: Preocndition in Isabelle/Solidity

```
abbreviation byOperator::"(unit, ex, 'a state) state_monad" where
 "byOperator ≡ assert Err (λs. sdata.Value (valtype.Address msg_sender)
                         = state.Storage s this (STR ''operator''))"
```

Isabelle/Solidity also supports Solidity statements such as control structures and assignment operators. From Line 7-9, an `amount` is subtracted from the `pot` using `assign_storage_monad` which besides identifiers also need store location depending on the variable type.

Next, in Line 10 (of Listing 2), `transfer_monad` transfers (special function) an amount equal to the bet is transferred to the operator account.

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 contracts. 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 internal balance satisfies:

$$(s(\text{“}balances\text{”})) \geq pot + bet \wedge bet \leq pot \tag{1}$$

and if not in `BET_PLACED`, then

$$(s(\text{“}balances\text{”})) \geq pot \tag{2}$$

Above invariant enusres payout safety for players.
The corresponding specification in Isabelle/Solidity is given in Listing 4:

Listing 4: 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"
```

6 Related Work

7 Conclusion

References

1 *16th Annual Symposium on Foundations of Computer Science, Berkeley, California, USA, October 13-15, 1975.* IEEE Computer Society, 1975.

2 Edsger W. Dijkstra. Letters to the editor: go to statement considered harmful. *Commun. ACM*, 11(3):147–148, 1968. doi:10.1145/362929.362947.

3 Jim Gray and Andreas Reuter. *Transaction Processing: Concepts and Techniques.* Morgan Kaufmann, 1993.

4 John E. Hopcroft, Wolfgang J. Paul, and Leslie G. Valiant. On time versus space and related problems. In *16th Annual Symposium on Foundations of Computer Science, Berkeley, California, USA, October 13-15, 1975* [1], pages 57–64. doi:10.1109/SFCS.1975.23.

5 Donald E. Knuth. Computer Programming as an Art. *Commun. ACM*, 17(12):667–673, 1974. doi:10.1145/361604.361612.