Developing Safe Smart Contracts

Sajjad Rezaei
ECE Department
University of Tehran
Tehran, Iran
sajjad.rezaei@ut.ac.ir

Ehsan Khamespanah
ECE Department
University of Tehran
Tehran, Iran
ekhamespanah@ut.ac.ir

Marjan Sirjani
School of IDT
Mälardalen University
Västerås, Sweden
marjan.sirjani@mdh.se

Ali Sedaghatbaf
School of IDT
Mälardalen University
Västerås, Sweden
ali.sedaghatbaf@mdh.se

Siamak Mohammadi
ECE Department
University of Tehran
Tehran, Iran
smohamadi@ut.ac.ir

***Abstract*—Blockchain is a shared, distributed ledger on which transactions are digitally recorded and linked together. Smart Contracts are programs running on Blockchain and are used to perform transactions in a distributed environment without need for any trusted third party. Since smart contracts are used to transfer assets between contractual parties, their safety and security are crucial and badly written and insecure contracts may result in catastrophe. Actor-based programming is known to solve several problems in building distributed software systems. Moreover, formal verification is a solid technique for developing dependable systems. In this paper, we show how the actor model can be used for modeling, analysis and synthesis of smart contracts. We propose Smart Rebeca as an extension of the actor-based language Rebeca, and use the model checking toolset Afra for verification of smart contracts. We implement a synthesizer to synthesise Solidity programs that run on the Ethereum platform from Smart Rebeca models. We examine challenges and opportunities of our approach in modeling, formal verification and synthesis of smart contracts using actors.**

***Index Terms*—Smart Contract, Actor Model, Safety Verification, Model Checking**

I. Introduction

Smart contracts are lines of code that are stored on a Blockchain and automatically executed when predetermined terms and conditions are met. Blockchain is a shared, distributed ledger on which transactions are digitally recorded and linked together. It can be seen as time-stamped series of immutable record of data that is managed by a cluster of computers not owned by any single entity, and provide the entire history of transactions. The benefits of smart contracts are most apparent in business collaborations. These contracts are typically used to enforce some kind of agreement, and are used to perform transactions in a distributed environment without need for any trusted third party.

Since smart contracts are used to transfer valuable assets between contractual parties, their safety and security are of paramount importance. Blockchain assures deterministic execution and consistent state representations, but it cannot guard against badly written or insecure contracts. Therefore, careful design and implementation of contracts is still necessary [1]. Moreover, contracts may be the target of adversaries with malicious intents e.g. to transfer money from legitimate users to themselves. A significant number of deployed smart contracts are intentionally fraudulent [2], and a recent analysis on 20K of them indicated that each of them had at least one security issue [3]. Besides, there is no way to patch bugs and vulnerabilities of smart contracts due to their immutability.

Accounts using smart contracts in a Blockchain are like threads using concurrent objects in shared memory [4]. There are well-known pitfalls in building distributed and concurrent systems using threads [5], and well-established practices for avoiding these problems. For example in Solidity [6] (one of the most mature languages for building smart contracts) execution of smart contracts callback functions and implicit arbitrary execution of fallback functions can cause concurrency problems. A contract may call an external service that calls the contract's callback function in response. If the contract calls more than one service the order of callbacks may be significant while there is no guarantee that the order is preserved. Fallback functions are called because of the Solidity payment mechanism and their order of execution can change the results tremendously.

Most of the currently existing work on verification of smart contracts, e.g. Ahrendt et.al. in [7] and Osterland et.al. in [8], focus on program verification rather than concurrency problems. However, a wide range of issues in safety and security of smart contracts are concurrency problems.

Using actors is a way towards building less error-prone distributed and concurrent models [5]. In this paper, we propose a model-driven approach for developing smart contracts using actors. We propose Smart Rebeca as an extension of the actor-based language Rebeca [9], [10]. Smart contracts are modeled using Smart Rebeca. Smart Rebeca has a syntax similar to Solidity and is supported by the model checking tool Afra [11]. Counter examples are generated and help in debugging the model. Then the correct model can be transformed to Solidity code. Hence, the modeler is enabled to assure the safety and security of Solidity smart contracts before the synthesis. By using the model checking tool of Rebeca, we can find a set of concurrency issues and program bugs. Synthesis of smart contracts based on a formally verified model provides correctness by design which results in improving public acceptance of smart contracts.

II. Preliminaries

A. Solidity

Currently, Bitcoin and the Ethereum virtual machine (EVM) are the most popular platforms that support smart contract development. Among the languages designed to target Ethereum,

Solidity [6] is the most mature and popular one. Solidity is an object-oriented language influenced by C++, Python and JavaScript. This language is statically typed, supports inheritance, libraries and complex user-defined types among other features. However, writing smart contracts in Solidity is still challenging [12], mainly due to the need to use unconventional programming paradigms like fallback functions, function modifiers, etc.

In Listing 1, the code of a simple contract which provides basic banking operations is shown in Solidity.

```
1  contract Bank{
2    mapping(address=>uint) userBalances;
3    function getUserBalance(address user)
         constant returns(uint) {
4      return userBalances[user];
5    }
6    function addToBalance() {
7      userBalances[msg.sender] =
8      userBalances[msg.sender] + msg.value;
9    }
10   function withdrawBalance() {
11     uint amount =
12     userBalances[msg.sender];
13     if (msg.sender.call.value(amount)()
           == false) {
14       throw;
15     }
16     userBalances[msg.sender] = 0;
17   }
18 }
```

Listing 1. Smart Contract's Code of a simple bank in Solidity

A contract is like a class in object-oriented languages, consisting of declaration of state variables, functions, function modifiers, events, structure types, enum types, etc. The values of the state variables (e.g. line 2 of Listing 1) are stored in the blockchain. Provided functions of smart contracts can be called by users or other contracts. In the contract of Listing 1, the user can deposit to the bank with the function `addToBalance` and withdraw using `withdrawBalance` function. The variable `userBalances` is a hash table that saves the money balance of each user with users addresses in the blockchain. Note that functions may have parameters and return values.

Functions have different levels of visibility for other contracts, including `public`, `private`, `external`, and `internal`. There is a set of function modifiers that can be associated with functions to amend the semantics of functions in a declarative way. For example, in line 3 of Listing 1, declaring `getUserBalance` as a `constant` function, disallows assignment to state variables. Keywords `pure`, `view`, `payable`, `anonymous`, `indexed`, `virtual`, and `override` are used for other modifiers which their detailed description is presented in [6].

In Solidity, there are special variables and functions that are used to retrieve information about the blockchain or provide widely-used utility functions. For example, `block.number` returns the current block number and `msg.sender` returns the identifier of the sender of the message (line 7 and 8). Another set of Solidity built-in functions are used for encoding/decoding purpose which are encapsulated in the `abi` package or independent functions like `sha256`, `keccak256`, and `ecrecover`. Solidity provides a set of functions for aborting the execution and reverting changes which are `assert`, `require`, and `revert`. The first two functions revert if their input condition (i.e. a boolean expression) is not met. Also, `selfdestruct` function can be used to destroy the current contract and send its funds to its given Address.

B. Rebeca Modeling Language

Rebeca is an actor-based language designed for modeling and verification of concurrent and distributed systems and is supported by a model checking tool, Afra [9], [10]. There is no shared variables among actors and the communication takes place by asynchronous message passing. Rebeca is designed with the goal to bridge the gap between software engineers and the formal methods community. With usability as one of the primary design goals, Rebeca's concrete syntax is Java-like (close to object-oriented languages), and its computation and communication model is kept simple. Learning Rebeca is fairly easy for programmers and using model checking tools requires far less expertise than deduction-based analysis methods. Rebeca has a formal semantics and allows efficient compositional verification based on model checking. There are multiple state space reduction techniques designed for Rebeca based on its actor-based model of computation [13], [14].

In Rebeca a set of actors are defined and each actor has an unbounded buffer, called message *queue*, for its arriving messages. Computation in Rebeca is event-driven, meaning that each actor takes a message that can be considered as an event from the top of its message queue and executes the corresponding message server. The execution of a message server is atomic which means that there is no way to preempt the execution of a message server of an actor and start executing another message server of that actor.

A Rebeca model consists of a set of reactive classes and the main block. In the main block, actors which are instances of the reactive classes are declared. The body of the reactive class includes the declaration of its known rebecs, state variables, private methods, and message servers. A very simple example of a bank reactive class is depicted in Listing 2 which its behavior is similar to the smart contract of Listing 1. Message servers and private methods consist of the declaration of local variables and the body of the message server. The statements in the body can be assignments (line 11), conditional statements, enumerated loops, non-deterministic assignment, and method or message server calls. Message server calls are sending asynchronous messages to other actors (line 13) or to itself. Private methods cannot be called by other actors. A private method starts with the type of its return value instead of the `msgsrv` keyword (lines 9 to 11 of Listing 2).

A reactive class has an argument of type integer denoting the maximum size of its message queue (line 1 shows that the size is 10 for this reactive class). Although message queues are unbounded in the semantics of Rebeca, to ensure that the

state space is finite, we need a user-specified upper bound for the queue size. The operational semantics of Rebeca has been introduced in [9] in more detail. Actors in Rebeca are single-threaded which is aligned with smart contracts in Solidity .

```
1  reactiveclass Bank(10){
2    knownrebecs {
3      Client client1, client2, client3,
4        client4, client5;
5    }
6    statevars{
7      int[5] balances;
8    }
9    int getUserBalance(int id) {
10     return balances[id];
11   }
12   msgsrv addToBalance(int id, int amount) {
13     balances[id] += amount;
14   }
15   msgsrv withdrawBalance(int id, int r) {
16     ((Client)sender).receive(balances[id]);
17     balances[id] = 0;
18   }
19 }
```

Listing 2. Bank contract in Rebeca

A Rebeca code can be model checked against a given set of Linear Temporal Logic (LTL) properties. These properties specify the correct behaviors/states of the model. For example, in the case of the bank contract, one correctness property is that the balance of all clients must be none-negative in all the states. This property can be specified in LTL using $\Box(balance1 \geq 0 \wedge \cdots balance5 \geq 0)$ formula. Figure 3 shows how the mentioned LTL property is specified in the Rebeca property file.

At the first step, the atomic propositions of the formula are defined in the `define` section, considering the state variables of the actors. As depicted in Figure 3, five atomic propositions are defined for the bank contract example which examine balances for all clients (lines 2 to 8). The name of atomic propositions are `b1` to `b5` and their corresponding formula is put after the equal sign. In the `LTL` section correctness properties are specified (line 10). In this example, only one property with the name `Safety` is defined. Textual presentation of LTL modality *Always* ($\Box$) is `G` in Rebeca property files and conjunction between atomic propositions is shown by `&&`.

```
1  property {
2      define {
3          b1 = bank.balance[1] >= 0;
4          b2 = bank.balance[2] >= 0;
5          b3 = bank.balance[3] >= 0;
6          b4 = bank.balance[4] >= 0;
7          b5 = bank.balance[5] >= 0;
8      }
9      LTL {
10         Safety:G(b1 && b2 && b3 &&
11         b4 && b5);
12     }
13 }
```

Listing 3. The property file for the Rebeca code in Listing 2 stating the safety property as an LTL formula

III. Smart Rebeca for Smart Contracts

Smart contracts are a sequence of source codes that are sequentially executed in an atomic step. It means that when a user or a device asks for calling one of the functions of a contract, lines of the code will be executed in the written order and no other request will be processed until ending the execution of that function (i.e. none-preemptive execution). Also when an error or an exception happens in the code, performed changes are reverted. Considering this characteristic of smart contracts, it seems that they are easy to model, analyze, and synthesize. However, there are many challenges in modeling and analyzing smart contracts which have to be considered. For example, supporting all the built-in functions of abi package of Solidity or accessing block and transaction properties in Rebeca is not easy. This makes program verification of Solidity contract in Rebeca more challenging.

To address the mentioned challenges we develop Smart Rebeca as an extension of Rebeca with a set of annotations to be able to model Solidity features that are not supported in Rebeca. In the following sections we propose a mapping between Solidity language constructs and Smart Rebeca and clarify the parts of Solidity that are not supported by Smart Rebeca.

A. Mapping between Smart Rebeca and Solidity

We use the Solidity source code of a casino contract as the running example to show how Solidity smart contracts can be modeled using Rebeca, and what extensions to Rebeca are needed to be able to model smart contracts efficiently. By applying these extensions to Rebeca we build the Smart Rebeca language. The casino contract regulates how a casino should make a coin-tossing game available to the players. In this model, a player makes a guess and the last player who makes the correct guess wins the game. This contract is developed based on the following legal contract items, which are presented in [7].

- The casino owner may deposit or withdraw money from the casino's bank as long as the bank's balance never falls below zero.
- As long as no game is in progress, the owner of the casino may make available a new game by tossing a coin and hiding its outcome. The owner must also set a participation cost of choice for the game.
- The bank balance may never be less than the sum of the participation cost of the game and its win-out.
- The win-out for a game is set to be 80% of the participating cost.

The Solidity implementation of the smart contract of this legal contract is depicted in Listing 4. The function `placeBet` is invoked by external players to place a bet (lines 45-51). The functions `withdraw` (lines 15-23), `startTheGame` (lines 31-36), and `endTheGame` (lines 37-44) are invoked by the casino owner to manage the game life cycle. Note that in this model the function `tossACoin` (lines 25-30) is developed to model random number generation which should be replaced

with a fair external random number generation mechanism. This implementation ensures that the mentioned legal contract is never violated.

```
contract Casino {
address owner;
bool coinResult, guessedValue;
uint gameState, betValue;
uint public constant STATE_GAME_STOPPED=0;
uint public constant STATE_GAME_STARTED=1;
uint public constant STATE_GAME_BET_PLACED=2;
uint nonce = 10;
address public player;

function Casino() {
  owner = msg.sender;
  gameState = STATE_GAME_STOPPED;
}
function withdraw(uint amount){
  assert( msg.sender==owner);
  //if game's not stopped we require to have
      enough money to pay the player
  if(gameState!=STATE_GAME_STOPPED)
    assert(amount<=this.balance-18*betValue/
        10);
  else
    assert(amount<=this.balance);
  owner.transfer(amount);
}
// a pseudo-random function which should be
    replaced by an external service using
    Oraclize
function tossACoin() internal returns (bool){
  uint randomnumber = uint(keccak256(now,
      msg.sender, nonce)) % 900;
  randomnumber = randomnumber + 100;
  nonce++;
  return randomnumber>450;
}
function startTheGame(){
  assert( msg.sender==owner);
  coinResult = tossACoin();
  // For simplicity we don't use a Oraclize
      service for a random guess
  gameState = STATE_GAME_STARTED;
}
function endTheGame() {
  assert( msg.sender==owner);
  if(gameState==STATE_GAME_BET_PLACED) {
    if(guessedValue==coinResult)
      player.transfer(18*betValue/10);
    gameState = STATE_GAME_STOPPED;
  }
}
function placeBet(bool coinGuess) payable {
  assert(gameState==STATE_GAME_STARTED);
  player = msg.sender;
  betValue = msg.value;
  gameState = STATE_GAME_BET_PLACED;
  guessedValue = coinGuess;
}
}
```

Listing 4. Casino Contract in Solidity

In the Rebeca implementation of the casino model, the contract is modeled as the `Casino` reactive class which is extended from the `Contract` reactive class. We embedded a model of the blockchain inside Smart Rebeca `Contract` reactive class, which its simplified version is shown in Listing 5. The embedded model contains the bookkeeping (balances of the contracts, e.g. line 3) and Ether transfer mechanisms (`send` and `receive` functions in lines 8-14 and 15-17). This way, to model a money transfer inside any contract, developer only needs to call function `send`. If there is no `receive` function, the `Contract` reactive class mimics the Ethereum mechanism for money transfer by executing the fallback function.

```
1  abstract reactiveclass Contract (10) {
2    statevars {
3      int balance;
4    }
5    Contract(int startBalance) {
6      balance = startBalance;
7    }
8    boolean send(Contract receiver,int value){
9      if (value > balance)
10       return false;
11     balance -= value;
12     receiver.receive(value);
13     return true;
14   }
15   msgsrv receive(int value) {
16     self.balance += value;
17   }
18   void fallback() { }
19   void callback() { }
20 }
```

Listing 5. The `Contract` abstract class in Rebeca

Functions of the contract are implemented as message servers of `Casino` as they can be called asynchronously by the external actors. The simplified version of the Smart Rebeca model of Casino is shown in Listing 6. Besides, external actors have to be added as other reactive classes. In the case of Casino, they are `Player` and `CasinoOwner` reactive classes. The complete code of this model is available at Rebeca homepage [11].

```
1  env int GameStopped = 0;
2  env int GameStarted = 1;
3  env int BetPlaced = 2;
4  reactiveclass Casino extends Contract (10) {
5    knownrebecs {
6      CasinoOwner owner;
7    }
8    statevars{
9      int betValue, state;
10     boolean coinResult, guessedValue;
11     Player player;
12   }
13   Casino(int startBalance){
14     balance = startBalance;
15     betValue = 0;
16     state = GameStopped;
17   }
18   msgsrv withdraw(int amount){ ... }
19   msgsrv deposit(int amount) { ... }
20   msgsrv startTheGame() { ... }
21   msgsrv endTheGame() { ... }
```

```
22    msgsrv placeBet(int value, boolean guess)
23      { ... }
24  }
25  reactiveclass Player(10) { ... }
26  reactiveclass CasinoOwner(10){ ... }
27
28  main {
29    Casino casino(owner):(100);
30    CasinoOwner owner(casino):(100);
31    Player player(casino):(20);
32  }
```

Listing 6. The Casino contract in Rebeca

The syntax of Rebeca is Java-like and very similar to the syntax of Solidity and transforming the body of functions to message servers is straightforward. For instance, the Rebeca codes of `endTheGame` and `placeBet` are shown in Listing 7. Message servers and some statements of this implementation is annotated with some keywords which will be described in the following sections.

```
1  msgsrv endTheGame() {
2    @Assert(sender == owner)
3    if(state == BetPlaced){
4      if(guessedValue == coinResult)
5        transfer(betValue+(8*betValue) /10);
6      state = GameStopped;
7    }
8  }
9  @Payable
10 msgsrv placeBet(int value, boolean guess) {
11   @Assert(state == GameStarted)
12   player = (Player)sender;
13   betValue = value;
14   state = BetPlaced;
15   guessedValue = guess;
16 }
```

Listing 7. Detailed implementation of two message servers of Casino in Rebeca

B. Smart Rebeca Annotations

To support special features of Solidity in Rebeca, instead of extending the syntax of the language we added a set of annotations. In the following sections, the detailed description of the Solidity features which are supported by Smart Rebeca and their corresponding annotations in Smart Rebeca are presented.

1) Functions Visibility: Smart Rebeca only provides `public` and `private` visibility modifiers of Solidity. Solidity functions are defined as message servers in Smart Rebeca and are publicly available for the other actors. Methods which are called synchronously from message servers of a smart contract are assumed private.

2) Function Modifiers: From the function modifiers of Solidity, Smart Rebeca supports only `payable` modifier. We will talk about this modifier and how the payment mechanism is developed in Smart Rebeca in Section III-C. In Solidity, new modifiers can be defined to encapsulate the common behaviors of functions. Smart Rebeca supports these user defined function modifiers. For example, in the following code `onlyOwner` modifier is associated with `abort` function to make this function available only to the owner (line 5).

```
1  contract Auction {
2    modifier onlyOwner() {
3      require(msg.sender == seller);
4    }
5    function abort() public onlyOwner {
6      // ...
7    }
8  }
```

In Smart Rebeca, user defined modifier functions are implemented as private methods which are annotated with `@Modifier`. A message server which the defined modifier has to be associated with, uses `@Modifier` annotation with a parameter, as shown in line 7 in the following listing.

```
1  reactiveclass Auction extend Contract (5) {
2    @Modifier
3    void onlyOwner() {
4      @Require(sender != owner)
5      return;
6    }
7    @Modifier(onlyOwner)
8    msgsrv abort() {
9      // ...
10   }
11 }
```

3) Block Chain Related Global Variables: In the current version of Smart Rebeca, none of the block chain variables (i.e. variables in the form of `block.<func-name>`) are supported. Smart Rebeca only supports `sender` which is passed to message servers implicitly.

4) Encryption/Decryption Libraries: In the current version of Smart Rebeca, none of the `abi` functions and other independent functions like `sha256`, `keccak256`, etc, which are used for encryption and decryption are supported.

5) Error Handling: Two functions `assert` and `require` of Solidity are used to make sure that their given expressions are valid by evaluating them to true. They throw an exception if the condition is not met. The function `assert` using up all remaining gas in the failing condition and `require` will refund the remaining gas. These two functions are modeled with `@Require` or `@Assert` annotations. Line 2 of Listing 7 shows an example of using `@Assert`. Note that Smart Rebeca does not support `revert` error handling mechanism.

6) Events: To inform the external component about the execution of a contract function, a component can subscribe to an event and when the event is emitted, its corresponding code will be executed. In the following an event is defined in line 2 and it is called in line 5.

```
1  contract Auction {
2    event inc(address bidder, uint amount);
3    function bid() {
4      // ...
5      emit inc(msg.sender, msg.value);
6    }
7  }
```

This mechanism is implemented by defining an empty body method in Smart Rebeca and annotating it with `@Event`. Upon needs for emitting the event, the empty body method

is called and the call statement is annotated with `@Event` as well, as shown below.

```
reactiveclass Auction extend Contract (5) {
  @Event
  void inc(int bidder, int amount){}
  msgsrv bid() {
    // ...
    @Event
    inc(sender, value);
  }
}
```

7) Variable Types: Rebeca supports a limited number of variable types and its current version does not provide mechanism for defining new data types. However, Solidity supports a wide range of variable types, some of which are shown below.

```
contract Auction {
  uint a;
  address b;
  ether c;
  mapping(address => uint) votes;
}
```

In Smart Rebeca, variable types which have to be synthesized to Solidity special types, should be annotated with target types like below.

```
reactiveclass Auction extend Contract (5) {
  @uint int a;
  @address int b;
  @ether int c;
  @addressidx int votes[10];
}
```

The same can be done for all subdenomination of Ether, i.e. wei, szabo, and ether.

C. Payment in Smart Rebeca

Defining a function as `payable` allows it to receive ether while being called, as shown in the following code.

```
contract Auction {
  function bid() payable { ... }
}
```

The payable functions are defined in Smart Rebeca using `@Payable` annotation as shown below. The first parameter of a payable function in Smart Rebeca has to be in form of `int value` which shows the transferred value.

```
reactiveclass Auction extend Contract (5) {
  @Payable
  msgsrv bid(int value) { ... }
}
```

But, calling `payable` functions is not the only way of transferring money in smart contracts. In smart contracts and EVM, there are `send` and `transfer` functions for low level money transfer. Using them, results in call of `receive` method in the receiver side. Smart Rebeca only supports `send` and implement it in the `Contract` reactive class. A modeler can use these methods or overwrite `receive` in its own contract.

In Solidity, a function named `Fallback`, can be defined inside a contract (in Solidity this function has no name and is shown by two parentheses in the code) which is called if the method `receive` is not defined. Contracts that receive Ether directly (without a function call, i.e. using `send` or `transfer`) call fallback function which may lead to reentrancy attacks in smart contracts. The fallback function is also called when a contract receives a request for calling a function which does not exists.

In Smart Rebeca, as the reactive class `Contract` defines `receive` for receiving money and defined `fallback` as an empty body private method, the mentioned attack does not happen.

D. Synthesizing Solidity Codes from Smart Rebeca Models

Transforming Smart Rebeca models to Solidity code is straightforward as suggested in the above examples. Note that in the transformation, there is no need for transforming `Contract` reactive class as its functionality is provided by EVM.

IV. Analysis of Smart Rebeca

In the Blockchain, the order of executing transactions (requests for executing contract functions) is determined by the miners to increase their profit, and therefore it is somewhat random. This policy leads to problem of the transactions ordering and needs to be considered in the verification process. In addition, arbitrary execution of callback functions as a result of external service call and execution of fallback functions as a result of Solidity payment mechanism may causes concurrency issues. These concurrency issues have to be taken into account in verification.

We transform Smart Rebeca models to Rebeca models to perform formal verification. Transforming from Smart Rebeca to Rebeca is straightforward and only function modifiers and error handling notations have to be considered. In this transformation, error handing annotations are replaced with an `if` statement to mimic their behavior. The body of the user-defined function modifiers are put as the first statement in the body of message servers. For the case of payable functions, one statement is added to the beginning of the body of those functions to increase the balance of the contract with the given value for the `value` parameter.

There is no need for more modification in the transformation of a Smart Rebeca contract to Rebeca except for multi-contract models. In some cases, a contract is defined as a set of independent contracts. Message passing between contracts in Ethereum is usually synchronous. It means that in corresponding Smart Rebeca model of such a contract, there is a need for the synchronous call between actors which is not allowed. To resolve this issue, a multi-contract Smart Rebeca model is transformed to a Rebeca model in which the body of all of the contracts are embedded in one reactive class. This way, the required synchronous function calls are possible.

A. Concurrency in Smart Contracts

Referring to the sequential and non-preemptive execution of smart contracts, it seems that there is no concurrent behavior in smart contracts. But in many cases, smart contracts have to communicate with the services outside of the Blockchain to acquire data and run external services. Since connecting to an external service is time-consuming and IO errors may happen during the call, execution is performed asynchronously. This way of calling external services is called Oracelizing service call and may results in concurrency in execution [4].

We will illustrate concurrency problems using a gambling contract called BlockKing [4], a part of it is shown in Listing 8. The gamble in BlockKing works as follows. At any given time there is a designated "Block King" which is initially set to the writer of the contract. When money is sent to the contract by a gambler *s*, a random number is generated and if the current block number modulo 10 is equal to that number, then *s* becomes the new Block King. Afterward, a portion of the money in the contract (from 50% to 90%) is sent as the reward to the new Block King, and the remained money is set to the writer of the contract. In this model, there is a need for generating random numbers, which is difficult to be provided in deterministic systems. So, BlockKing uses a trusted external web service to generate unpredictable random numbers, Wolfram Alpha (an Oraclize service). In this contract the `enter` function is called by the player and the bet money is sent to the contract (lines 1-16). After setting the contract variables (lines 10-12), a request is sent to the Oraclize service (lines 13-15) and the function is terminated. The called external service checks the request and after generating the random number calls the callback function in the contract with the generated number as the parameter. Meanwhile, many other functions of the contract may be executed, and some blocks and the state of the Blockchain may be changed. So, there is no guarantee that the state of the BlockKing contract at the time of callback be the same as its state at the time of Oraclize service call.

```
1  function enter() payable {
2    /* 100 finney = .05 ether minimum payment
3     * otherwise refund payment and stop
4     * contract
5     */
6    if (msg.value < 50 finney) {
7      msg.sender.send(msg.value);
8      return;
9    }
10   warrior = msg.sender;
11   warriorGold = msg.value;
12   warriorBlock = block.number;
13   bytes32 myid =
14   oraclize_query(0,"WolframAlpha",
15     "random number in [1, 9]");
16 }
17 function __callback(bytes32 myid,
18         string result) {
19   if (msg.sender != oraclize_cbAddress ())
20     throw;
21   randomNumber = uint(bytes(result)[0])-48;
22   process_payment ();
23 }
24 function process_payment() {
25   ...
26   if (singleDigitBlock == randomNumber) {
27     rewardPercent = 50;
28     // If the payment was more than .999
29     // ether then increase reward percentage
30     if (warriorGold > 999 finney) {
31       rewardPercent = 75;
32     }
33     king = warrior;
34     kingBlock = warriorBlock;
35   }
36 }
```

Listing 8. Smart BlockKing Contract

There is a concurrency bug in this implementation of BlockKing contract. Assume that multiple gamblers wish to try their luck and attempt to play in a short period of time. The presented implementation of this contract does not have a mechanism for keeping track of different players and the newly entered player overwrites the values of `warriorBlock` and `warrior` in lines 10 to 12. This way, the contract only knows the information regarding the last player. So, every time the external service calls the callback function, only the last player is considered for a chance of winning; although, the other players paid for it. The problem lies in lines 24 to 32 of `process_payment`, which is called at the end of the `__callback` function. Note that in Smart Rebeca, the reactive class `Contract` has an empty body `callback` function which mimics the behavior of the `__callback` function.

Modeling of these kinds of behaviors can be performed efficiently in Smart Rebeca, because it is designed for modeling concurrent behaviors and a modeler can easily model not only the contract logic but also service providers and external services. For instance, in BlockKing, a modeler can implement the contract logic, users, and external service as different reactive classes, then defines the correctness properties and check the correctness.

B. Arbitrary Order of Execution

The fact that transactions and function calls are executed in a nondeterministic order decided by miners may cause concurrency problems. Model checking smart contracts can reveal such potential problems. A newly issued request for executing a function may be executed before the previously issued requests. This nondeterminism may affect the correctness of a contract when the logic of the contract depends on the outcome of more than one request and the ordering is significant.

For instance, assume that there is a bank contract and the bank manager wants to add the yearly profit to the saving accounts. He wants to add 5 percent profit to the account, which can be implemented by multiplying the balances of accounts by 1.05. Also, assume that there is a customer who wants to withdraw 10 dollars from his account which its balance is 100 dollars. Now in a scenario that these two

requests are waiting for execution on Blockchain, the balance of the customer account may have two different values since the order of the transactions cannot be determined no matter which request comes earlier (one of 94.5 or 95 dollars). This was an example of how racing condition affects the outcome of a contract. This type of nondeterminism is considered by Afra. In the case of having more than one message to handle, model checkers chooses one of the messages nondeterministically and handle it.

V. Related Work

A. Smart Contracts and Solidity

In the recent years, several programming languages have been defined on top of Ethereum virtual machine (EVM) platform. For example, Bitcoin script is a non-Turing complete language, which allows implementing a limited form of smart contracts. In particular, this language supports some basic arithmetic and logical operators, but no loops, and it suffers from low expressiveness [15]. Ivy [16] is another Bitcoin language, which is more secure than Bitcoin script. In fact, by constraining the expressiveness of Bitcoin script as well as introducing financial asset and transaction notions as first-class concepts, it renders large classes of potential smart contract vulnerabilities simply impossible by design. The functional language Simplicity is another notable language for designing smart contracts that are secure by design [17]. Simplicity is a functional language without loops and recursion, which comes with formal denotational semantics defined in Coq. This language is still under development. Vyper [18] is another Ethereum language, which is designed to be secure and simple. Following these goals, this language does not provide support for some features such as modifiers, inheritance and overloading. As another Ethereum language, Bamboo [19] is defined to provide support for formal verification. In particular, this language makes state transitions explicit, which is beneficial for model-checking purposes. Developers define which functions can be called in each state, and the language provides constructs to specify changes of state explicitly. However, Bamboo does not present any additional features geared towards the safety of programs [20].

Among the languages designed to target Ethereum, Solidity [6] is the most mature and popular which we described it in detail in Section II.

B. Verification of Smart Contracts

The semantics of EVM bytecode is formalized in the F* proof assistant in [21], obtaining executable code that is validated against the official Ethereum test suite. Furthermore, the authors have formally defined a number of central security properties for smart contracts, such as call integrity, atomicity, and independence from miner controlled parameters. Luu et al. presented Oyente [3], a state-of-the-art static analysis tool for EVM bytecode that relies on symbolic execution. Oyente comes with a semantics of a simplified fragment of the EVM bytecode and, in particular, misses several important commands related to contract calls and contract creation. Oyente supports a variety of security properties, such as transaction order dependency, timestamp dependency, and reentrancy, but the security definitions are rather syntactic and are described informally. Brent et al. [22] introduce a security analysis framework for Ethereum contracts, called Vandal, which converts EVM bytecode to semantic relations, which are then analyzed to detect vulnerabilities described in the Souffle language. Note that all of the above mentioned approaches consider EVM bytecodes which may make the analysis more complicated.

On the other hand, VeriSolid [23] is a framework for formal verification of smart contracts specified using a transition-system based model. This framework provides natural-language-like templates for specifying safety and liveness properties. However, the need to be familiar with the concept of transition systems and state machines may limit the adoption of this framework. Tesnim et al. [24] propose an approach to model smart contract and Blockchain execution protocol along with users' behaviors using the BIP framework. BIP [25] includes a strong component-based modeling formalism and a statistical model checking engine for systems verification. The latter two may not be friendly for the practitioners working with the existing programming languages and Solidity. In [8], the authors propose a method to translate Solidity programs to PROMELA models. The SPIN model checker is then used to verify the correctness properties. They do not discuss how they handle callback and fallback functions which are essential for developing real-world smart contract.

C. Synthesis of Smart Contracts from Models

In addition to the verification facilities, VeriSolid also supports generation of Solidity code from the verified models. In [26], the authors present a tool to model smart contracts as finite state machines (FSM), which are then transformed to Solidity code automatically. They also introduce a set of design patterns, which they implement as plugins that developers can easily add to their contracts to enhance security and functionality. UML statecharts are used in [27] to model contract behaviors. The statechart models are then translated to Solidity code based on some predefined mapping rules.

In [1] an algorithm is proposed to translate ADICO models to Ethereum smart contracts. ADICO allows behavior specification in terms of human-readable statements. The currently generated smart contract skeletons using ADICO require significant amount of manual inputs to enable them to be executable in EVM.

VI. Conclusion

In this paper, we present an approach for modeling and verification of smart contracts using Smart Rebeca and show how modeling and analysis challenges are addressed. We illustrate that the model of computation of Smart Rebeca and smart contracts developed in Solidity are very close. We make it clear that statements in Smart Rebeca and Solidity are similar, so, the transformation is straightforward. We show how annotations can be used to cover the limitations of

Smart Rebeca for modeling Solidity contracts. To illustrate the applicability of our approach, we present a set of Solidity smart contracts and discuss how issues that are caused by concurrency and nondeterminism can be revealed by model checking.

In the future, we plan to extend Afra to support the automatic transformation from Solidity to Smart Rebeca. We also plan to develop a more comprehensive set of case studies to provide samples for Solidity contract developers to support them in the modeling phase.

ACKNOWLEDGMENT

The work of the third and forth authors is supported in part by KKS SACSys Synergy project (Safe and Secure Adaptive Collaborative Systems), and KKS DPAC Project (Dependable Platforms for Autonomous Systems and Control) at Malardalen University, and MACMa Project (Modeling and Analyzing Event-based Autonomous Systems) at Software Center, Sweden.

REFERENCES

[1] C. Frantz and M. Nowostawski, "From institutions to code: Towards automated generation of smart contracts," in *2016 IEEE 1st International Workshops on Foundations and Applications of Self* Systems (FAS*W), Augsburg, Germany, September 12-16, 2016*, 2016, pp. 210–215.

[2] N. Atzei, M. Bartoletti, and T. Cimoli, "A survey of attacks on ethereum smart contracts (sok)," in *Principles of Security and Trust - 6th International Conference, POST 2017, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2017, Uppsala, Sweden, April 22-29, 2017, Proceedings*, 2017, pp. 164–186.

[3] L. Luu, D. Chu, H. Olickel, P. Saxena, and A. Hobor, "Making smart contracts smarter," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, Vienna, Austria, October 24-28, 2016*, 2016, pp. 254–269.

[4] I. Sergey and A. Hobor, "A concurrent perspective on smart contracts," in *Financial Cryptography and Data Security - FC 2017 International Workshops, WAHC, BITCOIN, VOTING, WTSC, and TA, Sliema, Malta, April 7, 2017, Revised Selected Papers*, 2017, pp. 478–493.

[5] E. A. Lee, "The problem with threads," *IEEE Computer*, vol. 39, no. 5, pp. 33–42, 2006.

[6] Ethereum, "The Solidity contract-oriented programming language." https://github:com/ethereum/solidity.

[7] W. Ahrendt, R. Bubel, J. Ellul, G. J. Pace, R. Pardo, V. Rebiscoul, and G. Schneider, "Verification of smart contract business logic - exploiting a java source code verifier," in *Fundamentals of Software Engineering - 8th International Conference, FSEN 2019, Tehran, Iran, May 1-3, 2019, Revised Selected Papers*, 2019, pp. 228–243.

[8] T. Osterland and T. Rose, "Model checking smart contracts for ethereum," *Pervasive and Mobile Computing*, pp. 101–129, 2020.

[9] M. Sirjani, A. Movaghar, A. Shali, and F. S. de Boer, "Modeling and Verification of Reactive Systems using Rebeca," *Fundam. Inform.*, vol. 63, no. 4, pp. 385–410, 2004.

[10] M. Sirjani and M. M. Jaghoori, "Ten years of analyzing actors: Rebeca experience," in *Formal Modeling: Actors, Open Systems, Biological Systems - Essays Dedicated to Carolyn Talcott on the Occasion of Her 70th Birthday*, 2011, pp. 20–56.

[13] M. M. Jaghoori, M. Sirjani, M. R. Mousavi, E. Khamespanah, and A. Movaghar, "Symmetry and partial order reduction techniques in model checking rebeca," *Acta Inf.*, vol. 47, no. 1, pp. 33–66, 2010.

[11] "Rebeca Formal Modeling Language," http://www.rebeca-lang.org/.

[12] M. Wohrer and U. Zdun, "Smart contracts: security patterns in the ethereum ecosystem and solidity," in *2018 International Workshop on Blockchain Oriented Software Engineering, IWBOSE@SANER 2018, Campobasso, Italy, March 20, 2018*, 2018, pp. 2–8.

[14] M. Sirjani, E. Khamespanah, and F. Ghassemi, "Reactive actors: Isolation for efficient analysis of distributed systems," in *23rd IEEE/ACM International Symposium on Distributed Simulation and Real Time Applications DS-RT 2019, Cosenza, Italy, October 7-9, 2019*, 2019, pp. 1–10.

[15] M. Bartoletti and L. Pompianu, "An empirical analysis of smart contracts: Platforms, applications, and design patterns," in *Financial Cryptography and Data Security - FC 2017 International Workshops, WAHC, BITCOIN, VOTING, WTSC, and TA, Sliema, Malta, April 7, 2017, Revised Selected Papers*, 2017, pp. 494–509.

[16] Chain Inc., "Ivy: A high-level language and ide for writing bitcoin smart contracts," https://github.com/ivy-lang/ivy-bitcoin.

[17] R. O'Connor, "Simplicity: A new language for blockchains," in *Proceedings of the 2017 Workshop on Programming Languages and Analysis for Security, PLAS@CCS 2017, Dallas, TX, USA, October 30, 2017*, 2017, pp. 107–120.

[18] Ethereum, "Vyper: New experimental programming language," https: //github:com/ethereum/vyper.

[19] Y. Hirai, "Bamboo: an embryonic smart contract language," https: //github.com/pirapira/bamboo.

[20] F. Schrans, S. Eisenbach, and S. Drossopoulou, "Writing safe smart contracts in flint," in *Conference Companion of the 2nd International Conference on Art, Science, and Engineering of Programming, Nice, France, April 09-12, 2018*, 2018, pp. 218–219.

[21] I. Grishchenko, M. Maffei, and C. Schneidewind, "A semantic framework for the security analysis of ethereum smart contracts," in *Principles of Security and Trust - 7th International Conference, POST 2018, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2018, Thessaloniki, Greece, April 14-20, 2018, Proceedings*, ser. Lecture Notes in Computer Science, L. Bauer and R. Küsters, Eds., vol. 10804. Springer, 2018, pp. 243–269.

[22] L. Brent, A. Jurisevic, M. Kong, E. Liu, F. Gauthier, V. Gramoli, R. Holz, and B. Scholz, "Vandal: A scalable security analysis framework for smart contracts," *CoRR*, vol. abs/1809.03981, 2018.

[23] A. Mavridou, A. Laszka, E. Stachtiari, and A. Dubey, "Verisolid: Correct-by-design smart contracts for ethereum," in *Financial Cryptography and Data Security - 23rd International Conference, FC 2019, Frigate Bay, St. Kitts and Nevis, February 18-22, 2019, Revised Selected Papers*, ser. Lecture Notes in Computer Science, I. Goldberg and T. Moore, Eds., vol. 11598. Springer, 2019, pp. 446–465.

[24] T. Abdellatif and K. Brousmiche, "Formal verification of smart contracts based on users and blockchain behaviors models," in *9th IFIP International Conference on New Technologies, Mobility and Security, NTMS 2018, Paris, France, February 26-28, 2018*. IEEE, 2018, pp. 1–5.

[25] A. Basu, M. Bozga, and J. Sifakis, "Modeling heterogeneous real-time components in BIP," in *Perspectives Workshop: Model Engineering of Complex Systems (MECS), 10.08. - 13.08.2008*, 2008.

[26] A. Mavridou and A. Laszka, "Designing secure ethereum smart contracts: A finite state machine based approach," *CoRR*, vol. abs/1711.09327, 2017.

[27] P. Garamvolgyi, I. Kocsis, B. Gehl, and A. Klenik, "Towards model-driven engineering of smart contracts for cyber-physical systems," in *48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks Workshops, DSN Workshops 2018, Luxembourg, June 25-28, 2018*. IEEE Computer Society, 2018, pp. 134–139.