On Detecting Bias in Smart Contracts Using Supervisor Synthesis

On Detecting Bias in Smart Contracts Using Supervisor Synthesis

Nishant Parekh, Wolfgang Ahrendt and Martin Fabian

***Abstract*— Smart contracts are computer programs executing in a blockchain environment, that can enforce agreements among mutually distrusting users without the involvement of any trusted third party. However, smart contracts may implement *bias* towards or against certain users, either intentionally or mistakenly by improper coding. Such bias may also manifest through multiple users colluding to put other users at a disadvantage. This study proposes a methodology to use supervisor synthesis to detect such biases in smart contracts. Given an extended finite-state machine model of a smart contract, by selecting which events are considered to be controllable, the perspective from which bias is considered can be decided, and by marking states, positive as well as negative bias can be captured. The existence of a supervisor may then reveal the existence of bias. The method is demonstrated on a multi-player game-based smart contract. The analysis shows that there does not exist positive bias towards any single player, that is, there is no guaranteed winning strategy for any single player. However, colluding players may enact negative bias against the other players, thus, there is negative bias against other players so that they may lose the game irrespective of their actions.**

I. INTRODUCTION

Smart contracts [14] are programs that execute within a blockchain environment, enforcing the terms defined by the contracts without any third party involvement. The trustless nature of the smart contract ecosystem enables interaction between multiple non-trusting parties, allowing them to participate in a given business (or game) scheme by placing their trust in the contract's implementation. However, absence of a third party assurance presents risks of its own, as interacting with poorly or maliciously designed smart contracts can harm some users. It could even be that a smart contract is intentionally designed to be advantageous to one side, while the opposing party remains exposed to the biases coded in the contract. Even if the implementation of smart contracts is openly available [13], it can be difficult to see potential biases embedded in the implementation.

Once deployed onto the blockchain, a smart contract implementation is immutable. Thus, it is important to ensure, prior to deploying the contract, that it does not implement unwanted behavior such as hidden biases. Equally important is the analysis of already deployed contracts, as it can warn potential users about the risk of being exposed to biased behavior disadvantageous to them, so they can refrain from interacting with the contract.

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

N. Parekh and M. Fabian are with the Dept. of Electrical Engineering, W. Ahrendt is with the Dept. of Computer Science and Engineering, Chalmers University of Technology, Gothenburg, Sweden

Previous work [8] outlines principles for automatically converting smart contracts from their source code to *Extended Finite State Machines* (EFSMs) [1, 12], and shows how the *Supervisory Control Theory* [10] can be applied to verify *non-blocking* behavior. Non-blocking is a progress property that when fulfilled guarantees that some states of particular interest, *marked* states, can always be reached, and in [9] this is used to verify whether a contract is resilient to a certain kind of denial-of-service attacks by a malicious user. If the contract is not resilient to such attacks, counter-examples can guide the developer in how to make the contract resilient.

Using the automatic conversion from smart contract source code to EFSMs of [8], this paper investigates how to detect whether a smart contract embeds biased behavior that is beneficial or detrimental to one or more users. For this, the distinction between *controllable* and *uncontrollable* events (which was irrelevant for the non-blocking verification of [9]) is introduced into the EFSM model of the smart contract. With careful selection of which events are controllable, and appropriate marking of states, computing a supervisor (which was not done in [9]) can reveal whether there exists a *strategy* that allows a particular user (or colluding users) to steer the contract towards beneficial outcomes regardless of what other users do. Revealing the presence of such a strategy reveals ways in which the contract's logic can be leveraged by a user (or users) to always reach their desired outcome. Similarly, the existence of a strategy can reveal if a (coalition of) user(s) can prevent a user (or users) from reaching their goals.

Current research on smart contract verification mainly focuses on security and safety using techniques such as model-checking and symbolic execution. Existing work around detecting bias and collusion is limited and primarily focuses on preventing collusion in auction and voting smart contracts. Related work by [4] presents a framework, FairCon, for auction and voting smart contracts, to verify truthfulness, efficiency, optimality and collusion-freeness by converting smart contracts to mechanism-design models and verifying fairness properties using SMT. Another related work, CReam [15] presents a smart contract based e-auction system that achieves collusion-freeness by building in incentive mechanism directly in the contract. Our work on the other hand, focuses on whether the contract's behavior enables enforcing strategies for some users, and provides a generalized framework for detecting bias in smart contracts. To the extent of our knowledge, little to no work addresses detecting biases embedded in smart contracts.

The use case of this paper, the NineGame smart contract, is a three-player variant of the Game of 21 [6, 7], a full

information multi-player game where players take turns to choose a number within a given range to increment a running total towards a target value, making it a suitable setting to investigate if a winning or losing strategy exists for the players. Such a strategy then defines at each state the actions that guarantee to reach the goal, irrespective of the actions of the other players. The results show that though there is no winning strategy for any single player, making the game appear fair, there do exist joint strategies in which two players can collude and force the third player to lose.

II. Background

This section presents a terse overview of the background necessary for the rest of the paper, the reader is referred to [8, 9, 10, 12, 14] for more comprehensive explanations.

A. Smart Contracts, Ethereum, and Solidity

The first, and still major, blockchain framework supporting smart contracts is Ethereum [14], with its built-in cryptocurrency Ether. In Ethereum not only users, but also contracts can receive, own, and send Ether. 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 (which can be 0). Miners are paid for their efforts with units of (Ether priced) *gas*, to be paid by the address that requested the transaction.

The most popular programming language for Ethereum smart contracts is Solidity[1], which follows largely an object-oriented paradigm. Example Solidity code is shown in Fig 1.

Built-in data types include unsigned integers (`uint`) of various bit-lengths, enums, and structs. All variables have well-defined initial values, which is `0` for integers, `false` for booleans, and `address(0)` for addresses. Users and contract instances all have unique addresses of type `address`. Each `address` owns and can send Ether, and sending Ether to a contract is the same thing as calling a `public` function of the contract. The current caller, and the amount sent with the call of a (payable) function, are available via `msg.sender` and `msg.value`, respectively. The statement `require(`b`)` checks the boolean expression b, and reverts if b is false, undoing any effects that took place since the beginning of the transaction. `require` statements are often used early in a function, before any effects can take place, to ensure that a call has no effect unless the caller is a specific address, unless the parameters satisfy certain conditions, unless the amount of Ether sent with the call is appropriate, to name a few use cases.

B. Extended Finite State Machines

Extended Finite State Machines [1, 12] are finite-state machines extended with bounded, discrete variables and associating with the transitions *guards* and *actions* over those variables. A guard is a condition that when satisfied enables the transition, and when the transition occurs the variables associated with the actions are updated. Define an EFSM as $E = \langle \Sigma_E, L_E, L_E^0, L_E^\omega, \delta_E \rangle$, with Σ_E the finite set of event labels, the *alphabet*; L_E the finite set of *locations*, with $L_E^0 \subseteq L_E$ the set of initial, and $L_E^\omega \subseteq L_E$ the set of marked locations; δ_E the *transition relation*, where a transition $\langle l_m, \sigma, g, a, l_n \rangle$ denotes that the EFSM can transit from location $l_m \in L_E$ to location $l_n \in L_E$ when the guard g evaluates to *true* and the event $\sigma \in \Sigma_E$ occurs, and then the action a is performed.

The *state* of an EFSM is given by the location together with the values of the variables. Just as locations can be marked, so can variable values be marked. Thus, a *marked state* is one where the location and all variable values are marked. The (prefix-closed) *language* of an EFSM E, $\mathcal{L}(E) \subseteq \Sigma_E^*$, is the set of all possible finite sequences of events. The *marked language*, $\mathcal{L}_m(E) \subseteq \mathcal{L}(E)$, is the set of all finite sequences of events reaching marked states. For a language $\mathcal{L}$, its *prefix-closure*, $\overline{\mathcal{L}}$, is the set of all prefixes of all traces of $\mathcal{L}$, that is, $\overline{\mathcal{L}} = \{s \in \Sigma^* \mid \exists t \in \Sigma^*, st \in \mathcal{L}\}$.

To allow non-deterministic assignments to variables, guards can refer to *post-transition* values, denoted by a prime ($'$) symbol on the variable name. For example, the guard expression in Fig. 4 ensures that `num` has either of the values 1, 2 or 3 *after* the transition.

EFSMs interact with each other via *synchronous composition* [2] (denoted by $||$) of *shared events*, that is, events common between the EFSMs. In the composition, a shared event is enabled only if it is enabled in all EFSMs containing that event in their alphabets. For instance, the event `move01` in Fig 2 synchronizes with the same labeled event of the EFSM in Fig 3 that models the `require` clause on line 31 of Fig 1, so that the transition labeled by `move01` in Fig 2 cannot occur unless the guard `state_total + num'<=9` of Fig 3 evaluates to true.

C. Supervisory Control Theory

A *supervisor* is a control function whose purpose is to regulate the behavior of a *plant* ensuring that the systems jointly operate within a given *specification*. The Supervisory Control Theory (SCT) [10] provides a formal framework for *synthesis* of a supervisor S such that, given a plant G and a specification K, the supervisor dynamically restricts the behavior of G, so as to always fulfill K.

In the SCT framework, the set of events is partitioned into the sets of *controllable* and *uncontrollable* events, Σ_c and Σ_u, respectively. The supervisor must be such that it exerts its control while respecting the controllability of the events; only controllable events can be disabled, while the supervisor must always enable the uncontrollable events. In addition, the supervisor is required to be *non-blocking*, meaning that each sequence of events possible in the controlled system $G||S$ can always be extended to reach a marked state.

III. On Bias and Strategies

A smart contract's behavior is typically triggered by users initiating *external actions*, as the result of calling, through the blockchain framework, the contract's public functions with certain parameter values. Upon such a call, the contract executes a sequence of *internal actions*, after which eventually a new external action may be triggered by the users.

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

A smart contract exhibits *bias* when a user[2] can steer, by carefully choosing their external actions, the contract to the benefit of themselves or the detriment of others. The sequences of internal and external actions resulting from such a steering of the contract is called a *strategy*.

Bias comes in two forms, *positive bias* towards a user, and *negative bias* against a user. Positive bias towards a user exists when that user can by choosing their external actions steer the contract towards states that are beneficial to themselves, irrespective of what other users do. Negative bias against a user exists when other users can by choosing their external actions steer the contract towards states that are detrimental to the user, irrespective of what that user does. In both cases the bias is revealed by the existence of a strategy to realize the bias.

Note that the existence of positive bias towards a user, does not necessarily mean that there exists negative bias against other users, or the other way around. This entirely depends on which states of the contract that are considered beneficial (or detrimental) to the respective users. In some contracts, users may have complementary goals so that positive bias exists towards all users simultaneously. In other contracts, negative bias may exist against all users.

For the EFSM model G of a smart contract, let $\mathcal{L}(G)$ and $\mathcal{L}_m(G)$ denote the closed and the marked languages, respectively. Partition Σ_G into the sets of user U's external actions, Σ_U, the other user's external actions, Σ_V, and the contract's internal actions, Σ_I. Let Q denote the full state space of G.

Definition 1: Let $Q_U \subseteq Q$ be the set of states beneficial for user U, and let $\mathcal{L}_m(G)$ denote the marked, with respect to Q_U, language of G. Then there exists *positive bias towards* U if there exists a strategy $K_U \subseteq \mathcal{L}_m(G)$ such that:

$$\overline{K_U}\,(\Sigma_V \cup \Sigma_I) \cap \mathcal{L}(G) \subseteq \overline{K_U}. \tag{1}$$

(1) expresses that irrespective of the other user's external actions or the contract's internal actions ($\Sigma_V \cup \Sigma_I$), user U can always choose external actions to remain within K_U.

Definition 2: Let $Q_V \subseteq Q \setminus Q_U$ be a set of states detrimental to user U, and let $\mathcal{L}_m(G)$ denote the marked, with respect to Q_V, language of G. Then there exists *negative bias against* U if there exists a strategy $L_U \subseteq \mathcal{L}_m(G)$ such that:

$$\overline{L_U}\,(\Sigma_U \cup \Sigma_I) \cap \mathcal{L}(G) \subseteq \overline{L_U} \tag{2}$$

(2) expresses that irrespective of user U's external actions or the contracts internal actions ($\Sigma_U \cup \Sigma_I$), the other users can always choose external actions to stay within L_U. Note that Q_V and Q_U are not necessarily the full partition of Q, states may exist in Q that belong to neither Q_V nor Q_U.

In essence, positive bias means that whatever the others do, U can steer the contract to remain within the strategy, and the strategy guarantees to reach states beneficial for U. And negative bias means that whatever U does, U cannot steer the contract to go outside the strategy, and the strategy guarantees to reach states detrimental to U.

Both (1) and (2) express variants of *controllability* [10]. What differs is the set of events considered uncontrollable. In both cases, the contract's internal events are uncontrollable, but for positive bias towards U, Def 1, the other user's events are considered uncontrollable, while for negative bias against U, Def 2, U's own events are considered uncontrollable.

As defined, a strategy allows a user to steer the contract by selecting external actions so as to reach some marked state. For this, the strategy has to be controllable wrt G and the uncontrollable events, so that uncontrollable events cannot divert from the strategy, and it has to be non-blocking so as to always be able to reach some marked state. This is much like a supervisor. However, non-blocking only guarantees the *possibility* for some marked state to be reached, it does not guarantee that any marked state will ever actually be reached; a supervisor can include cycles of non-marked states, from which no controllable user action can escape. Thus, to be a strategy, a supervisor must be free from such cycles of non-marked states. This means that if no supervisor exists, then no strategy and hence no bias exists. However, that a supervisor does exist is no guarantee that a strategy exists. Finding cycles of non-marked states can be done by Johnson's algorithm [3] in time complexity $\mathcal{O}((|Q_S| + |\delta_S|)(c+1))$, with $|Q_S|$ the number of states, $|\delta_S|$ the number of transitions, and c the number of cycles of the strategy.

A special case where a supervisor *is* a strategy, is when the supervisor has a finite language, which is the case in the following examples.

IV. EXAMPLE

The Solidity code implementing the 3-player NineGame contract is shown in Fig 1. Players take turns choosing a number (1, 2 or 3) to increase a running total starting at 0. The player to first reach 9 wins the game and receives the funds held in the contract. The NineGame is a variant of the Game of 21 [6, 7], where instead of two, the game has three players. Addition of the third player allows us to analyze scenarios where users collude. The games are structurally similar, but lowering the target total from 21 to 9 significantly reduces the state space and allows to more comprehensibly illustrate the results.

The game begins with a user launching the contract and placing an initial bet set in the `constructor` at line 15. The contract creator's address is stored in the variable `player1`. Other user can participate in the game, if the game is available, by calling the function `join`, line 17, and placing an identical bet, line 19, following which the second and third user's addresses are stored as `player2` and `player3` respectively, lines 21 and 23. The variable `state.whoseTurn` at line 9 keeps track of whose turn it is next in the game, and is set at line 24 once all three players have joined. The game play takes place in the `move` function, where players take turns calling `move`, passing an integer indicating their choice. The value of the parameter is either 1, 2 or 3, ensured

[2]For brevity, in the following, the term "user" should be interpreted as "user or users", since bias can exist towards or against multiple users, especially so in cases where a set of users can collude against other users.

```
1  contract NineGame {
2      address payable public player1;
3      address payable public player2;
4      address payable public player3;
5      uint256 public betAmount;
6      bool public gameOver;
7      struct GameState {
8          uint8 total;
9          address whoseTurn;
10     }
11     GameState public state;
12
13 constructor() public payable {
14     player1 = payable(msg.sender);
15     betAmount = msg.value;
16 }
17 function join() public payable {
18     require(!gameOver);
19     require(msg.value == betAmount);
20     if (player2 == address(0) {
21         player2 = payable(msg.sender);
22     } else if (player3 == address(0) {
23        player3 = payable(msg.sender);
24        state.whoseTurn = player1;
25     }
26 }
27 function move(uint8 num) public {
28     require(!gameOver);
29     require(msg.sender == state.whoseTurn);
30     require(num >= 1 && num <= 3);
31     require(state.total + num <= 9);
32     state.total += num;
33     if (msg.sender == player1) {
34         state.whoseTurn = player2;
35     } else if (msg.sender == player2) {
36         state.whoseTurn = player3;
37     } else {
38         state.whoseTurn = player1;
39     }
40     if (state.total == 9) {
41         gameOver = true;
42         payable(msg.sender).transfer(
43             address(this).balance);
44     }
45 }
```

Fig. 1. Solidity code for the NineGame contract (some details are omitted).

by the **require** at line 30, increasing a global running total `state.total`, line 32, initialized to 0 at line 11.

The **require** statements in `move`, lines 28–31, ensure that the game is running, that it is the proper player's turn, that the chosen `num` is between 1 and 3, and that the running total, `state.total`, does not exceed 9. After each move, `state.whoseTurn` is assigned the next player in the three player cycle. If `state.total` reaches nine, line 40, `gameOver` is set to **true** and the winnings are transferred to the player whose turn achieved the target total.

V. METHODOLOGY

Smart contracts are automatically converted from their source code to EFSMs by the framework detailed in [8]. The generated EFSM models jointly represent all possible behavior of the code, above denoted by G. For illustration, a reduced version of the EFSM model [3] generated for function `move` (lines 27–45 of Fig. 1) is shown in Fig 2. While the full model is too large to be presented here, the reduced model preserves enough detail to follow the execution behavior and to investigate bias.

A. Function Invocation

Each EFSM modeling a function has for each user a *function invocation* transition from its initial location labeled

[3]The models together with the code for the automatic conversion are available from `https://github.com/nishantparekh01/Solidity_to_EFSM/`

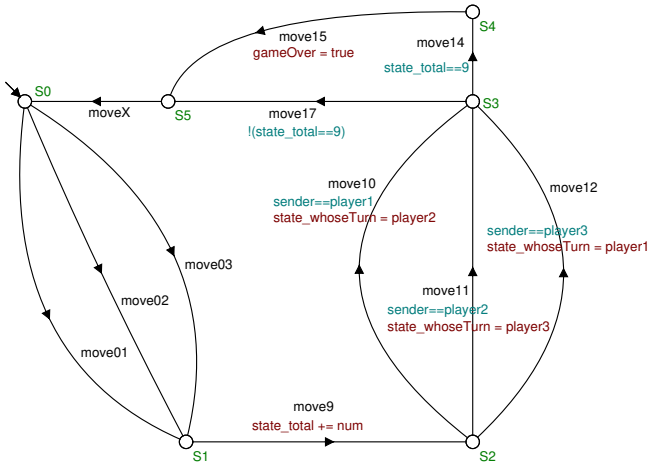

Fig. 2. EFSM model of the `move` function.

by an event representing the external action by the respective user, see Fig. 2. These events synchronize with EFSM models that implement the **require** statements that typically appear at the beginning of a function, see for example Fig. 3. All EFSMs that model the various **require** statements at the beginning of `move` (lines 28–31) synchronize with these events. Since the different function invocation events need to be regarded as controllable or uncontrollable depending on the perspective from which bias is considered, each function invocation event needs to be tied to a specific user. Thus, there is a self-looped EFSM model for each user such as the one in Fig. 4, which ties `move01` to `player1`.

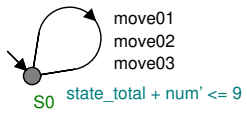

Fig. 3. EFSM model of the require statement at line 31. The three events represent invocation of `move` by `player1`, `player2` and `player3`, respectively.

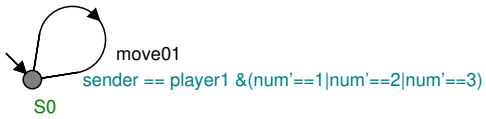

Fig. 4. EFSM model tying `move01` to `player1`, and modeling the non-deterministic assignment of the `num` variable by `player1`.

In addition, Fig. 4 models the non-deterministic assignment of the parameter `num` passed with the call of `move`. When `move01` occurs, the EFSM of Fig. 4 guarantees that `player1` invoked `move`, and that the post-transition value of `num` is non-deterministically assigned 1, 2, or 3. The `move01` event synchronizes with the same labeled events in the EFSMs of figures 2 and 3. The guard of the EFSM in Fig. 3 then further guarantees that only values of `num` that do not overreach the target nine can be chosen. This is a generic way to model function invocation with parameters.

B. Configuration

To analyze positive and negative bias, certain events are set to be controllable, see (1) and (2), and certain states are marked. Defining controllable events and marking states has to be done manually, as it cannot be extracted from the code. Common to analyzing both positive and negative bias is that all events representing the internal actions Σ_I of the contract are considered uncontrollable, capturing that no user can interfere with the contract's internal execution. Once invoked, a smart contract executes autonomously in a blockchain execution environment, and neither the caller nor any other user can interrupt its internal execution.

1) Configuration for positive bias: To analyze positive bias for a user, the events representing external actions of the user are considered controllable, and states that express beneficial outcomes for the user are marked. Events representing external actions by the other users are considered uncontrollable. This way, when a supervisor exists, it represents a strategy that the user under inspection can use to achieve the desired outcome; as controllable events in the supervisor correspond to external actions by the user, the user can always choose actions to reach some beneficial state.

In the NineGame, when analyzing from `player1`'s perspective, `player1`'s events are considered controllable while the other player's events are uncontrollable. In addition, the variable value representing the end of game, `state_total==9`, is marked, owing that the player reaching the target 9 wins the game, as is the variable value of `whoseTurn` representing that the *next* player in sequence (`player2` in this case) has the turn. Thus, when analyzing from `player1`'s perspective, `state_whoseTurn == x0002` and `state_total==9` are both marked, representing that when the game is over, `player1` was the player that counted up to win the game.

2) Configuration for negative bias: To analyze negative bias for a user U, the events representing external actions of U are made *uncontrollable*, and states that express detrimental outcomes for U are marked. Events representing external actions by the other users are considered controllable. In this way, when a supervisor exists, it reveals strategies that prevent U from reaching beneficial states regardless of U's actions; as controllable events in the supervisor correspond to external actions by the other users, whatever external actions U takes, some other user can always choose actions to reach states detrimental to U.

In the NineGame, when analyzing negative bias from `player1`'s perspective, `player1`'s events are considered uncontrollable, while the other player's events are controllable. As above, the variable value representing the end of game, `state_total==9`, is marked, but now the the variable values of `whoseTurn` representing that the *current* and *previous* player in sequence (`player1` and `player2` in this case) has the turn, is marked. Thus, when reaching a marked state, the game is over and it is either `player1`'s turn or `player2`'s turn, representing that either `player3` or `player2` won the game. Concretely, this is done by marking the values `state_whoseTurn != x0002`, indicating that it was *not* `player1`'s turn at which `state_total` reached 9.

VI. RESULTS AND ANALYSIS

This section presents the results of analyzing both positive and negative bias of the NineGame smart contract. As the example is rather small, the monolithic synthesis of SUPREMICA [11] is used, which synchronizes all model components and generates a single supervisor. For larger contracts, more efficient algorithms, like abstraction-based compositional synthesis [5], would have to be used. In any case, if no supervisor exists, this means that there is no strategy towards or against the player(s) from whose perspective bias is analyzed.

A. Analyzing positive bias

Analyzing positive bias in the three-player NineGame involves marking for each player its respective winning state, and considering that player's external action events as controllable, to then synthesize a supervisor. In all three cases, no supervisor exists, indicating that there exists no positive bias, and hence no guaranteed winning strategy, for any individual player.

In contrast, for the two-player version of the NineGame, from the perspective of `player1` who takes the first turn, a supervisor can be synthesized, meaning that there does exist positive bias towards `player1`. The supervisor disables `player1` from initially choosing anything but 1. After that, no matter what `player2` does, `player1` can steer the game to eventually reach 9. Analyzing positive bias towards `player2` results in no supervisor, so no positive bias towards `player2` exists. This is not surprising, due to the fact that there exists positive bias towards `player1`, and thus there exists a guaranteed winning strategy for `player1`, and this being a two-player game, if `player1` wins then `player2` loses.

B. Analyzing negative bias

Analyzing negative bias for the three-player NineGame involves marking for each player the winning states of the other players, and considering the other player's external actions controllable. Synthesizing a supervisor now gives a result, showing that negative bias exists for each player; that is, for each player the other players can force that player to lose no matter what the player does. Analysis of the synthesized supervisor further reveals that the strategy relies on coordinated behavior by the remaining two players. In other words, if two players collude and work jointly, they can enforce a losing outcome for the third player.

An abstraction of the supervisor synthesized when analyzing negative bias against `player1` is shown in Fig 5. This supervisor describes how `player2` and `player3` can jointly play the game so that one of them wins (the marked states `P3:S9` and `P1:S9`), hence `player1` is guaranteed to lose. Here, an event `pi:v` represents that player i chooses value v. As shown in Fig 5, `player1` can make any move during its turn. For instance at the initial state `P1:S0`, all events `!p1:1`, `!p1:2`, and `!p1:3` are enabled. On the other hand,

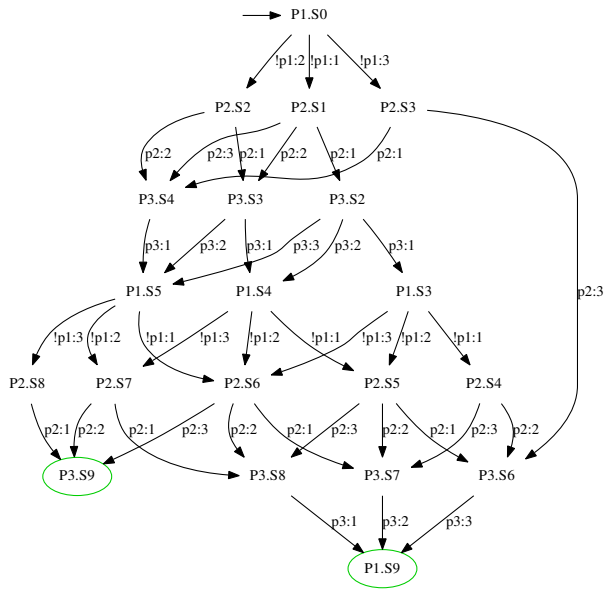

Fig. 5. Abstract form supervisor revealing negative bias against `player1`. Events prefixed with an exclamation mark ! are uncontrollable.

at location `P3:S4` only `p3:1` is enabled indicating that `player3` can continue to steer the game to `player1`'s loss only by selecting 1.

It can also be seen from Fig 5 that `player3` can steer the game to `P1.S5`, from where `player2` is guaranteed to win. This could happen if `player2` and `player3` are the same user (human or contract).

VII. CONCLUSIONS AND FUTURE WORK

This paper shows how supervisor synthesis can be used to detect bias that might be embedded in, seemingly fair, smart contracts. Though the source code for deployed smart contracts is publicly available on platforms like Etherscan [13], detecting bias is hard even for simple contracts, like the NineGame. The analysis presented in this work can reveal such crucial information for deciding which contracts a user should (or should not) interact with. This is a subtle yet significant problem that traditional verification methods typically do not deal with.

The example discussed in this paper is a game in the narrow sense of the word, where one player winning means the others lose, so there are competing goals. In that case, positive bias towards one implies negative bias against the others. In general this is not the case, though, as user goals can coincide. For instance, in an auction contract, one user wants to sell and one user wants to buy, then positive bias for the seller is also positive bias for the buyer. Thus, the presented approach is relevant for all smart contracts where different users have adversarial or coinciding interests.

Currently, identifying and generating configurations associated to beneficial or detrimental outcomes requires an understanding of the underlying EFSM models. A future direction of this research is to investigate how this can be generated automatically from high-level goals, making it easier for users to specify what beneficial and/or detrimental behavior means for a specific party.

REFERENCES

[1] Kwang Ting Cheng and A. S. Krishnakumar. "Automatic functional test generation using the extended finite state machine model". In: *Proceedings of the 30th ACM/IEEE Design Automation Conference*. 1993, pp. 86–91. DOI: 10.1145/157485.164585.

[2] C. A. R. Hoare. "Communicating sequential processes". In: *Commun. ACM* 21.8 (Aug. 1978), pp. 666–677. ISSN: 0001-0782. DOI: 10.1145/359576.359585.

[3] Donald B. Johnson. "Finding All the Elementary Circuits of a Directed Graph". In: *SIAM Journal on Computing* 4.1 (1975), pp. 77–84. DOI: 10.1137/0204007.

[4] Shang-Wei Lin Liu Ye Li Yi and Zhao Rong. "Towards automated verification of smart contract fairness". In: *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 2020, pp. 666–677. DOI: 10.1109/CSCI46756.2018.00112.

[5] Robi Malik, Sahar Mohajerani, and Martin Fabian. "A survey on compositional algorithms for verification and synthesis in supervisory control". In: *Discrete Event Dynamic Systems* 33.3 (Sept. 2023), pp. 279–340. ISSN: 1573-7594. DOI: 10.1007/s10626-023-00378-8.

[6] David J. Butler Martin Dufwenberg Ramya Sundaram. "Epiphany in the Game of 21". In: *Journal of Economic Behavior & Organization* (2010), pp. 132–143. ISSN: 0167-2681. DOI: 10.1016/j.jebo.2010.03.025.

[7] Steve Marx. *Two-Player Games in Ethereum*. https://programtheblockchain.com/posts/2018/05/04/two-player-games-in-ethereum/. [Online; accessed 27-March-2025]. 2018.

[8] Nishant Parekh, Wolfgang Ahrendt, and Martin Fabian. "Automatic Conversion of Smart Contracts for Non-Blocking Verification". In: *IFAC-PapersOnLine* 58.1 (2024). 17th IFAC Workshop on Discrete Event Systems WODES 2024, pp. 282–287. ISSN: 2405-8963. DOI: 10.1016/j.ifacol.2024.07.048.

[9] Nishant Parekh, Wolfgang Ahrendt, and Martin Fabian. "Smart contract denial-of-service analysis using non-blocking verification". In: *Discrete Event Dynamic Systems* (Dec. 2025). ISSN: 1573-7594. DOI: 10.1007/s10626-025-00418-5.

[10] Peter J. G. Ramadge and W. Murray Wonham. "The Control of Discrete Event Systems". In: 77.1 (Jan. 1989), pp. 81–98. DOI: 10.1109/5.21072.

[11] Supremica – Public Repo. https://tinyurl.com/Supremica. [Online; Accessed: 2025-03-31].

[12] Markus Sköldstam, Knut Åkesson, and Martin Fabian. "Modeling of discrete event systems using finite automata with variables". In: *2007 46th IEEE Conference on Decision and Control*. 2007, pp. 3387–3392. DOI: 10.1109/CDC.2007.4434894.

[13] *The Ethereum Blockchain Explorer*. https://etherscan.io/. [Online; accessed 27-March-2025].

[14] Gavin Wood. "Ethereum: A secure decentralised generalised transaction ledger". In: *Ethereum Project Yellow Paper* (2023). URL: https://ethereum.github.io/yellowpaper/paper.pdf.

[15] Shuangke Wu, Yanjiao Chen, Qian Wang, Minghui Li, Cong Wang, and Xiangyang Luo. "CReam: A Smart Contract Enabled Collusion-Resistant e-Auction". In: *IEEE Transactions on Information Forensics and Security* 14.7 (2019), pp. 1687–1701. DOI: 10.1109/TIFS.2018.2883275.