Automatic Conversion of Smart Contracts for Non-Blocking Verification

Citation for the original published paper (version of record):
Parekh, N., Ahrendt, W., Fabian, M. (2024). Automatic Conversion of Smart Contracts for Non-Blocking Verification. IFAC-PapersOnLine, 58(1): 282-287.
http://dx.doi.org/10.1016/j.ifacol.2024.07.048

Available online at www.sciencedirect.com

ScienceDirect

IFAC PapersOnLine 58-1 (2024) 282–287

Automatic Conversion of Smart Contracts for Non-Blocking Verification

Nishant Parekh * **Wolfgang Ahrendt** ** **Martin Fabian** *

* *Department of Electrical Engineering*
** *Department of Computer Science and Engineering*
Chalmers University of Technology, Göteborg, Sweden
{*nishantp, ahrendt, fabian*}*@chalmers.se*

Abstract: Smart contracts are programs stored on a blockchain ledger, thus being immutable after deployment, which makes assessment of their correctness before deployment vital. Extended finite state machines (EFSM) offer a structured framework for modeling complex systems, thus providing a systematic approach to scrutinize smart contract functionalities. This paper describes a methodology to automatically convert from the abstract syntax tree of a smart contract to an EFSM model. A smart contract implementing a casino is the specific use case, and verification of the EFSM model reveals it to be blocking. This blocking represents that a malicious player can lock the funds of the casino so that they can never be retrieved.

Keywords: Extended finite state machines, smart contracts, security, verification, non-blocking

1. INTRODUCTION

In the previous decade, smart contracts have emerged as an integral part of the blockchain ecosystem. Eliminating the need for intermediaries, smart contracts can enforce agreements among mutually distrusting parties. As smart contracts become increasingly capable of handling more complex interactions and transactions, the potential for errors and vulnerabilities increases. Even if the underlying blockchain protocols cannot feasibly be compromised, a smart contract can itself allow behavior, unintended by the programmer, that may be exploited to the disadvantage of some of the users. Since smart contracts, once deployed on the blockchain, are immutable, assessing their correctness beforehand is crucial.

Formal methods is one technique that allows for rigorous analysis and can guarantee correctness. Formal methods are a set of mathematical techniques used in specification, design and verification of software and hardware systems. Model checking (Baier and Katoen, 2008), one of several formal methods, allows for verifying correctness of a finite state system by evaluating the state space of a transition system against given specifications. One significant advantage of modelling smart contract behaviour as extended finite state machines (EFSMs) (Skoldstam et al., 2007) is that it allows to use model checking techniques to verify its correctness. Smart contracts can be modelled as EFSMs by abstracting their high-level behaviour, ignoring intermediate execution details. However, manually modelling smart contract behavior as EFSMs can be labour intensive and prone to error and bias.

Related work towards modelling smart contracts and their verification is presented by Fekih et al. (2022), that describe modelling smart contracts as EFSMs and verifying them using the nuXmv (Cavada et al., 2014) model checker. However, in the abovementioned work, EFSM models of smart contracts are generated manually. Godoy et al. (2022) present an approach to assist in validation of smart contracts using predicate abstraction, focusing on generating models by abstracting smart contract behaviour at function call level.

Early works by Bhargavan et al. (2016) present a study analysing functional correctness and runtime safety of smart contracts by translating them to F* programs. Modelling of smart contract as PROMELA models and verifying them using SPIN (Holzmann, 1997) is presented in Bai et al. (2018). Works of Suvorov and Ulyantsev (2019) and Mavridou and Laszka (2018) explore strategies aimed at synthesis of secure smart contracts from EFSMs that fulfill requirement specifications. Another approach presented by Madl et al. (2019) investigate using interface automata for verification purposes.

The recent work of Mohajerani et al. (2022) shows how state based smart contracts can be manually modelled as EFSMs, following which *supervisory control theory* (SCT) (Ramadge and Wonham, 1989) can be applied for verification of non-blocking behaviour.

This paper presents a set of principles for modelling smart contracts as EFSMs automatically from the source code. These principles are attributed towards modelling variables, modifiers, functions and generic framework behaviour. As a running example, the simple Casino smart contract treated also by Mohajerani et al. (2022) is used. Just as in Mohajerani et al. (2022), formal verification of the generated EFSM model reveals it to be blocking.

The paper is structured as follows: Section 2 provides a brief background on the smart contract implementation language Solidity, EFSMs and Supremica. Section 3

⋆ This work was partially supported by the Wallenberg AI, Autonomous Systems and Software Program (WASP) funded by the Knut and Alice Wallenberg Foundation.

10.1016/j.ifacol.2024.07.048

describes the Casino contract, and Section 4 examplifies the automatic conversion from the source code to EFSMs. Section 5 briefly presents the non-blocking verification of the EFSM model, while Section 6 concludes the paper.

2. BACKGROUND

2.1 Smart Contracts: Ethereum and Solidity

The first, and still major, blockchain framework supporting smart contracts is Ethereum (Wood, 2023), with its built-in cryptocurrency Ether. In Ethereum not only the users, but also the contracts can receive, own, and send Ether. Sending Ether to a contract, and calling the contract, is the same thing in this framework. Sending funds to a contract without passing control to the receiver is not possible. Ethereum miners look for transaction requests on the network. A transaction request contains the address of a contract to be called, the call data, and the amount of Ether to be sent. Miners are paid for their efforts with units of (Ether prised) *gas*, to be paid by the address that requested the transaction.

A transaction may not necessarily be executed successfully. It can be reverted for various reasons: running out of gas, sending of unbacked funds, failing runtime assertions, or a simple revert statement in the code. If the miner attempts to execute a top-level (i.e., externally triggered) transaction, a reverting action anywhere inside the transaction execution will *undo the entire transaction*, all the way up the call stack. All the effects so far are also undone (except for the paid gas), as if the original call never happened. For instance, consider a case where a user calls some smart contract C, which during execution of the request sends Ether to another contract D. Recall that sending funds from C to D means that control is passed, for the moment, from C to D, and C can only resume once D returns. If D aborts before returning, the entire original request from the user gets undone.

The most popular programming language for Ethereum smart contracts is Solidity [1], which follows largely an object-oriented paradigm. Each external user and each contract instance has a unique `address`. Each `address` owns Ether (possibly 0), can receive Ether, and send Ether to other addresses. For instance, `a.transfer(v)` transfers the amount of v Wei ($= 10^{-18}$ Ether) from the caller to a. Built-in data types include unsigned integer (`uint`), enums, and structs. The current caller, and the amount of Wei sent with the call, are always available via `msg.sender` and `msg.value`, respectively. Only `payable` functions accept payments. `require(b)` checks the boolean expression b, and aborts if b is false. Fields marked `public` are read-public, not write-public. Solidity offers also some cryptographic primitives, for instance the function `keccak256` computing a crypto-hash of its argument.

Solidity further features programmable *modifiers*. For instance, the Casino contract in Fig 1 uses the modifiers `byOperator`, `inState(s)`, and `noActiveBet`. These modifiers expand to `require` statements that abort the transaction if not fulfilled. The above modifiers expand to, respectively:

- `require (msg.sender == operator);`
- `require (state == s);`
- `require (state != State.BET_PLACED);`

[1] https://docs.soliditylang.org/en/latest/

2.2 Extended Finite State Machines

EFSMs extend finite-state machines with variables, and guard and action expressions associated with transitions. A guard is a predicate for the transition, which when true allows the transition to occur. The action, if specified for a transition, then updates the variables of the action. An EFSM is defined by the tuple $E = \langle \Sigma, S, S^\circ, \rightarrow, S^\omega \rangle$ where:

- Σ is the finite set of labels representing events.
- S is a finite set of locations, with $S^\circ \subseteq S$ representing the set of initial locations.
- $\rightarrow$ represents the transition relation. A transition $s \xrightarrow{\sigma:g:a} s'$ associated with event σ, where $s, s' \in S$, means that the system can transit from location s to s' if the guard g evaluates to *true*, and then the action a is performed.
- $S^\omega \in S$ is the set of marked locations.

EFSMs are illustrated as directed graphs with nodes representing locations and arrows representing transitions. Events, guards and actions associated with a transition are represented by labels and expressions on the transition. In guards, the post-transition value of a variable is denoted by a prime, while the pre-transition value is un-primed. For instance, in Fig 2, where {`S0, S1, S2`} is the set of locations, the transition `S1` to `S2` is guarded by the expression (`guess' == HEAD | guess' == TAILS`), and the action function (`bet = value`) is executed if the transition occurs. Thus, after the transition, the value of `guess` is either `HEADS` or `TAILS`, and the variable `bet` is assigned the value of the variable `value`.

Interaction between EFSMs is modelled by *synchronous composition* (Hoare, 1985) of *shared events*, that is, events that appear in the alphabets of more than one composed EFSMs. A shared event is enabled in the composition, only if it is enabled by all EFSMs containing that event in their alphabet. For example, the event `placeBet1` of Fig 2 is synchronized with the EFSM modeling the `require` clause of Fig 5, so that the transition from `S0` to `S1` in Fig 2 cannot occur unless `sender` is different from the `operator`.

2.3 Supremica

Supremica (Akesson et al., 2006) is a tool for synthesis, simulation, and verification of discrete event systems. Efficient algorithms for assessing and guaranteeing well-known SCT properties such as controllability and non-blocking are implemented in Supremica. In this paper, the compositional abstraction-based non-blocking (Malik et al., 2023) verification algorithm is used. Non-blocking is a progress property that, when fulfilled, guarantees that some significant *marked* state(s) of the system can always be reached. In the context of this paper, this aims to guarantee the ability of the Casino smart contract to always reach its initial state.

To verify non-blocking, *conflict check* (Akesson et al., 2006) of Supremica is used. If the verification determines

that the system is blocking, a counter-example is provided, a trace that leads to a blocking situation. This counter-example may be replayed in SUPREMICA's simulator to reveal the core of the problem.

3. RUNNING EXAMPLE

The Solidity code of the Casino contract (originally proposed by Gordon Pace) is shown in Fig 1 [2]. The implementation features three explicit states: `IDLE`, `GAME_AVAILABLE`, and `BET_PLACED`, see line 3, defined by the `enum` type `State`.

Based on the modifier `inState(s)`, in the `IDLE` location the operator may create a game by invoking the `createGame` function (line 9). To ensure a fair betting, the Casino must place its bet at the time of game creation. Thus, when calling `createGame`, `hashedNumber` is assigned a value (line 11) to later decide the game outcome. After creating a new game, the state changes to `GAME_AVAILABLE` (line 12) where a game is now available. In this state, the player can call `placeBet` to place a bet, up to the size of the pot, on HEADS or TAILS (lines 16-21). This then changes the state of the contract to `BET_PLACED` (line 23).

Next, the operator may by `decideBet` submit the original secret number to resolve the bet (line 25). If the secret number is even the coin toss is HEADS, else it is TAILS (line 29). If the player wins, the original bet is set to zero and only the bet amount is deducted from the pot representing the sum lost by the casino (lines 39–40). Then, double the bet is transferred from the contract to the player (line 41). If the operator wins, the bet is added to the pot and then set to zero (lines 44–45).

The operator may add money to the pot at any state, `addToPot` (line 47). Also, the operator may remove money from the pot, `removeFromPot` (line 51), but only if the player has not placed a bet, that is, if the casino is not in the state `BET_PLACED`. This is ensured by the modifier `noActiveBet`.

4. CONVERSION TO EFSM

The automated conversion traverses the source code's *abstract syntax tree* (AST), which is obtained in JSON format from the official Solidity compiler `solc` by the command `solc --ast-compact-json`. The AST consists of nodes of designated types corresponding to specific Solidity constructs, such as *FunctionDefinition*, *FunctionCall*, *VariableDeclaration* etc. Each such node can itself contain nodes in a hierarchy. The conversion recursively mines the AST for data relevant for generating the EFSMs. For each node type, specific code is executed and the conversion is kept as "local" as possible, meaning no global overview of parts of the code is necessary; only in the case of the *Conditional* on line 29 is it necessary to pass data from a higher hierarchy level to a lower.

The automatically converted EFSM model differs significantly from the manually crafted model presented by Mohajerani et al. (2022). The main difference being that the `State` variable is in the manual model represented by a specific EFSM, which embeds the control of the other functions, thus making the modifiers redundant. In the automatically converted model, `State` is represented by an EFSM variable `state`, much like in the Solidity code, and the modifiers are thus explicitly modeled.

[2] Slightly simplified, for a detailed presentation see https://verifythis.github.io/02casino/

```
1  contract Casino {
2
3    enum State {IDLE, GAME_AVAILABLE,
         BET_PLACED}
4    State private state;
5    address public operator, player;
6    bytes32 public hashedNumber;
7    struct Wager {uint bet; Coin guess;}
8
9  function createGame(bytes32 hashNum) public
10   byOperator, inState(State.IDLE) {
11   hashedNumber = hashNum;
12   state = State.GAME_AVAILABLE;}
13
14 function placeBet(Coin _guess) public payable
15   inState(State.GAME_AVAILABLE) {
16   require (msg.sender != operator);
17   require (msg.value > 0 && msg.value <= pot);
18   player = msg.sender;
19   wager = Wager({
20     bet: msg.value,
21     guess: _guess
22   });
23   state = State.BET_PLACED;}
24
25 function decideBet(uint secretNumber) public
26   byOperator, inState(State.BET_PLACED) {
27   require (hashedNumber ==
28         keccak256(secretNumber));
29   Coin secret = (secretNumber % 2 == 0)?
         Coin.HEADS : Coin.TAILS;
30   if (secret == wager.guess) {
31     playerWins();
32   } else {
33     operatorWins();
34   }
35   state = State.IDLE;}
36
37 function playerWins() private {
38   tmp = wager.bet;
39   wager.bet = 0;
40   pot = pot - tmp;
41   player.transfer(tmp*2);}
42
43 function operatorWins() private {
44   pot = pot + wager.bet;
45   wager.bet = 0;}
46
47 function addToPot() public payable
48   byOperator {
49   pot = pot + msg.value;}
50
51 function removeFromPot(uint amount) public
52   byOperator, noActiveBet {
53   pot = pot - amount;
54   operator.transfer(amount);}
55 }
```

Fig. 1. Solidity code for Casino (some details are omitted).

4.1 Modeling variables

SUPREMICA allows bounded, discrete variables, so Solidity variables are directly converted to SUPREMICA variables. The *constructor* (not shown in Fig 1) assigns initial val-

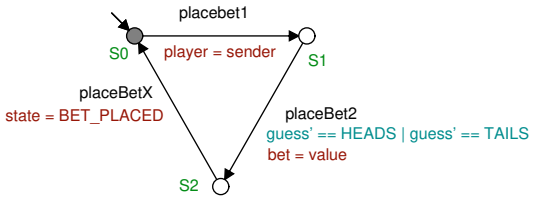

Fig. 2. EFSM model of the `placeBet` function.

ues to some variables, and these are initialized accordingly in SUPREMICA. Most variables have unknown initial values, though, which is in SUPREMICA modeled by non-deterministic initial value assignments over the entire range of the variable domain.

Some care has to be taken, though, when converting the unbounded Solidity types to bounded SUPREMICA variables. Particularly, the `pot` variable cannot be modeled since if `pot` is modeled as an (upper) bounded variable, then a trivial blocking trace calls `addToPot` enough times for `pot` to reach its upper bound, plus once more, and then the system deadlocks. The `pot` variable is automatically ignored by adding it to an *ignore list*, so the converter does not generate any `pot` variable in the EFSM model, nor any transitions for statements involving `pot` (lines 17, 40, 44, 49, 53).

4.2 Modeling Functions

Each function of Fig 1 is modeled as a separate EFSM, typically interacting with other EFSMs through shared events. Generally, an EFSM modeling a function has one transition for each statement, which roughly corresponds to each line of the code in Fig 1. From its initial location, the EFSM has a transition labeled by the *initial event*, which is constructed from the function name, appended with the number `1`. The EFSM also has a *final event* that labels a transition back to its initial location; this label is constructed by appending the function name with `X`. This naming scheme guarantees that it is known beforehand which events denote the call and return, respectively, of a function, so that these events can be used even before a function has been modeled.

The EFSM model of the `placeBet` function is shown in Fig 2; this models lines 14–23 of Fig 1. The initial event `placeBet1` labels the transition from the initial location `S0`, and the assignment of `player` on line 18 is added as an action. The handling of the modifiers and the `require` clauses, lines 15–17 are described in Section 4.3, below.

Inside the `placeBet` function there is an assignment to the `wager` variable (lines 19–21), which is of type `struct Wager` (line 7). Since there are no `structs` in SUPREMICA, the struct constructor call of `wager` has to be "flattened". This is automatically done, and results in two distinct variables `guess` and `bet`.

The `placeBet` function is called with a parameter `_guess` of type `Coin`, which is an `enum` (not shown in Fig 1, but see line 29) with two values, HEADS and TAILS. The converter collects this type information, and since the actual value of `_guess` is unknown for the `placeBet` call, a non-deterministic assignment is made to the variable `guess`, see the guard on the transition labeled `placeBet2` of Fig 2.

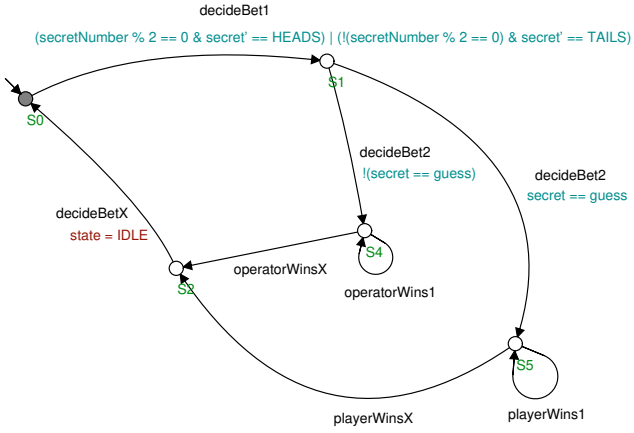

Fig. 3. EFSM model of the `decideBet` function.

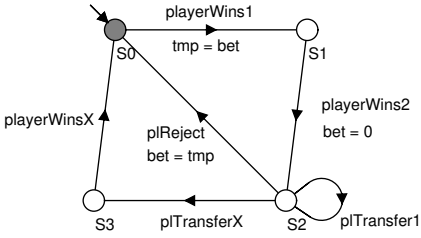

Fig. 4. EFSM model of the `playerWins` function.

When a function is called from within another function, this is modeled by a selfloop labeled by the called function's initial event, and with the called function's final event labeling a transition from the selflooped location. This is illustrated in Fig 3, where the `operatorWins` and `playerWins` functions are called in the locations labeled `S4` and `S5`, respectively. This corresponds to lines 33 and 31, respectively. When the EFSM of Fig 3 is in, say, `S5`, it cannot transit to `S2` until the `playerWinsX` event is enabled, which requires the EFSM that models `playerWins` (Fig 4) to first transit on its initial event, `playerWins1` and then go through its other transitions until both EFSMs synchronously transit on the `playerWinsX` event. In this way, `placeBet` initiates the execution of `playerWins`, and then waits for `playerWins` to return.

Note that though lines 19–22, and 28 are in the AST designated as *FunctionCall*, these are treated separately and do not result in selfloops. The initialization of the `wager` struct variable, is described above. The external hashing function `keccak256` (line 28) is not modelled at all, as its internal workings are not known. This is handled by adding `keccak256` to the abovementioned *ignore list*, so that the call `keccak256(secretNumber)` is automatically converted into simply `secretNumber`.

4.3 Modifiers and require statements

Modifiers and `require` statements are modeled as single-location EFSMs, with selflooped transitions with the Boolean expression as guard and labeled by the initial events of the functions that the modifiers and/or `require` clauses relate to. For instance, the `byOperator` modifier relates to the `createGame`, `addToPot`, `removeFromPot`, and `decideBet` functions, and so the selflooped transition is

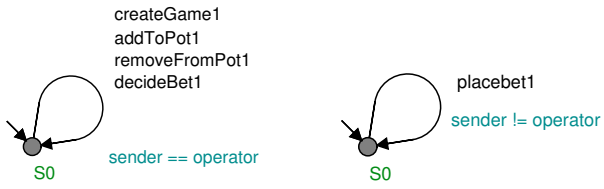

Fig. 5. EFSM models of (left) the `byOperator` modifier, and (right) the `require` clause on line 16.

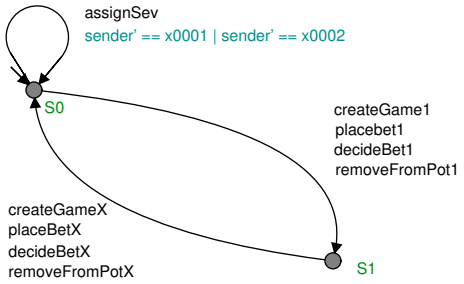

Fig. 6. EFSM model of the assignment of `sender`.

labeled by their initial events, see Fig 5, left. Similarly, the `require` clause of line 16 of Fig 1 appears only in `placeBet`, and so has only the event `placeBet1`, see Fig 5, right.

A unique naming scheme is essential to guarantee that modifiers and, especially, `require` statements are not accidentally named the same. The current implementation of the conversion uses a hashing function to create a unique (with high probability) name for the EFSMs based on the logical expression of their guards.

4.4 Modeling framework behavior

The above discussion and models deal with what can be converted directly from the Solidity source code of Fig 1. However, this is not enough to have a useful model, as this code executes within the Ethereum framework, which adds some behavior of its own that is necessary to capture. Specifically, this concerns the assignment to addresses of the `msg.sender` variable, and the behavior of `transfer`. As this behavior is not possible to extract from the code, these models are manually predefined, but in a generic way.

Assign Sender Within the Solidity framework, contracts interact with each other by calling public functions. With each such call follows a data packet `msg`, which includes among other things a reference (address), `msg.sender`, to the contract that called the function. The assignment to `msg.sender` of the reference to the caller is handled by the framework, outside of the Solidity code. In the Casino case there are two participants, `player` and `operator`, and the behavior of the public functions depends on which of the participants that called it, so the EFSM model must include a model for the assignment of `msg.sender`. This is done by the EFSM of Fig 6.

The selfloop in the `S0` location, labeled by the `assignSev` event, non-deterministically assigns the variable `sender` the "address" `x0001` (player) or `x0002` (operator). Since not much can happen with the Casino until the operator has created the game, see lines 9–12, the initial value of

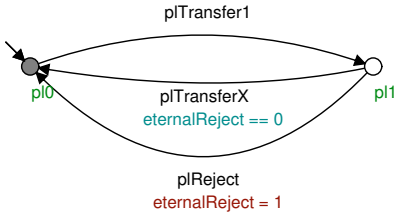

Fig. 7. EFSM model of the `transfer` to the player.

`sender` is set to the address of the operator. This might be changed by the non-deterministic assignment in the self-loop, but since `createGame` cannot be called by the player, only traces that start with the sender being the operator are of interest for the non-blocking verification.

Out from `S0` is also a transition to location `S1`, labeled with the initial event of each public function. From `S1` is then a transition back to `S0` labeled with the final events of the public functions. In this way, `sender` is assigned an address in location `S0`, representing either `player` or `operator`, and this address remains constant while any public function executes, as the EFSM is in its `S1` location.

Modeling transfer When a `transfer` occurs, the receiver can choose to either accept the transferred funds, or reject it. The EFSM of Fig 7, models this by having an initial event `plTransfer1` from the initial location `pl0` to `pl1`, which represents the transfer to the receiver, and two transitions back from `pl1` to `pl0`, where `plTransferX` represents an accepted transfer, while `plReject` represents a rejected transfer.

Rejection of `transfer` is modeled in the `playerWins` EFSM, see Fig 4, as a transition from the location where `transfer` is called, `S2`, directly back to the initial location, `S0`, bypassing the `playerWinsX` transition. Also, on that bypassing transition, the value of the `bet` variable is reset to what it was at the call of `playerWins`. This models that rejecting a `transfer` reverts any changes made by the call to `playerWins`.

Since the issue investigated in this paper concerns a malicious player that once rejecting a transfer will always reject any re-transfer, the `plTransferX` transition is guarded by `eternalReject == 0`, while the transition representing the rejected transfer sets the `eternalReject` variable to `1`. Whether this malicious player (the operator does not reject any transfer) can block the Casino so that the funds can never be retrieved, is what verification is to find out.

5. NON-BLOCKING VERIFICATION

The automatically converted model consists of 16 EFSMs and 16 variables[3]. Many of the EFSMs have only a single location, the biggest one, `decideBet`, has 5 locations. Most of the variables are 0–1 variables, the `state` variable has the largest domain of three symbolic values, `IDLE`, `GAME_AVAILABLE`, and `BET_PLACED`, just as its Solidity counterpart. When the EFSM model is "flattened" into ordinary finite-state machines (FSMs), which are then composed into a single FSM, this model has 127 states, 36 events, and 251 transitions.

[3] The model together with the code for the automatic conversion are available from https://github.com/martinfabian/Casino

Running SUPREMICA's *conflict check* on the automatically converted EFSM model generates a counter-example that shows the system to be blocking. This counter-example is not identical to the counter-example found for the manually crafted model (Mohajerani et al., 2022); it is longer, for instance, at 24 events long instead of 10, but it blocks in a similar way. When the player wins but decides to reject the transfer of the winnings, and from then on continues indefinitely to do so, the system ends up in a cyclic trace from which no marked state can be reached. Inspection of the code, Fig 1, maps the blocking to line 41. If `player.transfer(wager.bet*2)` fails, line 35 (`state = State.IDLE`) will not be executed. Though `decideBet` can be called again, when a malicious player refuses the `transfer` on each call, the contract will not progress to reach its `IDLE` state. Thus, the funds of the Casino can never be retrieved, meaning that the *liquidity* of the contract is compromised.

Mohajerani et al. (2022) show how the Casino code can be corrected to avoid this vulnerability, and prove that the corrected model is indeed non-blocking. The corrected code was also automatically converted to EFSMs by the conversion described in this paper, and this converted model was also shown to be non-blocking.

6. CONCLUSION

This paper investigates automatic conversion from Solidity code to a model of interacting EFSMs, where the different Solidity function calls and statements are modeled as transitions of the EFSMs. Non-blocking verification of the EFSM system shows that failure in transferring money to the player may result in the Casino contract to block as it will not be able to go back to its `IDLE` state. As a consequence, the investments made by the operator (through `addToPot`) are locked into the Casino forever. This type of blocking [4] has actually occurred on real contracts, with huge financial damage.

Since this work only concerns non-blocking, the controllability of events is irrelevant, but as the player versus the operator is a two-player game, modeling the player's actions as uncontrollable and the operator's actions as controllable (or the other way around) might be useful to reveal other aspects of the Casino code in particular, and of smart contracts in general. Work on this is ongoing.

REFERENCES

Akesson, K., Fabian, M., Flordal, H., and Malik, R. (2006). Supremica – an integrated environment for verification, synthesis and simulation of discrete event systems. In *2006 8th International Workshop on Discrete Event Systems*, 384–385. IEEE. doi:10.1109/WODES.2006.382401.

Bai, X., Cheng, Z., Duan, Z., and Hu, K. (2018). Formal modeling and verification of smart contracts. In *Proceedings of the 2018 7th International Conference on Software and Computer Applications*, ICSCA '18, 322–326. Association for Computing Machinery, New York, NY, USA. doi:10.1145/3185089.3185138.

Baier, C. and Katoen, J.P. (2008). *Principles of Model Checking*. MIT Press.

Bhargavan, K., Delignat-Lavaud, A., Fournet, C., Gollamudi, A., Gonthier, G., Kobeissi, N., Kulatova, N., Rastogi, A., Sibut-Pinote, T., Swamy, N., and Zanella-Béguelin, S. (2016). Formal verification of smart contracts: Short paper. In *Proceedings of the 2016 ACM Workshop on Programming Languages and Analysis for Security*, PLAS '16, 91–96. Association for Computing Machinery, New York, NY, USA. doi:10.1145/2993600.2993611.

Cavada, R., Cimatti, A., Dorigatti, M., Griggio, A., Mariotti, A., Micheli, A., Mover, S., Roveri, M., and Tonetta, S. (2014). The nuXmv symbolic model checker. In A. Biere and R. Bloem (eds.), *Computer Aided Verification*, 334–342. Springer International Publishing, Cham.

Fekih, R., Lahami, M., Jmaiel, M., Ali, A., and Genestier, P. (2022). Towards model checking approach for smart contract validation in the eip-1559 ethereum. 83–88. doi:10.1109/COMPSAC54236.2022.00020.

Godoy, J., Galeotti, J.P., Garbervetsky, D., and Uchitel, S. (2022). Predicate abstractions for smart contract validation. In *Proceedings of the 25th International Conference on Model Driven Engineering Languages and Systems*, MODELS '22, 289–299. Association for Computing Machinery, New York, NY, USA. doi:10.1145/3550355.3552462.

Hoare, C.A.R. (1985). *Communicating Sequential Processes*.

Holzmann, G. (1997). The model checker SPIN. *IEEE Transactions on Software Engineering*, 23(5), 279–295. doi:10.1109/32.588521.

Madl, G., Bathen, L., Flores, G., and Jadav, D. (2019). Formal verification of smart contracts using interface automata. In *2019 IEEE International Conference on Blockchain (Blockchain)*, 556–563. doi:10.1109/Blockchain.2019.00081.

Malik, R., Mohajerani, S., and Fabian, M. (2023). A survey on compositional algorithms for verification and synthesis in supervisory control. *Discrete Event Dynamic Systems*, 33(3), 279–340. doi:10.1007/s10626-023-00378-8.

Mavridou, A. and Laszka, A. (2018). Designing secure Ethereum smart contracts: A finite state machine based approach. In S. Meiklejohn and K. Sako (eds.), *Financial Cryptography and Data Security*, 523–540. Springer Berlin Heidelberg, Berlin, Heidelberg.

Mohajerani, S., Ahrendt, W., and Fabian, M. (2022). Modeling and security verification of state-based smart contracts. *IFAC-PapersOnLine*, 55(28), 356–362. doi:10.1016/j.ifacol.2022.10.366. 16th IFAC Workshop on Discrete Event Systems WODES 2022.

Ramadge, P.J.G. and Wonham, W.M. (1989). The control of discrete event systems. 77(1), 81–98.

Skoldstam, M., Akesson, K., and Fabian, M. (2007). Modeling of discrete event systems using finite automata with variables. In *2007 46th IEEE Conference on Decision and Control*, 3387–3392. doi:10.1109/CDC.2007.4434894.

Suvorov, D. and Ulyantsev, V. (2019). Smart contract design meets state machine synthesis: Case studies. URL https://arxiv.org/abs/1906.02906.

Wood, G. (2023). Ethereum: A secure decentralised generalised transaction ledger. *Ethereum Project Yellow Paper*. URL https://ethereum.github.io/yellowpaper/paper.pdf.

[4] https://coingeek.com/over-1-million-permanently-locked-in-defi-smart-contract/