# Smart contract denial-of-service analysis using non-blocking verification

Discrete Event Dynamic Systems (2025) 35:355–387
https://doi.org/10.1007/s10626-025-00418-5

# Smart contract denial-of-service analysis using non-blocking verification

Nishant Parekh[1] · Wolfgang Ahrendt[2] · Martin Fabian[1]

Received: 10 December 2024 / Accepted: 4 July 2025 / Published online: 20 December 2025

**Abstract**
Smart contracts are programs that can enforce agreements between mutually distrusting parties, eliminating the need for intermediaries, such as lawyers or banks. As smart contracts are stored on a blockchain ledger, they are immutable after deployment, which makes assessment of their correctness before deployment vital. Many vulnerabilities of smart contracts are known, and having means to assess whether a contract is prone to one or more of these is crucial. A specific such vulnerability is *denial-of-service* (DoS), which can make a smart contract unresponsive so that users (including other smart contracts) cannot interact with it as intended. This can lead (and has led) lead to financial losses, or disrupt critical services that rely on the contract. Extended finite state machines (EFSM) are a modelling formalism for discrete-event systems, which provides a systematic approach to scrutinize smart contract functionalities. With careful modeling, non-blocking verification can be used to determine whether a contract is vulnerable to DoS attacks. This paper describes a methodology to automatically convert from the abstract syntax tree of a smart contract to an EFSM model, and then shows how non-blocking verification can indeed assess whether DoS attacks can cause harm. Two specific use cases are treated, a contract implementing a (simple) on-line casino, and an auction contract. Verification of the EFSM models reveals both contracts to be prone to DoS attacks, and counterexamples hint at how the contracts can be made non-blocking, meaning that they can be corrected not to be vulnerable. Automatic conversion and non-blocking verification of the corrected contracts indeed show that they are no longer prone to DoS attacks.

**Keywords** Extended finite state machines · Smart contracts · Verification · Non-blocking

## 1 Introduction

*Smart contracts* are micro-services executing within a blockchain ecosystem. Their main purpose is to enforce agreements among mutually distrusting parties without the need for intermediaries. In addition to traditional communication, smart contracts have the additional ability to receive and send assets, typically in the form of crypto-currency. As smart

Extended author information available on the last page of the article contracts become increasingly capable of handling more complex interactions and transactions, the potential for, and the effects of, errors and vulnerabilities increase. Even if the underlying blockchain protocols cannot feasibly be compromised, a smart contract can itself allow behaviour, unintended by the programmer, that may result in, or be exploited to, the disadvantage of some users. For instance, an unintended programming error in the DeFi contract (CoinGeek 2020) has permanently made over $1 million inaccessible to its owners.

Multiple vulnerabilities of smart contracts are known (SWC Registry 2020; Richter Vidal et al. 2024), and exploits of these vulnerabilities, called *attacks*, have caused loss of huge funds (Atzei et al. 2017; Bartoletti et al. 2024). Moreover, smart contracts are immutable once deployed on the blockchain. Thus, it is crucial to, before deployment, assess their correctness and resilience to vulnerabilities.

That funds become inaccessible to users who have legitimate interest to access them (according to what was intended by the programmer and understood by the benevolent users), is referred to as compromised *liquidity* of the contract. Liquidity problems are a (prominent) special case of a more general issue smart contracts can have: that a certain state that was intended to be *reachable* can become unreachable due to a malicious user exploiting a *reachability vulnerability*. Often, the ability to reach a certain state can be seen as a *service* offered by the smart contract. Accordingly, exploiting a reachability vulnerability is in effect a *denial-of-service (DoS)* attack. This specific vulnerability, also the focus of our work, is common in smart contracts and is categorized as “DoS with Failed Call” (SWC-113), in the Smart Contract Weakness Classification registry, (SWC Registry 2020).

The aim of the work presented here is precisely the detection of reachability vulnerabilities, i.e., it is analysed whether some party using the contract can provoke a DoS. Every effort should be made to avoid unintended non-reachability (of actions or states), since it leads to some parties not being able to execute their rights, and become subject to—potentially substantial—damage that cannot be repaired.

*Formal methods* are a set of mathematical techniques used for design and verification of software and hardware systems that allows rigorous analysis and can guarantee correctness in relation to given requirement specifications. For finite state transition systems, *model checking* (Baier and Katoen 2008), one of several formal methods, can verify correctness by exhaustively evaluating the state space against given specifications.

A specific type of finite state transition systems are *Extended Finite State Machines* (EFSMs) (Cheng and Krishnakumar 1996; Chen and Lin 2000; Skoldstam et al. 2007). These are similar to ordinary finite-state machines, but add bounded discrete variables together with guards and actions defined over these variables. Mohajerani et al. (2022) show how state based smart contracts can be modelled as EFSMs by abstracting the contract’s high-level behaviour, ignoring intermediate execution details. This then allows to use model checking techniques to verify correctness.

In terms of a finite state model of a smart contract, compromised liquidity means that all states where the funds can be accessed are unreachable, whatever actions are taken. This is exactly the notion of *blocking*; from some state reachable from the initial state, no marked state where the funds can be accessed is reachable. So, given the source code of a smart contract, states where the funds are guaranteed to be accessible are specified as marked. Then, an attacker model is introduced, whereafter the model is verified to be non-blocking or not. If the model with the specification is non-blocking, that particular malicious behavior cannot compromise the liquidity of the contract. On the other hand, if the model should turn out to be blocking, then the malicious behaviour can compromise the contract's liquidity, and measures should be taken against that. Typically, if verification shows that the model is blocking, a counterexample is given that can help indicate how to correct the contract.

Mohajerani et al. (2022) show how state based smart contracts can be modelled as EFSMs, and how methods from *supervisory control theory* (Ramadge and Wonham 1989) can be applied to verify non-blocking behaviour. Mohajerani et al. (2022) specifcally targeted an online Casino contract, and the model was shown to be blocking with a malicious player, which indicated a vulnerability of the contract to DoS attacks. The work thus showed that non-blocking verification can be useful to find smart contract vulnerabilities. However, Mohajerani et al. (2022) modeled the smart contract manually, a tedious and error prone task practically impossible for larger contracts. Manual modeling also leads to a more abstracted model, less true to the actual behavior of the executing contract, which might lead to discovering issues not present in the actual code (false positives), or missing issues that are present in the code (false negatives).

This paper, which is a significant extension of Parekh et al. (2024), presents an approach to *automatically* convert, from their source code, smart contractsto EFSMs. Modeling smart contracts as EFSMs offers a more compact and descriptive form of modeling over FSMs. The approach converts variables, modifiers, functions and generic framework behaviour, into a set of interacting EFSMs, the composition of which models the overall behaviour of the smart contract. The automatic conversion allows to handle large contracts, and results in more detailed models, closer to the actual behaviour of the code. The Casino smart contract treated by Mohajerani et al. (2022) is used as an example also here, but differently from their work, first the behaviour of the code with parties that do not specifically exhibit malicious behaviour is modelled; this is considered the *plant*. Then a *specification* that expresses malicious behaviour is added for the verification. To show the generality of the presented approach, an Auction contract is also automatically converted. Again, first the plant is converted, to which then a specification is added for verification. Formal verification of the EFSM models of the two contracts reveals both to be blocking, which shows that they are vulnerable to DoS attacks. The counterexamples generated by the verification indicate how the vulnerabilities can be amended, and automatically converting and verifying the updated contracts show them to be non-blocking and hence not vulnerable to the DoS attacks.

The paper is structured as follows: Section 2 presents an overview of related work. Section 3 provides a brief background on the smart contract implementation language Solidity, EFSMs and Supremica. Section 4 describes the Casino and the Auction contracts, and Section 5 details the automatic conversion from the source code, via the abstract syntax tree to EFSMs. Section 6 presents the non-blocking verification of the EFSM models and the corrected code, while Section 7 concludes the paper.

## 2 Related work

Given their increasing importance, and the amount of financial damage that can be caused, it is not surprising that verification of smart contracts has lately received a lot of attention. As already mentioned, a number of vulnerabilities are known and have been categorized (Atzei et al. 2017; SWC Registry 2020; Richter Vidal et al. 2024).

Related work on 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 above-mentioned 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. Modelling of smart contract as PROMELA models and verifying them using SPIN (Holzmann 1997) is presented in Bai et al. (2018). 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.

Another work, presented by Bartoletti et al. (2024), propose a tool, Solvent, which translates a smart contract and its set of user-defined liquidity properties into SMT constraints (Barrett et al. 2009) that can be analyzed by SMT solvers such as Z3 (de Moura and Bjørner 2008) or cvc5 (Barbosa et al. 2022).

The work of Bartoletti et al. (2024) and our work are related in so far as some of the DoS attacks we can detect result in a loss of liquidity. For instance, the Casino example (Sect. 4.1) relates to liquidity problems, whereas the Auction example (Sect. 4.2) does not. But even in the cases where our work addresses liquidity, the work of Bartoletti et al. (2024) and our work analyse very different root causes of liquidity. They study the effects of unsolicited, 'silent' Ether transfers through contract destruction, which is not covered by our analysis. Conversely, we study the effects of reverting ('failing') transfers, which is not covered by their analysis. On that point, they comment "Since considering each transfer as potentially failing would make most contracts illiquid, a possible approach would be to allow queries to specify which transfers or addresses to be considered trusted". We have a different take on this issue. Having to trust payment-receiving addresses creates a level of reliance that contradicts the fundamentally trustless ecosystem of smart contracts. As soon as payment is involved, sender and receiver may have adversarial interests. Indeed, transfers being potentially unsuccessful presents a risk of *denial-of-service* (DoS) attacks, which are well documented among other vulnerabilities in the Smart Contract Weakness Classification (SWC Registry 2020) as SWC-113, *DoS with Failed Call*. This weakness is also recognized as *1.3.2 Improper Exception Handling of External Calls* by Richter Vidal et al. (2024) who present a hierarchical taxonomy of smart contract vulnerabilities. In an empirical evaluation of smart contract implementations, Parizi et al. (2018) note that all implementations, in three different smart contract programming languages, of their Contract 3, "King of the currency" (Konstantopoulos 2018) were vulnerable to DoS with unexpected revert attacks.

Finally, we do not concur with the view, put forward by Bartoletti et al. (2024), that "transfer [...] failing would make most contracts illiquid". Contracts can be programmed in such a way that they are resilient against failing transfers. This article describes a method showing whether or not a contract implementation achieves resilience against failing transfers.

## 3 Background

This section gives the background necessary to fully appreciate the rest of the paper. First a brief overview of Ethereum Smart Contracts and the programming language Solidity is given. This is followed by a description of the EFSM modeling formalism used for verification.

### 3.1 Smart contracts: Ethereum and Solidity

The first, and still major, blockchain framework for smart contracts is Ethereum (Wood 2023), with its built-in cryptocurrency Ether. Ethereum smart contracts can be thought of as objects, with fields[1] making up the state space of the contract, and code offering functionalities to callers of the contract. In order to get a first impression, we suggest to glance at Fig. 2, showing an example contract with fields `state`, `hashedNumber`, `operator`, `player`, `wager`, and `pot`. The **`public`** functions, here `createGame`, `placeBet`, `decideBet`, `addToPot` and `removeFromPot`, offer functionality to callers of the contract. We will return to the details of the contract language, as well as this concrete contract, later on.

In Ethereum, every user and every contract has a unique address. Every address (user or contract) has an Ether balance, and can receive and send Ether in any direction (user to user, user to contract, contract to user, contract to contract). In contrast to user addresses, contract addresses have the additional feature of code being assigned to them, which is executed once the contract is called (by a user or by another contract). The executable code is stored on the blockchain in the form of EVM (Ethereum Virtual Machine) bytecode.

Contract execution in Ethereum features a transaction mechanism. Every call starts a transaction which is either completed successfully, or reverted if not successful. In the latter case, all effects so far, like Ether transfer or changes to fields, are undone. An unsuccessful transaction may revert for various reasons, like for instance running out of gas (see below), sending of unbacked funds, a failing runtime assertion, a **`revert`** statement in the code, a reverting call to another contract, or a reverting transfer, see below.

Ethereum miners look for transaction requests on the network. A transaction request contains the address of the contract to be called, the call data, and the amount of Ether to be sent. Miners execute the transaction requests locally on an EVM, one by one, in a fully sequential manner. Miners are paid for their efforts with units of Ether-prised *gas*, to be paid by the address that requested the transaction. A miner logs the transaction requests they executed, together with the respective effects of the transaction (Ether transfers and field value changes). When the log reaches a certain size, it is packaged by the miner as a new *block*, and suggested as the next block of the Ethereum *blockchain*. A consensus algorithm among other miners in the Ethereum network will then check whether the transactions and the effects reported in the block are in synch (by recomputing all effects and voting on the results). Once consensus is reached, the block is committed as the next block of the blockchain.

[1] called ‘state variables’ in Ethereum terminology

The by far most popular programming language for Ethereum smart contracts is Solidity[2]. Accordingly, we target Solidity smart contracts in this work. Data types include `uint` (unsigned integer), `address` (addresses of users and contracts), enums, structs, arrays, and mappings associating keys with values. For instance, the declaration `mapping (address => uint) public m` declares a field `m` which contains a mapping from addresses to unsigned integers. Fields marked `public` are read-public, not write-public. In general, fields of a contract can only ever be modified by code of the same contract. Solidity offers also some cryptographic primitives, for instance the function `keccak256` computing a crypto-hash of its argument. The statement `require`($b$) checks the boolean expression $b$, and reverts if $b$ is false. If `require` reverts, the entire transaction (function execution) reverts. The same is true for any other potentially reverting statement, like also `transfer` , see below. The current caller, and the amount of Ether sent with the call, are always available via `msg.sender` and `msg.value`, respectively. The default unit used for Ether balance and Ether payments is Wei (= $10^{-18}$ Ether). Units can also be made explicit (like `wei` or `ether`). Only `payable` functions accept payments.

Solidity further features programmable, potentially parameterised, *modifiers*. For instance, the Casino contract in Fig. 2 uses the modifiers `byOperator`, `inState(s)`, and `noActiveBet`. They are implemented in the contract but their declarations are omitted from Fig. 2 for brevity. These modifiers expand to, respectively: The Solidity operations triggering payments and calls deserve special attention as they offer an attack surface addressed in this work. First of all, sending Ether to another contract, and calling another contract, is basically the same mechanism in Ethereum. In particular, sending Ether to an address passes control to the receiver (if the receiver is a contract). This can have problematic consequences of various kinds. One problem that is studied widely in the literature is *re-entrancy*, where the receiver of a call or payment calls back before returning. This can jeopardize the programmer's attempt to fully reflect the external flow of Ether in the values of the internal fields. The problem of re-entrancy is addressed in other works, see for instance (Ahrendt and Bubel 2020). A different problem of control-passing calls and payments, in combination with the transaction concept, is unwanted effects of *reverting calls and payments*. This problem is much less discussed in the literature, and indeed the target of our work.

If an ongoing transaction executes a call or payment to another contract, this opens a *nested transaction*. Whether a revert in the nested transaction also reverts the outer transaction depends on the programming construct being used. The standard call statement `require (msg.sender == operator);` (where $c$ is a contract address and `require (state == s);` is a function of $c$) reverts if execution in $c$ reverts. In contrast, the low-level call statement `require (state != State.BET_PLACED);` does not revert if execution in $c$ reverts, but returns `c.f(...)` in that case. For payments, there is a similar distinction. The statement `f` transfers the amount of $v$ Wei from the caller to $a$. It reverts if $a$ is a contract and the code of $a$ reverts during execution of the transfer. In contrast, the statement `c.call(...)` does not revert if execution in $a$ reverts, but returns `false` in that case.

This means that the reverting of one's own contract code is in the hands of an external party whenever we use statements like `a.transfer(v)` or `a.send(v)`, and *can* lead to

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

a DoS of our contract to its users.[3] To be clear, reverting of code does not have to result in a DoS. It is precisely the aim of this work to analyse whether or not an externally caused revert leads to a DoS.

## 3.2 Extended finite state machines

*Extended finite-state machines (EFSM)* (Cheng and Krishnakumar 1993; Skoldstam et al. 2007) extend finite-state machines (FSMs) with bounded, discrete *variables*, and *guard* and *action* expressions, collectively called *updates*, associated to the transitions. The guards and actions are formulas constructed from variables, integer constants, the Boolean literals $\mathrm{true}$ (T) and $\mathrm{false}$ (F), and the usual arithmetic and logic connectives. 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.

A variable $v$ takes values within a bounded discrete domain $\mathcal{D}(v)$, and has an initial value $v^\circ \in \mathcal{D}(v)$. Let $V = \{v_0, \ldots, v_n\}$ be the set of variables with domain $\mathcal{D}(V) = \mathcal{D}(v_0) \times \cdots \times \mathcal{D}(v_n)$. An element of $\mathcal{D}(V)$ is called a *valuation* and is denoted by $\hat{v} = \langle \hat{v}_0, \ldots, \hat{v}_n \rangle$ with $\hat{v}_i \in \mathcal{D}(v_i)$, and the value associated to variable $v_i \in V$ is denoted $\hat{v}[v_i] = \hat{v}_i$. The *initial valuation* is $v^\circ = \langle v_0^\circ, \ldots, v_n^\circ \rangle$.

A second set of variables, called *post-transition variables*, denoted by $V' = \{\, v' \mid v \in V \,\}$ with $\mathcal{D}(V') = \mathcal{D}(V)$, is used to describe the values of the variables after a transition occurs. Variables in $V$ are referred to as *pre-transition variables* to differentiate them from the post-transition variables in $V'$. The set of all update formulas using variables in $V$ and $V'$ is denoted by $\Pi_V$.

For an update $p \in \Pi_V$, the terms $\mathrm{vars}(p)$ and $\mathrm{vars}'(p)$ denote the set of all variables, and the set of post-transition variables, respectively, that occur in $p$. Updates $p \in \Pi_V$ can thus be interpreted as predicates over their variables, evaluating to T or F, i.e., $p : \mathcal{D}(V) \times \mathcal{D}(V') \to \{\mathrm{T}, \mathrm{F}\}$.

**Definition 1** An *extended finite-state machine (EFSM)* is a tuple $E = \langle \Sigma, S, S^\circ, \to, S^\omega \rangle$, where $\Sigma$ is a set of events; $S$ is a finite set of *locations*; $\to \subseteq S \times \Sigma \times \Pi_V \times S$ is the *conditional transition relation*; $S^\circ \subseteq S$ is the set of *initial locations*; and $S^\omega \subseteq S$ is the set of *marked locations*.

A transition in $E$ is given as $q \xrightarrow{\sigma:p} q'$, which means that if update $p$ evaluates to T, the system can transit from location $q$ to location $q'$ on the occurrence of the event $\sigma$. When the transition occurs the variables in $\mathrm{vars}'(p)$ are updated while the variables not contained in $\mathrm{vars}'(p)$ are unchanged.

EFSMs can be represented 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. 1, where {S0, S1, S2, S3, S4, S5, S6, S7} is the set of locations, all marked as

[3] Using the non-reverting statements `player` and `transfer` instead is often not a solution either. Not reverting on a failed call or payment can cause safety issues. Therefore, most style guides strongly advise to combine `player` and `closeAuction` with a `MAX_BID` on the returned boolean, which brings us back to the problem of a local revert being in the hand of an external party.

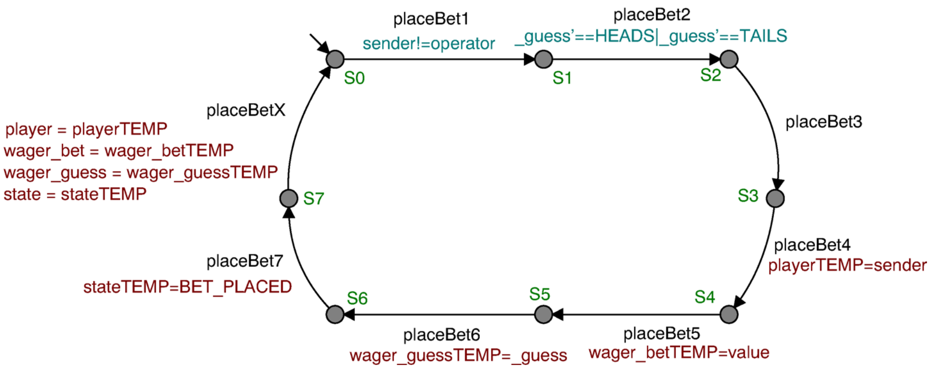


**Fig. 1** EFSM model of the Casino `false` function (see Fig. 2, lines 18–27, and Section 5.2)

denoted by the filled circles, with $\{S0\}$ the initial location (shown by the small arrow), the transition $\{S1\}$ to $\{S2\}$ is labelled by the event `placeBet2`, and is guarded by the expression (`_guess' == HEADS | _guess' == TAILS`), which non-deterministically assigns `HEADS` or `TAILS` to the `_guess` variable. Thus, after the transition, in location $\{S2\}$, the value of `_guess` is either `HEADS` or `TAILS`.

Typically, EFSM models consist of several interacting components. Such a model is called an *EFSM system*, a collection of interacting EFSMs, $\mathcal{E} = \{E_1, \ldots, E_n\}$.

Component interaction in an EFSM system is modeled by *synchronous composition* (Hoare 1985), where *shared* events, that is, events that appear in more than one component EFSM, are *lock-step* synchronized, while other events are *interleaved*. A shared event is thus enabled in the composition if and only if it is enabled by all the EFSM containing that event in their alphabet. Furthermore, updates of transitions labeled by shared events are combined by conjunction.

**Definition 2** Given two EFSMs $E_1 = \langle \Sigma_1, S_1, S_1^\circ, \rightarrow_1, S_1^\omega \rangle$ and $E_2 = \langle \Sigma_2, S_2, S_2^\circ, \rightarrow_2, S_2^\omega \rangle$, the *synchronous composition* of $E_1$ and $E_2$ is $E_1 \| E_2 = \langle \Sigma_1 \cup \Sigma_2, S_1 \times S_2, \rightarrow, S_1^\circ \times S_2^\circ, S_1^\omega \times S_2^\omega \rangle$, where:

$$\begin{aligned}
(x_1, x_2) &\xrightarrow{\sigma:p_1 \wedge p_2} (y_1, y_2) && \text{if } \sigma \in \Sigma_1 \cap \Sigma_2,\ x_1 \xrightarrow{\sigma:p_1}_1 y_1, \\
& && \text{and } x_2 \xrightarrow{\sigma:p_2}_2 y_2\ ; \\
(x_1, x_2) &\xrightarrow{\sigma:p_1} (y_1, x_2) && \text{if } \sigma \in \Sigma_1 \setminus \Sigma_2 \text{ and } x_1 \xrightarrow{\sigma:p_1}_1 y_1\ ; \\
(x_1, x_2) &\xrightarrow{\sigma:p_2} (x_1, y_2) && \text{if } \sigma \in \Sigma_2 \setminus \Sigma_1 \text{ and } x_2 \xrightarrow{\sigma:p_2}_2 y_2\ .
\end{aligned}$$

Note that synchronous composition is associative.
For example, the event `placeBet1` of Fig. 1 synchronizes with the same label in the EFSMs modeling the `c.f(...)` modifier shown in Fig. 8, right, and the `assignSender` of Fig. 9, so that the transition from `S0` to `S1` in Fig. 1 cannot occur unless `a.transfer(v)` is different from the `call` (as required by Fig. 1), and `send` (as required by Fig. 8), and `assignSender` is in its `S0` location (see Fig. 9).

The global behavior of an EFSM system $\mathcal{E} = \{E_1, \ldots, E_n\}$ is given by $E_1 \| \cdots \| E_n$.

Non-blocking of an EFSM system, is defined on the *flattened* system (Mohajerani et al. 2016), where the EFSMs and the variables have been converted into ordinary FSMs. For EFSMs each location becomes one state and the transitions are labeled by their respective events, but for variables this is more involved. Essentially each value that may be assigned to a variable is represented by an explicit state with transitions between these states corresponding to the updates (Mohajerani et al. 2016).

**Definition 3** Let $E = \langle \Sigma, S, S^\circ, \rightarrow, S^\omega \rangle$ be an EFSM with variable set $\mathrm{vars}(E) = V$. The *monolithic flattening* of $E$ is $U(E) = \langle \Sigma, S_U, \rightarrow_U, S^\circ_U, S^\omega_U \rangle$ where

- $Q_U = Q \times \mathcal{D}(V)$;
- $(x, \hat{v}) \overset{\sigma}{\rightarrow}_U (y, \hat{w})$ if $E$ contains a transition $x \xrightarrow{\sigma:p} y$ such that $p(\hat{v}, \hat{w}) = \mathrm{T}$;
- $Q^\circ{}_U = Q^\circ \times \{v^\circ\}$;
- $Q^\omega_U = Q^\omega \times \mathcal{D}(V)$.

$U(E)$ is the FSM representation of the EFSM, where all the values $\hat{v}$ of all the variables $v$ have been embedded into the state set $Q_U$. This ensures the correct sequencing of transitions in the FSM. The monolithic flattened EFSM system $\mathcal{E}$ is denoted $U(\mathcal{E}) = U(E_1 \| \ldots \| E_n)$. Non-blocking for an EFSM system can be defined in multiple ways, for instance, based soleley on marked locations or also considering marked variables. In this work, we define non-blocking for an EFSM system considering both marked locations and marked variables which is equivalent to $U(\mathcal{E})$ being non-blocking;

**Definition 4** An EFSM system $\mathcal{E}$ is non-blocking if $U(\mathcal{E})$ is non-blocking.

So, the task of determining whether an EFSM system is non-blocking or not boils down to determining whether the synchronous composition of the flattened system can always reach some marked state. Though non-blocking verification is performed on the underlying flattened FSM, so that EFSM is just an intermediate representation between the Solidity source code and the FSM, there are benefits of using EFSMs. One benefit is the model compactness that comes from allowing bounded, discrete variables; for example, the FSM model of a variable with a 0..255 domain has 256 states, whereas in the EFSM formalism this can be a single variable with just that domain. Also, EFSMs allow to associate guards and actions to transitions, which aligns closely with the source code, making it easier to interpret counterexamples on the original source code.

In general, verifying non-blocking is computationally hard, but there exist tools that can perform this for systems of considerable sizes as measured by the number of states and transitions. One such tool is Supremica (Akesson et al. 2006).

## 3.3 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 models of smart contracts to always be able to reach some state where certain properties hold, such as being able to pay out the funds held by the contract.

A benfit of using SUPREMICA is that the flattening of the EFSM model is fully automatic, using *partial unfolding* (Mohajerani et al. 2013) which results in an FSM model structure beneficial for the verification procedure, thus potentially allowing to verify larger contracts within reasonable time and memory limits.

To verify non-blocking, *conflict check* (Mohajerani et al. 2016; Malik et al. 2023) of SUPREMICA is used. If the verification determines that the system is blocking, a counterexample is provided, a trace that leads to a blocking state, that is, a state from where no marked state can be reached. This counterexample may be replayed in SUPREMICA's simulator to reveal the core of the problem.

## 4 Use cases

Two use cases are investigated in this paper, a (simple) on-line Casino contract (VerifyThis 2021), and an Auction contract[4]. Both of these contracts are prone to DoS attacks, as is revealed by the respective EFSM models being blocking.

The Casino contract allows an operator to open a casino by submitting to the contract a secret number on which players can then bet heads or tails. The operator eventually reveals whether the secret number was heads or tails, and any winnings are then transferred to the respective players, while the operator can retrieve what remains after having paid out to the winners.

The Auction contract implements an auction where bidders submit their bids, and if the submitted bid is higher than the current bid, the new bidder is recorded as the one with the currently highest bid, and the previous current bidder is reimbursed its bid. The auction owner can at any time close the auction, at which point the current bid is transferred to the owner.

### 4.1 The Casino smart contract

The Solidity code of the Casino contract is shown in Fig. 2[5]. The implementation features three explicit states: `call`, `send`, and `require`, see line 3, defined by the `placeBet` type `inState`.

Based on the modifier **`sender`**, in the `operator` location the operator may create a game by invoking the `state==GAME_AVAILABLE` function (line 13). To ensure a fair betting, the Casino must place its bet at the time of game creation. Thus, when calling `IDLE`, `GAME_AVAILABLE` is assigned a value (line 15) to later decide the game outcome. After creating a new game, the state changes to `BET_PLACED` (line 16) where a game is now available. In this state, the player can call **`enum`** to place a bet, up to the size of the pot, on

[4] A variant of the contract OneAuction in Ahrendt and Bubel (2020)

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

```
1  contract Casino {
2
3    enum State {IDLE, GAME_AVAILABLE, BET_PLACED}
4    enum Coin {HEADS, TAILS}
5    struct Wager {uint bet; Coin guess;}
6
7    State private state;
8    bytes32 public hashedNumber;
9    address public operator, player;
10   Wager private wager;
11   uint public pot;
12
13 function createGame(bytes32 hashNum) public
14   byOperator, inState(State.IDLE) {
15   hashedNumber = hashNum;
16   state = State.GAME_AVAILABLE;}
17
18 function placeBet(Coin _guess) public payable
19   inState(State.GAME_AVAILABLE) {
20   require (msg.sender != operator);
21   require (msg.value > 0 && msg.value <= pot);
22   player = msg.sender;
23   wager = Wager({
24     bet: msg.value,
25     guess: _guess
26   });
27   state = State.BET_PLACED;}
28
29 function decideBet(uint secretNumber) public
30   byOperator, inState(State.BET_PLACED) {
31   require (hashedNumber ==
32         keccak256(secretNumber));
33   Coin secret = (secretNumber % 2 == 0)? Coin.HEADS : Coin.TAILS;
34   if (secret == wager.guess) {
35      playerWins();
36   } else {
37      operatorWins();
38   }
39   state = State.IDLE;}
40
41 function playerWins() private {
42   tmp = wager.bet;
43   wager.bet = 0;
44   pot = pot - tmp;
45   player.transfer(tmp*2);}
46
47 function operatorWins() private {
48   pot = pot + wager.bet;
49   wager.bet = 0;}
50
51 function addToPot() public payable
52   byOperator {
53   pot = pot + msg.value;}
54
55 function removeFromPot(uint amount) public
56   byOperator, noActiveBet {
57   pot = pot - amount;
58   operator.transfer(amount);}
59 }
```

**Fig. 2** Solidity code for Casino (some details are omitted)

HEADS or TAILS (lines 20-25). This then changes the state of the contract to `State` (line 27).

Next, the operator may by `inState(s)` submit the original secret number to resolve the bet (line 29). If the secret number is even the coin toss is HEADS, else it is TAILS (line 33).

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 43–44). Then, double the bet is transferred from the contract to the player (line 45). If the operator wins, the bet is added to the pot and then set to zero (lines 48–49).

The operator may add money to the pot at any state, `IDLE` (line 51). Also, the operator may remove money from the pot, `createGame` (line 55), but only if the player has not placed a bet, that is, if the casino is not in the state `createGame`. This is ensured by the modifier `hashedNumber`.

## 4.2 The Auction smart contract

The Solidity code of the Auction contract is shown in Fig. 3. The contract begins by defining a Boolean variable `auctionOpen` and initializing it to `true`, line 2, to denote that the auction is open to accept biddings. Functions `GAME_AVAILABLE` (lines 11–24) and `placeBet` (lines 26–31) can only be called when the auction is open (see lines 13 and 28, respectively). The `BET_PLACED` is initialized to 0 (line 3), and the `decideBet` is

```
1  contract Auction {
2      bool public auctionOpen = true;
3      uint public currentBid = 0;
4      address private auctionOwner;
5      address private currentBidder;
6
7      constructor() public {
8        auctionOwner = msg.sender;
9      }
10
11 function placeBid() public payable {
12       require (msg.sender != auctionOwner);
13       require (auctionOpen);
14       require (msg.value > currentBid);
15
16       address oldBidder = currentBidder;
17       uint oldBid = currentBid;
18       currentBidder = msg.sender;
19       currentBid = msg.value;
20
21       if (oldBid != 0) {
22         payable(oldBidder).transfer(oldBid);
23       }
24     }
25
26 function closeAuction() public {
27       require (msg.sender == auctionOwner);
28       require (auctionOpen);
29       auctionOpen = false;
30       payable(auctionOwner).transfer(currentBid);
31     }
32 }
```

Fig. 3 Solidity code for Auction contract (some details are omitted)

set to be the one constructing the auction (line 8). A bidder can place their bid by calling the `addToPot` function (line 11). If the bid placed is higher than the current bid (line 14), `removeFromPot` is updated to the caller of `BET_PLACED` and `noActiveBet` is updated to the new bid (lines 16–19). The old bid amount, if there were such, is then transferred back to the previous highest bidder through `placeBid` (line 22). The auction owner can close the auction by calling function `closeAuction` (line 26), which sets `currentBid` to false (line 28), preventing bidders from placing any bids further, and transferring the current highest bid to themself (line 30). Once the Auction is closed, there is no way to re-open it or (meaningfully) interact with it.

## 5 Automatic conversion to EFSMs

The automated conversion traverses the source code's *abstract syntax tree* (AST Wikipedia 2024), which is obtained in JSON (ISO/IEC 21778 2017) 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.

For the Casino contract, the automatically converted EFSM model differs significantly from the manually crafted model presented by Mohajerani et al. (2022). One difference is that the automatically converted model describes only the plant behavior, it includes no specification, whereas the model given by Mohajerani et al. (2022) has the specification embedded in a rather complicated way. Another difference is that the `auctionOwner` 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, `placeBid` is represented by an EFSM variable `state`, much like in the Solidity code, and the modifiers are thus explicitly modeled.

### 5.1 Modeling variables

As EFSMs allow bounded, discrete variables, Solidity variables are directly converted to EFSM variables. However, some care has to be taken when converting unbounded Solidity types to bounded EFSM variables. Particularly, in the Casino contract, the `currentBidder` variable cannot be modelled since if `placeBid` is modelled as an (upper) bounded variable, then a trivial blocking trace calls `currentBid` enough times for `oldBidder.transfer(oldBid)` to reach its upper bound, plus once more, and then the system deadlocks. The `closeAuction` variable is automatically ignored by adding it to an *ignore list*, so the converter does not generate any `auctionOpen` variable in the EFSM model, nor any transitions for statements involving `State` (lines 21, 44, 48, 53, 57).

Also the Auction contract has a problematic variable, `State`, that causes trivial blocking when the bidding has reached is upper bound, `pot`. The `pot` clause on line 14 of Fig. 3, requires that a new bid is always higher than the current bid, which is not possible once the current bid has reached `addToPot`, and then the `pot` function will always revert. This is a problem, not only for the EFSM model, but also for the running code, since Solidity's uint type is upper bounded to $2^{256} - 1$. Thus, it is impossible to show that a bid can always be successfully placed, since this is not true, neither for the EFSM model nor the actual code. However, in the Auction model, we cannot simply ignore everything related to pot, as was done with pot, so the pot clause is modelled and issues related to this are discussed in Section 6.

Solidity contracts typically have a *constructor* (shown in Fig. 3, lines 7–9, but not shown in Fig. 2) that assigns initial values to some variables, and these are initialized accordingly when converted to EFSM variables. But many variables have unknown initial values, which can be modelled by non-deterministic assignments over the entire range of the variable domain, see for instance the assignment to _guess on the transition from S1 to S2 in Fig. 1.

Additionally, variables of mapping type are handled by first recognizing the key-value structure. Then, the list of already defined variables of the same type as the key is used to generate a new set of variables that are defined as type of the value. Variable withdrawable defined as mapping type with key-value structure as (address=>uint) at line 60, Fig. 18, is converted to variables namely withdrawable_player and withdrawable_operator which are of int types.

## 5.2 Modeling functions

Each function 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. 2. 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 currentBid function is shown in Fig. 1; this models lines 18–27 of Fig. 2. The initial eventplaceBet1 labels the transition from the initial location S0, and the MAX_BID clause at line 20, ensuring that **require** is not the caller of MAX_BID is added as a guard. The handling of the modifiers and the placeBid clauses, lines 19–21 are described in Section 5.3, below.

Inside the currentBid function there is an assignment to the pot variable (lines 23–25), which is of type **require** (line 5). Since there are no structs in SUPREMICA, the struct constructor call of decideBet has to be "flattened". This is automatically done, and results in two distinct variables wager_guess and wager_bet.

The placeBet function is called with a parameter **require** of type operator, which is an enum (its definition is not shown in Fig. 2, but see line 33) with two values, HEADS and TAILS. The converter collects this type information, and since the actual value of placeBet is unknown for the **require** call, a non-deterministic assignment is made to the variable placeBet, see the guard on the transition labeled placeBet2 of Fig. 1.

When a function is called from within another function, this is modeled by a self-loop labeled by the called function's initial event, and with the called function's final event label-

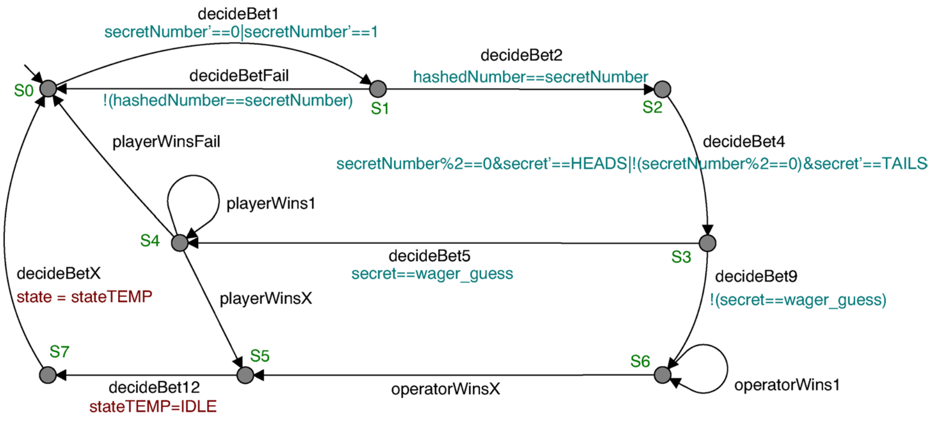


**Fig. 4** EFSM model of the `Coin` function, Fig. 2, lines 29–39

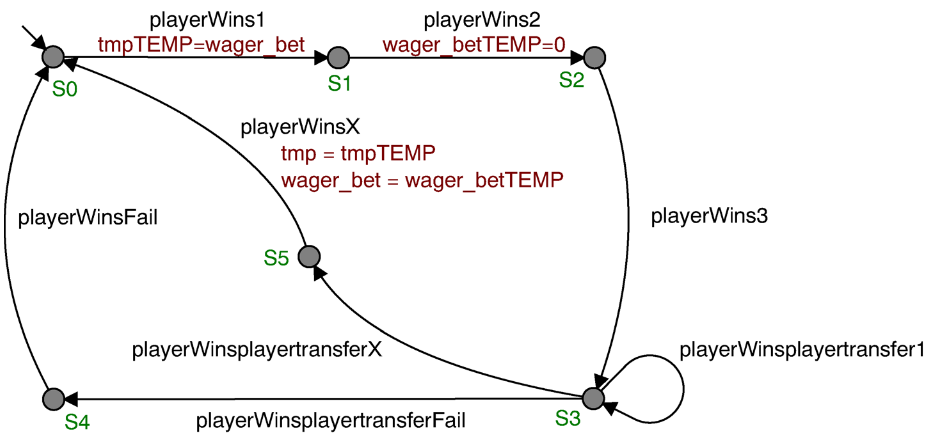


**Fig. 5** EFSM model of the `_guess` function, Fig. 2, lines 41–45

ing a transition from the self-looped location. This is illustrated in Fig. 4, where the `wager` and **`struct`** `Wager` functions are called in the locations labeled `S6` and `S4`, respectively. This corresponds to lines 37 and 35, respectively. When the EFSM of Fig. 4 is in, say, `S4`, it cannot transit to `S5` until the `playerWinsX` event is enabled, which requires the EFSM that models `wager` (Fig. 5) 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, `decideBet` initiates the execution of `placeBet`, and then waits for `_guess` to return.

Note that though lines 23–26, and 32 are in the AST designated as *FunctionCall*, these are treated separately and do not result in self-loops. The initialization of the `operatorWins` struct variable, is described above. The external hashing function `playerWins` (line 32) is not modelled at all, as its internal workings are not known. This is handled by adding `playerWins` to the abovementioned *ignore list*, so that the call `playerWins` is automatically converted into simply `playerWins`.

### 5.3 Modifiers and require statements

Modifiers are modeled as single-location EFSMs, with self-looped transitions with the Boolean expression as guard and labeled by the initial events of the functions that the modifiers and/or `playerWins` clauses relate to. For instance, the `placeBid` modifier relates to the `closeAuction`, `byOperator`, `inState(s)`, and `wager` functions, and so the self-looped transition is labeled by their initial events, see Fig. 8, left. Modifier **`keccak256`**, see Fig. 8, right, asserts the current state of the Casino contract by having a parameter `_state`, thus enabling certain functions while disabling others, which is why **`keccak256`** has multiple self-loop transitions instead of one.

Along with modifiers, it is common practice to use **`require`** statements within the function for additional checks. Modeling of `byOperator` is done by adding the Boolean expression as a guard on the transition. For instance, `createGame` statement on line 20 in the Casino contract is converted as a guard for the transition from node `S0` to `S1` in Fig. 1. However, guards corresponding to `addToPot` clauses that contain parameters passed to the function must be added after the non-deterministic assignment of the parameter, along with a transition back to the source node, in case the guard evaluates to `false`. For instance, in the Casino contract, `removeFromPot` statement on line 32 containing parameter `decideBet`, gets converted to the guard on transition from node `S1` to `S2` in Fig. 4, after `inState(s)` has been non-deterministically assigned. Case where the condition specified for `inState(s)` on line 32 is false is modeled by the transition `decideBetFail` from `S1` to `S0`.

### 5.4 Modeling framework behavior

The above discussion and models deal with what can be converted directly from the Solidity source code. 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 **`require`** variable, and the behavior of **`require`**. As this behavior is not possible to extract from the code, these models are manually predefined, but in a generic way.

#### 5.4.1 Assign sender and value

Within the Solidity framework, contracts interact with each other by calling public functions. With each such call follows a data packet **`require`**, which includes among other things a reference (address), **`require`**, and the amount of Wei sent with the message, **`require`**, to the contract that called the function. The assignment to `secretNumber` and `secretNumber` is handled by the framework, outside of the Solidity code. In the Casino contract there are two participants, **`require`** and **`msg.sender`**, 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 **`transfer`** and **`transfer`**. This is done by the EFSM of Fig. 9.

The self-loop in the `S0` location, labeled by the `assignSev` event, non-deterministically assigns the variable `sender`the "address" `x0001` (player) or `x0002` (operator) and `value` to either 0 or 1. Since not much can happen with the Casino until the operator has created the game, see lines 13–16, the initial value of `sender` is set to the address of the operator. This might be changed by the non-deterministic assignment in the self-loop, but since **`msg`** 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 eventof each public function. From `S1` is then a transition back to `S0` labeled with the final eventsof the public functions. In this way, `sender`is assigned an address in location `S0`, representing either **`msg.sender`** or **`msg.value`**, and this address remains constant while any public function executes, as the EFSM is in its `S1` location.

### 5.4.2 Modelling transfer

When a **`msg.value`** occurs, control is passed to the receiver (see Section 3.1) that can choose either to accept or reject the transferred funds. The EFSMs of Fig. 10, model this by having an transition from the initial state `S0` to `S1` representing the transfer to the receiver, and two transitions back from `S1` to `S0`, one of which represents accepted transfer, and the other (ending with `Fail`) representing rejected transfer.

Unique event names corresponding to `player` to an address is generated by concatenating the name of the function in which the `operator` occurs with the value of the `address` variable and the identifier `transfer`. For instance, in Fig. 5, the self-loop transition on location `S3` models initiating a **`msg.sender`** to **`msg.value`**. The event name, `playerWinsplayertransfer1`, is generated by concatenating the function name `createGame` with the address `player` and, `transfer1`, which makes up the initial transition of that particular `operator` model, see Fig. 10, left. This naming convention avoids unintentional synchronization with transfers to the same recipient in other functions. Consequently, distinct EFSMs modeling the outcome of **`transfer`** are generated for the same address from different functions.

When a variable, such as **`transfer`**, holds the address of a recipient of a **`transfer`**, such as in Fig. 18, line 70, instead of a self-loop initiating the **`transfer`**, multiple transitions originate from the same location, see `S2` in Fig. 11, each of which corresponds to a predefined address.

Rejection of `player` is modelled in a calling function by a transition from the location where `playerWins` is called to a location from where a transition representing the unsuccessful completion of the function leads to the initial location. For example, in the `player` EFSM (Fig. 5), **`transfer`** to **`transfer`** (Fig. 2, line 45) is initiated by the self-loop, and then rejection of the transfer is represented by the `playerWinsplayerFail` transition, which is followed by the `playerWinsFail` that represents reverting of the **`msg.sender`** function.

As mentioned in Section 3.1, when a **`transfer`** is rejected and the calling function reverts, all effects of the function containing the **`transfer`** are restored to the state before the call of the function, and this propagates upwards along the call chain. This means that when modeling a function, a *Restore-on-Revert* mechanism has to be implemented. There are several ways to implement this, but what seems simplest is to have a function that could potentially revert work on temporary *shadow* variables rather than the actual variables, and then to assign the values of the shadow variables to the actual variables on successful completion of the function. In this way, if a revert occurs and the function completes unsuccess-

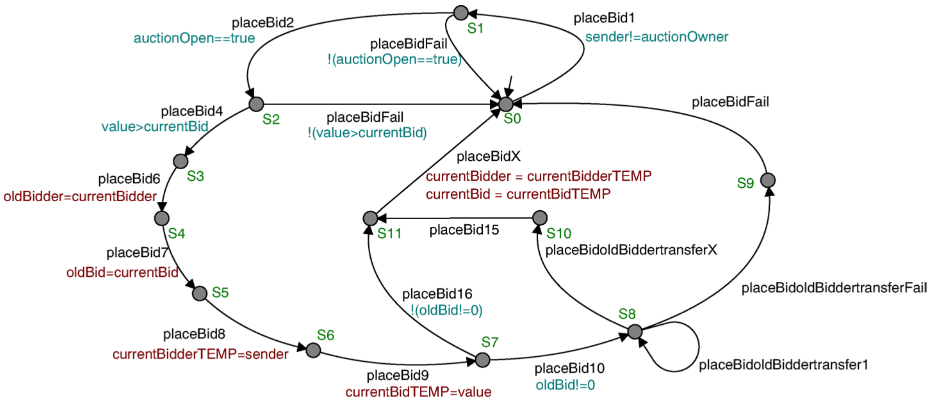


**Fig. 6** EFSM model of the `placeBet` function, Fig. 3, lines 11–24

fully, the actual variables have not changed value. Such shadow variables have to be used for all global variables used in a function, but not for local variables or parameters as when the function reverts these are just forgotten.

In Fig. 6, the shadow variables are suffixed with "TEMP", and as can be seen, these are assigned within the function instead of the actual variables, and then the actual variables are assigned from their shadows on the `withdraw` transition, which represents successful completion of the function.

## 5.5 Overview of the models

The automatically converted EFSM models do not include any specification, only the behavior of the code is modeled, so in SCT terms these make up the *plant*. The plant is unmarked, meaning that all EFSMs and all variables have all their locations and domain values, respectively, marked. This is natural, as marking can be regarded as a type of specification, the plant does not have any idea about what it is to be used for. Verification of the plants show then to be non-blocking, as is expected.

For the Casino contract, the plant model consists of 13 EFSMs and 26 variables[6], while the Auction plant model has 5 EFSMs and 11 variables. The biggest EFSMs of Casino, **`transfer`** and `playerWins`, have 8 locations, while the biggest one for Auction is **`transfer`** with 12 locations. The flattened Casino plant model has 3112 states, 88 events, and 8064 transitions, while the flattened Auction plant model has 9436 states, 65 events, and 48244 transitions.

Many of the EFSMs have only a single location with self-loops. Notably all EFSMs that model modifiers, see Fig. 8. However, also `player` of Casino is a single location with a single self-loop labelled by `addToPot1`. This is because `playerWins` only changes the value of the **`transfer`** variable, which as discussed in Section 5.1 must be omitted from the conversion.

Many of the variables are 0–1 variables, but notably for Auction the variables related to bids and bidding, **`transfer`**, `placeBidX`, and their shadow variables, plus `decideBet`

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

have larger domains to make bidding at all possible. The `state` variable of Casino has the same symbolic domain as in the contract `placeBet`, `placeBid`, and `addToPot`. In Casino, the `sender` variable has a binary domain as there are only two parties involved, the operator and the player. Though there could be more than one player involved, the verification results presented in Section 6 do not change when the number of players increase. For the Auction, `sender`has a domain of size three, the auction owner and two bidders. Again, though there could be more than two bidders involved, the verification results do not change with a larger domain for `sender`.

The task is now to add a specification to the plant models to verify whether they can still exhibit desired behaviour with one of the parties exhibiting a malicious behavior.

## 6 Non-blocking verification

Although the generated plant models account for failing transfers, the issue investigated in this paper, DoS by rejecting transfer (SWC Registry 2020), concerns a malicious party that once rejecting a transfer will always reject any re-transfer. Bad about this malicious behavior is that the ones exhibiting it can at a small financial cost to themselves cause large financial damage to the other parties involved with the contract.

Non-blocking is a progress property that guarantees that from every reachable state of a system, some marked state can always be reached. Ramadge and Wonham (1989) formally defined this as $\mathcal{L}(S) = \overline{\mathcal{L}_m(S)}$, where $\mathcal{L}(S)$ is the set of all possible traces of the system $S$, and $\mathcal{L}_m(S)$ is the set of all traces reaching marked states; $\overline{\mathcal{L}_m(S)}$ then defines all *prefixes* of these marked traces. In Computation Tree Logic (CTL, Baier and Katoen 2008) this can be expressed as `AG EF` *marked*. Carefully selecting the marked states of the system, non-blocking verification can determine if the system can always reach some (marked) state from where desired behavior can be effected. For smart contracts, such desired behavior could be to always pay out funds to the respective owner, that is, to guarantee liquidity. A generalization of this desired behavior is for all transactions to complete successfully.

Note that though the specifications as presented here are specific to the use cases, they are in fact generic in the sense that the structure will be the same for other use cases, only the event labels will change.

### 6.1 The specification

The automatically converted models of Section 5 describe the overall behavior of the smart contracts as implemented by the code, so these models have all locations marked and take the place of the plant. To verify properties, specifications that express those properties need to be added, and these specifications have to be such that non-blocking verification actually says something meaningful about the system.

Two different models are added as specification, both expressed as FSMs, the *attacker model* and the *progress specification*. The first one describes the malicious behavior that the attacker exhibits, while the second one describes the desired behavior that should always be possible even under the attack. With these two specification models added to the plant, if the system is verified to be non-blocking then this means that the malicious behavior of the attacker cannot prevent the desired behavior.

Separating the attacker model, the progress specification, and the plant is beneficial as different models can be taken out and "plugged in", without (ideally) changing any of the other models. This *modular structure* is also beneficial for the compositional abstraction-based non-blocking verification algorithms (Mohajerani et al. 2016) implemented by SUPREMICA.

### 6.1.1 The attacker model

Attacker models are introduced to explicitly capture pessimistic assumptions of the environment, where the environment (including attackers) can exercise actions attempting to prevent the system or other users from reaching its/their goal. Having attacker models allows to model the adversarial behavior explicitly, and to verify whether the system remains non-blocking even under attack.

The DoS by rejecting transfer attack, also known as "Dos with Failed Call" (listed as number 113 by SWC Registry 2020), exhibits the malicious behaviour that once rejecting a transfer the attacker always rejects a transfer. Figure 12 shows the attacker models for Casino (left) where the player is considered malicious, and Auction (right) where a bidder is malicious. These models capture the core adversarial behavior relevant to the SWC-113 DoS vulnerability regardless of whether the failed call is caused by a rejected transfer or another source. The models synchronize with the events of the respective `addToPot` models of Fig. 10. The attacker models stay in their respective initial state until a rejected `pot` occurs, whereafter only rejected `currentBid` are enabled.

As the receiver of a transfer can always choose to reject it, and thus an attack cannot be prevented, both states of the attacker model are marked so that the attack on its own cannot cause blocking. What is to be investigated is whether the consequences of the attack are harmful, which is captured by the progress specification.

These attacker models capture a simple but very important scenario. There is nothing that prevents the use of more complicated attacker models, though, as long as these attacks can be modeled by EFSMs.

### 6.1.2 The progress specification

In addition to specifying that the malicious party keeps rejecting the transfer, the desired global state(s) also have to be specified so it can be determined whether the malicious behaviour can prevent the system from reaching this state (or states), and thus actually cause harm.

The generic progress specification is shown in Fig. 13. This model creates a one-to-one correspondence between the possibility of reaching the marked state (which is the non-blocking property), and the possibility of executing the $W$ event multiple times (Fig. 13, left) or at least once (right). Thus, such a specification allows to use non-blocking verification to determine whether a specific event, $W$, can always eventually occur, which represents meaningful progress of the smart contract.

The set of events of the plant alphabet $\Sigma$, excluding $W$, is denoted $E = \Sigma \setminus \{w\}$ in Fig. 13. Let $G$ be the plant automaton (which can consist of multiple automata, as described in Section 5) and $K$ the progress specification of Fig. 13. The composition of these two is then denoted $G||K$. By definition, the states marked in $G||K$ are the ones marked by both automata, that is $S^{\omega}_{G||K} = S^{\omega}_{G} \times S^{\omega}_{K}$ (see Def 2). This means that in $G||K$ a state is marked when $G$ is in a marked state and $K$ is in its marked state `S1`.

As mentioned above, $G||K$ being non-blocking means that $\mathcal{L}(G||K) = \overline{\mathcal{L}_m(G||K)}$, that is, every trace of the prefix-closed language $\mathcal{L}(G||K)$ is a prefix of some trace of the marked language $\mathcal{L}_m(G||K)$; this means that from every reachable state some marked state can always be reached.

For the left progress specification of Fig. 13, call it $K$, which is applicable when $W$ is required to occur an unbounded number of times, the marked language consists of all possible traces ending with $W$, that is, $\mathcal{L}_m(K) = \mathcal{L}(K) \cap \Sigma^* w$, with $\Sigma^* w$ denoting the concatenation of all traces of $\Sigma^*$ with $W$. Since $\Sigma_G = \Sigma_K$ it holds that $\mathcal{L}_m(G||K) = \mathcal{L}_m(G) \cap \mathcal{L}_m(K)$, and thus $\mathcal{L}_m(G||K) = \mathcal{L}_m(G) \cap \mathcal{L}(K) \cap \Sigma^* w$. Also, since all plant locations and variable domain values are marked, all plant states are marked, so that $\mathcal{L}(G) = \mathcal{L}_m(G)$. Therefore, $\mathcal{L}_m(G||K) = \mathcal{L}(G) \cap \mathcal{L}(K) \cap \Sigma^* w$. Again due to $\Sigma_G = \Sigma_K$, it holds that $\mathcal{L}(G||K) = \mathcal{L}(G) \cap \mathcal{L}(K)$, and thus $\mathcal{L}_m(G||K) = \mathcal{L}(G||K) \cap \Sigma^* w$, that is, all traces possible in $G||K$ and ending with $W$. Now, $G||K$ being non-blocking means that $\mathcal{L}(G||K) = \overline{\mathcal{L}(G||K) \cap \Sigma^* w}$, that is, every trace can be extended to a trace that ends with *w*; or put another way, from every reachable state there exists a trace that ends with *w*.

The above means that the event *w* can be selected as any event in the system alphabet $\Sigma$, and non-blocking verification can then determine whether this event can always occur or not; if the verification determines the system to be blocking, there are reachable states from where *w* cannot eventually occur. Generally, smart contracts offer mutually exclusive outcomes to the involved parties and it is reasonable to guarantee that the outcomes are independently reachable. An automatic procedure could even try *all* events, one by one, and flag whenever the system is determined to be blocking for one of them.

For the Casino, from the operators perspective the ability to always successfully remove any winnings from the pot is crucial, else the liquidity of the contract is compromised. Thus, it is natural to choose *w* to be `removeFromPotX`.

For the Auction, the ability to successfully place a bid is crucial, so in that case the left progress specification of Fig. 13 is used with *w* as `oldBid`. However, the Auction has an added property that it should always be possible to successfully close it, that is, that it should always be possible for **`msg.value`** to occur. But since after closing the Auction, it is not possible to (meaningfully) interact with it again, it cannot be required that `IDLE` occurs an unbounded number of times, which is why the right progress specification of Fig. 13 is used with *w* equal to `GAME_AVAILABLE`.

## 6.2 Counterexamples

Non-blocking verification is done by running Supremica's *conflict check* on the automatically converted EFSM models together with the relevant specifications. If blocking issues are discovered, counterexamples are generated by Supremica. These are traces that lead from the initial state of the system, to a state from where it is not possible to reach any marked state. These traces can be played back in Supremica, and manually stepped through event by event, to give insight into why the issue occurs.

If Supremica reports that the system is non-blocking, then it has exhaustively checked the state-space of the system so it is guaranteed, relative to the given specification, that no blocking issues are present. The non-blocking verification algorithms are presented in Mohajerani et al. (2016) and Malik et al. (2023).

### 6.2.1 The Casino counterexample

For the Casino contract the counterexample found is 56 steps long (so for space-saving only the last part is shown), much longer than the 10-step counterexample for the manually crafted model (Mohajerani et al. 2022), among other things due to the more detailed model with the shadow variables that are not present in the model of Mohajerani et al. (2022). But the two models block in the same 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. Specifically, this means that `BET_PLACED` cannot be successfully completed, and so the funds stored in the pot are locked in forever.

To find the core problem, inspecting the code reveals that **`transfer`** can only be called by the operator when the contract has no active bet, see Fig. 2, line 56, that is, when the `state` variable has the value either **`transfer`** or **`transfer`**. Looking at the counterexample (Fig. 14) and relating it to the code (Fig. 2) maps the blocking to line 45. If `placeBidX` fails, line 39 of `closeAuctionX` will not be executed and thus `state` will not be assigned the value `closeAuctionX`, which prevents `closeAuctionX` to successfully complete. It also prevents `removeFromPot` from re-initializing the game (line 14). Though `removeFromPot` can be called again, when a malicious player refuses the `IDLE` on each call, the contract will never progress to reach its `GAME_AVAILABLE` state. Thus, the funds of the Casino can never be retrieved, so that its liquidity is compromised.

### 6.2.2 The Auction counterexamples

The Auction, though smaller in size of code compared to Casino, has multiple properties interesting to verify. In all cases the experiments are done with the automatically converted plant model of the Auction smart contract (Fig. 3), together with the AttackerModel of Fig. 12, right.

An important property to verify is if the Auction can always be successfully closed. If this is not always possible, a malicious attacker can disrupt the Auction contract to prevent it from being closed.

To verify this, the progress specification on the left in Fig. 13 with `player.`**`transfer`**`(wager.bet*2)` as $w$ is added to the model. This model blocks. Investigating the counter example (Fig. 15) shows that the blocking happens because `decideBet` cannot occur more than once. This is reasonable, since after the auction is closed there is no meaningful way to interact with it, the owner and bidders can try, but all calls are reverted (Fig. 3, lines 13 and 28). Thus, this progress spec requires too much, and breaking it is legitimate.

The progress specification on the right in Fig. 13, with $w =$ `operator`, models that whatever happens after `decideBet` has occurred the first time is fine. With this progress specification the system is non-blocking, meaning that even under attack by a malicious bidder, the auction can always be closed.

Now that it is known that the auction can always be successfully closed, even when under attack (and under other issues, see below), the progress specification for **`transfer`** can be removed. In fact, the progress specification for `IDLE` *must* be removed to verify other properties, since otherwise, trivial counterexamples are obtained in which the auction is first closed and then no meaningful event can occur, simply because the auction has already been closed. As discussed in the following paragraphs a separate specificationis introduced that prevents the auction from closing, and this specification of course conflicts with the progress specification for `closeAuctionX`.

Another important property to verify is whether a bid can always be successfully placed. For this, the progress specification to the left in Fig. 13 with $w =$ `closeAuctionX` is added to the model. Non-blocking verification shows that this is blocking. Inspecting the counterexample, Fig. 15, right, shows that the blocking occurs due to the auction having been closed; after `closeAuctionX`, which is always possible (see above) bids can no longer be placed due to `closeAuctionX` reverting `closeAuctionX` on line 13, Fig. 3. However, this is a legitimate reason for not being able to place a bid, as no (meaningful) interaction can be had with a closed auction.

The progress specification over `placeBidX` expresses that *under all circumstance should it always be possible to successfully place a bid*, but this is too strong. What needs to be expressed is that *assuming that the auction is not successfully closed, it should always be possible to successfully place a bid*. This can be expressed by adding a specification that globally disables `closeAuctionX` and then verify the progress specification over `closeAuctionX`. If this is non-blocking, then it is known that the only issue is with `closeAuctionX` so that the progress specification holds unless `placeBidX` is allowed. So we add a `closeAuctionX` specification, which is an automaton with a single marked and initial state, and a blocked events list containing only `auctionOpen == `**`false`**. Of course, this requires the `placeBid` progress specification to be removed, otherwise it would conflict with the `placeBidX` specification as it can never happen that `closeAuctionX` is always executable, while at the same time that event is disabled.

This blocks. In fact, it deadlocks in a non-marked state, as SUPREMICA's deadlock check shows and gives a 5-step counter example trace to (Fig. 16, left). This trace immediately calls `placeBidX` which then executes until state `S5` (Fig. 7) from where only `closeAuctionX` is enabled, but which the `closeAuctionX` specification prevents.

So the `disableCloseAuctionX` specification is replaced by the `closeAuctionX` specification, which globally disables `closeAuctionX`, the initial event of the

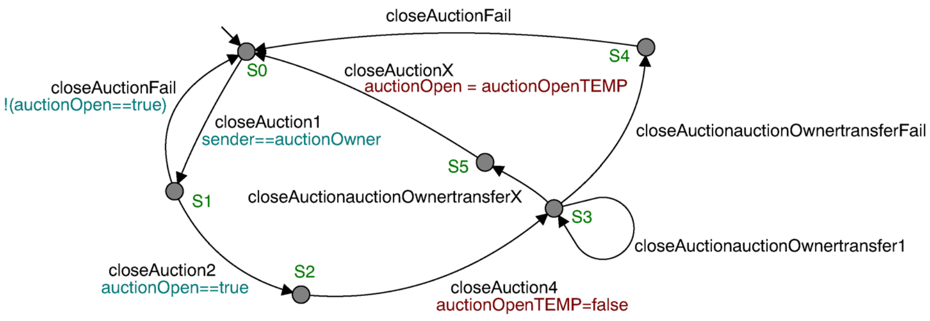


**Fig. 7** EFSM model of the `guess` function, Fig. 3, lines 26–31

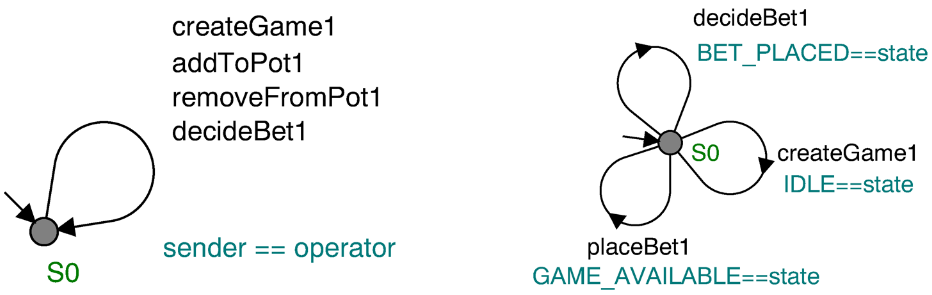


**Fig. 8** EFSM models of (left) the `keccak256(secretNumber)` modifier, and (right) the `secretNumber` modifier

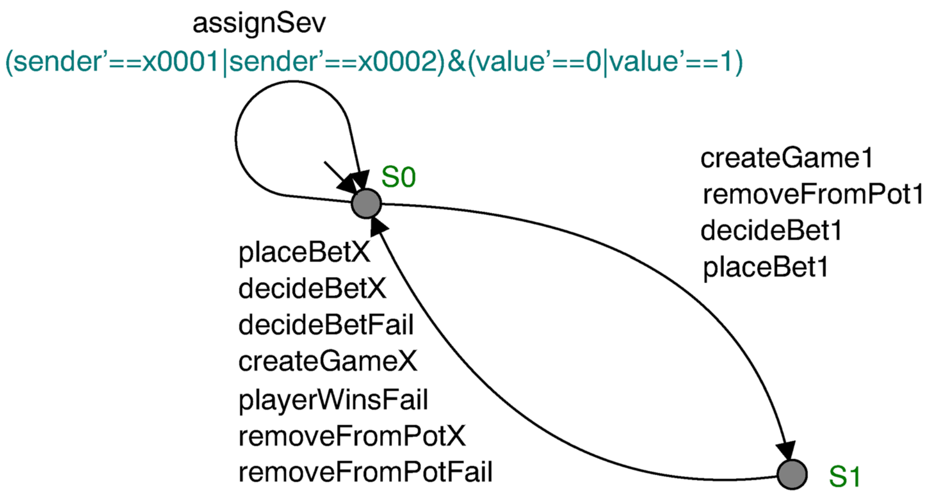


**Fig. 9** EFSM model of the assignment of `sender`for Casino contract

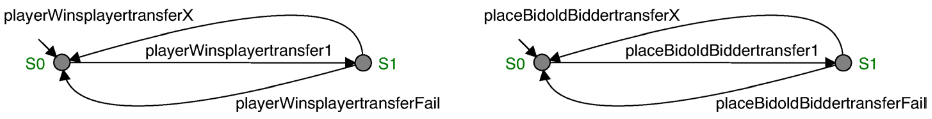

playerWinsplayertransferX
playerWinsplayertransfer1
S0
S1
playerWinsplayertransferFail
placeBidoldBiddertransferX
placeBidoldBiddertransfer1
S0
S1
placeBidoldBiddertransferFail

**Fig. 10** EFSM models of `msg.sender` to the player (left) and to the bidder (right)

`disableCloseAuctionX` model. This is a stronger specification, expressing *assuming the auction is never attempted to be closed, it should always be possible to successfully place a bid*. Effectively `closeAuctionX` removes the entire `closeAuction` EFSM, since disabling the initial event means that the EFSM can never leave its initial (and marked) location.

This again blocks, but now not due to `closeAuctionX`, but due to another issue, here called "the `disableCloseAuctionX` issue".

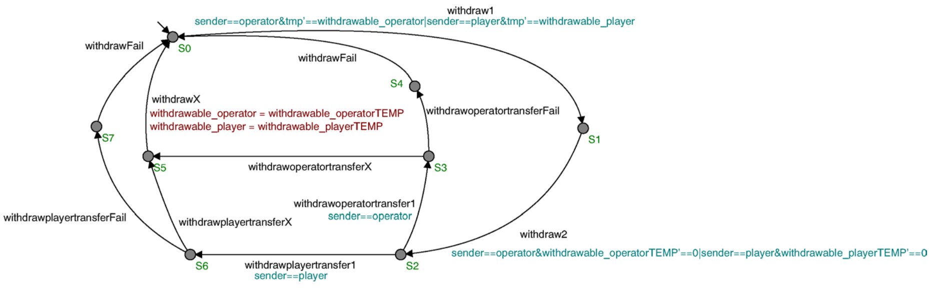

withdraw1
sender==operator&tmp'==withdrawable_operator|sender==player&tmp'==withdrawable_player
withdrawFail
S0
withdrawFail
S4
withdrawX
withdrawable_operator = withdrawable_operatorTEMP
withdrawable_player = withdrawable_playerTEMP
withdrawoperatortransferFail
S1
S7
S5
withdrawoperatortransferX
S3
withdrawoperatortransfer1
sender==operator
withdrawplayertransferFail
withdrawplayertransferX
withdraw2
sender==operator&withdrawable_operatorTEMP'==0|sender==player&withdrawable_playerTEMP'==0
S6
withdrawplayertransfer1
sender==player
S2

**Fig. 11** EFSM model of the `transfer` function in the corrected Casino contract

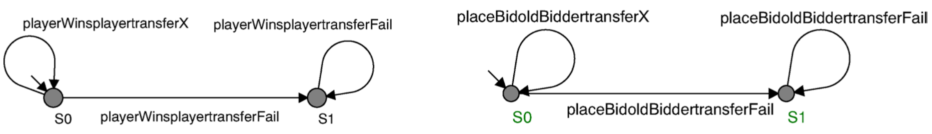

playerWinsplayertransferX
playerWinsplayertransferFail
S0
playerWinsplayertransferFail
S1
placeBidoldBiddertransferX
placeBidoldBiddertransferFail
S0
placeBidoldBiddertransferFail
S1

**Fig. 12** The attacker model for the Casino, left, and the Auction, right

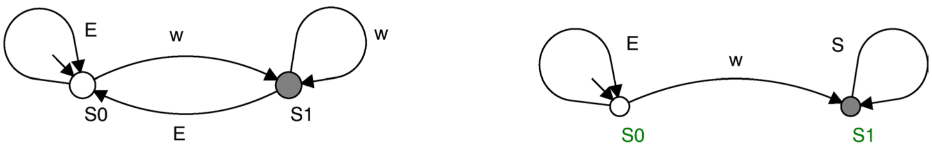


**Fig. 13** Generic versions of progress specifications $K$. These specifications aim to guarantee that the event $w \in \Sigma$ can always eventually occur. $S = \Sigma$, and $E = \Sigma \setminus \{w\}$ is the set of all events except $w$. The left model specifies that $w$ should be able to occur again and again. The right model specifies that it is enough that $w$ occurs once, after which any event (including $w$) may occur

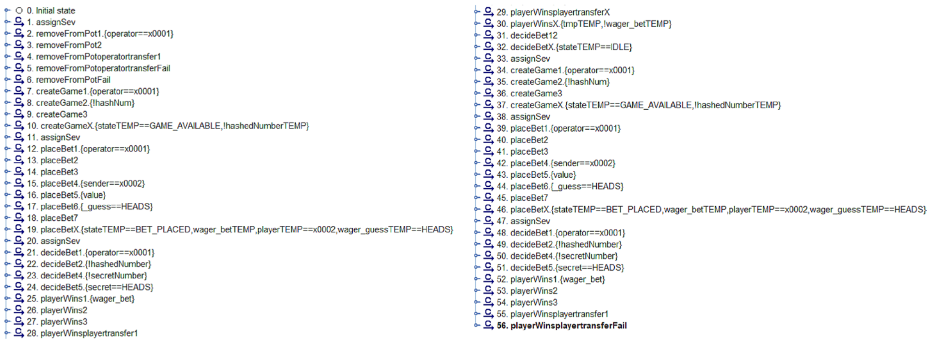

```
0. Initial state
1. assignSev
2. removeFromPot1.{operator==x0001}
3. removeFromPot2
4. removeFromPotoperatortransfer1
5. removeFromPotoperatortransferFail
6. removeFromPotFail
7. createGame1.{operator==x0001}
8. createGame2.{!hashNum}
9. createGame3
10. createGameX.{stateTEMP==GAME_AVAILABLE,!hashedNumberTEMP}
11. assignSev
12. placeBet1.{operator==x0001}
13. placeBet2
14. placeBet3
15. placeBet4.{sender==x0002}
16. placeBet5.{value}
17. placeBet6.{_guess==HEADS}
18. placeBet7
19. placeBetX.{stateTEMP==BET_PLACED,wager_betTEMP,playerTEMP==x0002,wager_guessTEMP==HEADS}
20. assignSev
21. decideBet1.{operator==x0001}
22. decideBet2.{!hashedNumber}
23. decideBet4.{!secretNumber}
24. decideBet5.{secret==HEADS}
25. playerWins1.{wager_bet}
26. playerWins2
27. playerWins3
28. playerWinsplayertransfer1
29. playerWinsplayertransferX
30. playerWinsX.{tmpTEMP,!wager_betTEMP}
31. decideBet12
32. decideBetX.{stateTEMP==IDLE}
33. assignSev
34. createGame1.{operator==x0001}
35. createGame2.{!hashNum}
36. createGame3
37. createGameX.{stateTEMP==GAME_AVAILABLE,!hashedNumberTEMP}
38. assignSev
39. placeBet1.{operator==x0001}
40. placeBet2
41. placeBet3
42. placeBet4.{sender==x0002}
43. placeBet5.{value}
44. placeBet6.{_guess==HEADS}
45. placeBet7
46. placeBetX.{stateTEMP==BET_PLACED,wager_betTEMP,playerTEMP==x0002,wager_guessTEMP==HEADS}
47. assignSev
48. decideBet1.{operator==x0001}
49. decideBet2.{!hashedNumber}
50. decideBet4.{!secretNumber}
51. decideBet5.{secret==HEADS}
52. playerWins1.{wager_bet}
53. playerWins2
54. playerWins3
55. playerWinsplayertransfer1
56. playerWinsplayertransferFail
```

**Fig. 14** The 56-event long blocking trace of the Casino contract with progress specification for `removeFromPotX`

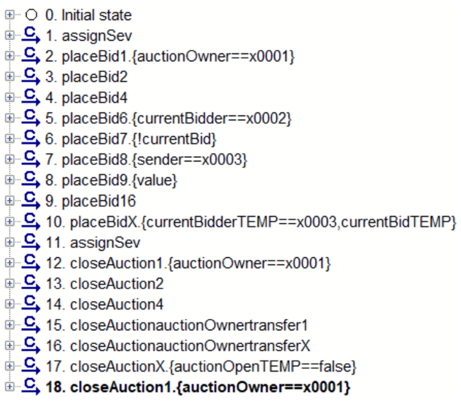
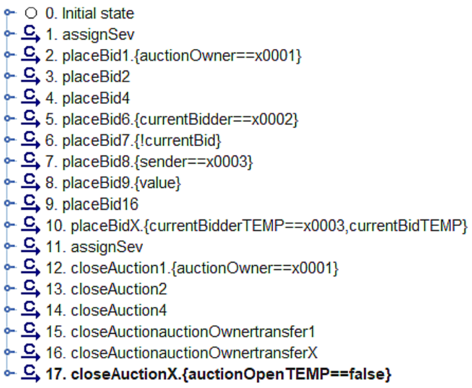

```
0. Initial state
1. assignSev
2. placeBid1.{auctionOwner==x0001}
3. placeBid2
4. placeBid4
5. placeBid6.{currentBidder==x0002}
6. placeBid7.{!currentBid}
7. placeBid8.{sender==x0003}
8. placeBid9.{value}
9. placeBid16
10. placeBidX.{currentBidderTEMP==x0003,currentBidTEMP}
11. assignSev
12. closeAuction1.{auctionOwner==x0001}
13. closeAuction2
14. closeAuction4
15. closeActionauctionOwnertransfer1
16. closeActionauctionOwnertransferX
17. closeAuctionX.{auctionOpenTEMP==false}
18. closeAuction1.{auctionOwner==x0001}
```

```
0. Initial state
1. assignSev
2. placeBid1.{auctionOwner==x0001}
3. placeBid2
4. placeBid4
5. placeBid6.{currentBidder==x0002}
6. placeBid7.{!currentBid}
7. placeBid8.{sender==x0003}
8. placeBid9.{value}
9. placeBid16
10. placeBidX.{currentBidderTEMP==x0003,currentBidTEMP}
11. assignSev
12. closeAuction1.{auctionOwner==x0001}
13. closeAuction2
14. closeAuction4
15. closeActionauctionOwnertransfer1
16. closeActionauctionOwnertransferX
17. closeAuctionX.{auctionOpenTEMP==false}
```

**Fig. 15** Counterexamples for Auction when verifying `IDLE` (left) and `removeFromPot` (right) with the progress specification of Fig. 13, left

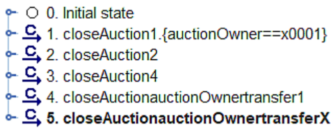
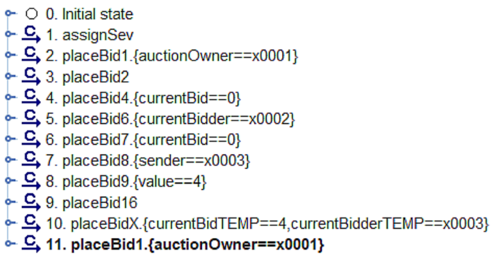

```
0. Initial state
1. closeAuction1.{auctionOwner==x0001}
2. closeAuction2
3. closeAuction4
4. closeActionauctionOwnertransfer1
5. closeActionauctionOwnertransferX
```

```
0. Initial state
1. assignSev
2. placeBid1.{auctionOwner==x0001}
3. placeBid2
4. placeBid4.{currentBid==0}
5. placeBid6.{currentBidder==x0002}
6. placeBid7.{currentBid==0}
7. placeBid8.{sender==x0003}
8. placeBid9.{value==4}
9. placeBid16
10. placeBidX.{currentBidTEMP==4,currentBidderTEMP==x0003}
11. placeBid1.{auctionOwner==x0001}
```

**Fig. 16** Counter example for Auction when verifying `placeBid` with `MAX_BID` (left) and with `currentBidTEMP==4`) (right)

EFSM variables, just as variables in Solidity, are upper bounded, in SUPREMICAby a given (typically small) bound, and in Solidity by default to $2^{256} - 1$ for unsigned integers[7]. Since in the Auction smart contract, the bids are of type `uint`, in both SUPREMICA and Solidity, there is is maximum possible bid, call it `disableCloseAuctionX`. In the Auction code, Fig. 3, line 14, it is checked that a new bid is higher than the currently active bid, and if this is not the case, then the call to `disableCloseAuction1` reverts. So once the current bid has reached `closeAuction1`, no further bids can be placed. The counterexample in Fig. 16, right, shows that when the `closeAuction` completes successfully with a `disableCloseAuction1`, set to 4 in this case (see event 10, `closeAuction`), then on the next call to `closeAuction`, blocking occurs since no bid higher than `MAX_BID` can be given.

Unfortunately, getting around the `placeBidX` issue is not as easy as was the case with `disableCloseAuctionX`, it does not seem possible to add a specification that effectively removes the check for higher bid on line 14 of Fig. 3. This check is in the EFSM model of `disableCloseAuction1`, Fig. 6, modelled as guards on the two transitions out from location `S2`, one guarded by `placeBidX` to `S3`, and the other by `disableCloseAuction1` to `S0`.

[7] Solidity does not support floating point numbers.

What can be done though, is to manually remove these two guards from the EFSM model. Typically, the plant is considered immutable, but in this particular case it seems appropriate to change the plant. If those checks for higher bids are removed, this results in a slightly different Auction, one that accepts any bid, whether higher or lower than the current one. But the question *will the current bid always increase?*, though interesting, is not a topic for this investigation. What is investigated is *assuming that the auction is not closed and that the*`currentBid` *issue does not occur, can a bid always be successfully placed?* Altering the auction to not check for higher bids would remove the `MAX_BID` issue in a way similar to adding the `placeBid` effectively removes the `MAX_BID` EFSM, and this would allow to verify whether it is actually the case that a bid can always be placed or not.

This EFSM system, with the attacker model, the progress specification over `placeBid` , the `MAX_BID`, and the Auction plant with the two guards checking `MAX_BID` removed, verifies to be blocking. The 30-step counterexample is shown in Fig. 17. As can be seen, this blocks when a `closeAuction` to the `placeBid` occurs at Step 30. From there on, the malicious bidder rejects all transfers, and so the system can only cycle around to try the **`value`**`>currentBid` again and again, to always be rejected by the receiver.

In more detail, first a bid of zero occurs at step 10. This is now allowed since the check for always placing a bid higher then the current one has been removed. No `disableCloseAuction1` occurs on this bid, since no old bid different from zero exists, Fig. 3, line 21. Next, at step 20 a bid of one is placed. Again, no `closeAuction` occurs since the `placeBid` is still zero. But the next time a bid is placed, since `disableCloseAuction1` is different from zero, a `currentBid` occurs, which is then rejected and the system enters a blocking cycle from where no marked state can be reached. This shows that a malicious bidder can attack Auction to disrupt it, to the detriment of the auction owner.

## 6.3 The corrected code

To address the vulnerability caused by unsuccessful transfers, functions calling **`transfer`** must be isolated from other functions. Furthermore, a withdrawal pattern where users "pull" funds instead of having the contract "pushing" them is implemented. This "pull instead of push" mechanism is a standard approach to handle the "DoS with Failed Call" vulnerability. Though the malicious player issue of Casino is a liquidity problem, the malicious bidder of

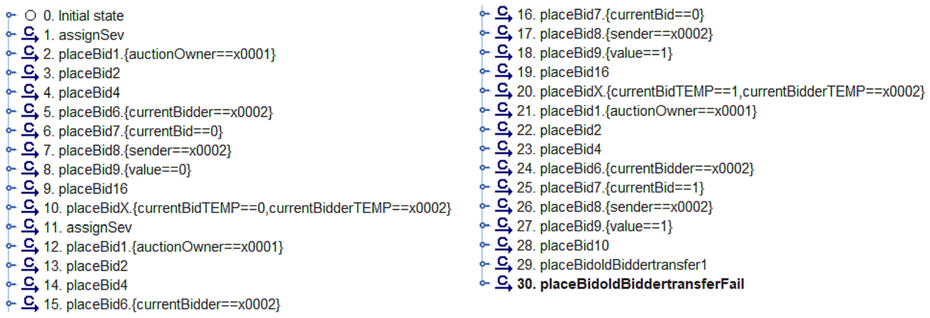

```
0. Initial state
1. assignSev
2. placeBid1.{auctionOwner==x0001}
3. placeBid2
4. placeBid4
5. placeBid6.{currentBidder==x0002}
6. placeBid7.{currentBid==0}
7. placeBid8.{sender==x0002}
8. placeBid9.{value==0}
9. placeBid16
10. placeBidX.{currentBidTEMP==0,currentBidderTEMP==x0002}
11. assignSev
12. placeBid1.{auctionOwner==x0001}
13. placeBid2
14. placeBid4
15. placeBid6.{currentBidder==x0002}
16. placeBid7.{currentBid==0}
17. placeBid8.{sender==x0002}
18. placeBid9.{value==1}
19. placeBid16
20. placeBidX.{currentBidTEMP==1,currentBidderTEMP==x0002}
21. placeBid1.{auctionOwner==x0001}
22. placeBid2
23. placeBid4
24. placeBid6.{currentBidder==x0002}
25. placeBid7.{currentBid==1}
26. placeBid8.{sender==x0002}
27. placeBid9.{value==1}
28. placeBid10
29. placeBidoldBiddertransfer1
30. placeBidoldBiddertransferFail
```

**Fig. 17** 30-step counterexample for Auction when verifying `!(`**`value`**`>currentBid)` with `MAX_BID` and the two guards checking `MAX_BID` removed

```
60   mapping (address => uint) withdrawable;
61
62 function playerWins() private {
63   pot = pot - wager.bet;
64   withdrawable[player] = withdrawable[player] + wager.bet*2;
65   wager.bet = 0;}
66
67 function withdraw() public {
68   uint tmp = withdrawable[msg.sender];
69   withdrawable[msg.sender] = 0;
70   msg.sender.transfer(tmp);}
```

**Fig. 18** The corrected parts of Casino. The other parts of the code of Fig. 2 remain the same

Auction is not. Still, both contracts suffer from the same attack scenario, and the correction is consequently the same in both contracts.

### 6.3.1 The corrected Casino smart contract

To prevent the contract from entering a blocking state, a correction described by Mohajerani et al. (2022) is to replace the push-based mechanism of transferring the winnings with a pull-based mechanism allowing players to withdraw their winnings. In Fig. 18, the balance of `oldBidder` is updated in the **`transfer`** function, line 64, following which **`transfer`** can call `withdraw` on line 67 to have the winnings transferred. This pull mechanism in the corrected contract ensures that progress of the contract is independent of acceptance or rejection of **`transfer`** by `oldBid`.

Non-blocking verification of the automatically converted corrected code together with the attacker model and progress specification does not generate any counterexample, which shows that this DoS vulnerability is no longer present.

### 6.3.2 The corrected Auction smart contract

The `oldBid` and **`transfer`** issues of Auction are legitimate. It is interesting to know that they are there, but there is nothing about them to correct. It should always be possible to successfully close the auction, and it is, but after being closed it is no longer possible to successfully place a bid, and this is just as it should be. Likewise, when the bid reaches **`transfer`**, which is theoretically possible but unrealistic in actual use of the Auction smart contract, no further bids can be placed, which is also as it should be.

That an attacker can disrupt the Auction is a real issue, though. The consequence of an attacker placing a small bid and then rejecting the transfer of that bid to herself is that the auction owner cannot receive higher bids. The only possibility is for the owner to close the auction. The malicious bidder then loses her bid, but apparently this cost was considered by the attacker to be worth it.

The correction of the malicious bidder issue is similar to the `player` solution of Casino. Instead of the contract issuing a `playerWins` pushing the funds to the receiver, the receiver has to pull the funds to themselves by explicitly calling a `withdraw` function. The corrected Auction code is not shown, nor is the automatically converted EFSM model, but non-blocking verification in the same way as described above shows that the corrected code is not susceptible to malicious bidder attacks.

## 7 Conclusion

This paper employs non-blocking verification to identify denial-of-service vulnerabilities in Solidity smart contracts. Specifically, by rejecting the reception of funds, a malicious user can disrupt the intended workflow of a contract to the detriment of other contract owners. Two examples of Solidity smart contracts, a Casino and an Auction, are described. An automatic conversion from a significant fragment of Solidity to a model of interacting EFSMs is described, where the different Solidity function calls and statements are modelled as transitions of the EFSMs. It is shown that non-blocking verification can find DoS issues (and other issues that are not as problematic), using EFSM specifications. The automatically generated EFSM models closely mirror functionalities in the smart contract and issues found in EFSM models were also found to exist in the smart contracts as well. It is also shown by non-blocking verification that the corrected contract does not suffer from the malicious behaviour attacks. This work fills a gap between safety and security, as it allows to investigate code correctness in relation to resilience against malicious attacks. As smart contracts are increasingly relied upon for financial applications, and since these contracts are immutable once stored on the blockchain, finding and correcting such issues is of utmost importance.

Ideas for future work include investigating the role of controllability and synthesis in relation to smart contracts. Since the presented work only concerns non-blocking, the controllability of events is neglected. But since smart contracts are a kind of two- (or multi-) player game, modelling some user's actions as controllable and the other users' moves as uncontrollable allows a refined analysis. For instance, scenarios where attackers damage only themselves could be distinguished from those where attackers damage others.

An interesting work focusing on finding vulnerabilities using Supervisor Control Theory is presented in Matsui and Lafortune (2022), where synthesis of a supervisor is used as an indicator of vulnerabilities being present. The analysis checks violation of safety and non-blocking properties for two communication protocols, namely, the Alternating Bit Protocol (ABP) and Transmission Control Protocol (TCP). However, the direction of our future work investigates using SCT to detect bias in a smart contract where one party is inherently advantageous over other involved parties.

Also, in this work the contracts were corrected manually, and then the corrected code was automatically converted and verified to be non-blocking. A seemingly useful research direction would be to investigate if corrections could be made automatically by synthesizing supervisors that manage to impose resilience on the contracts by avoiding parts of the underlying state space.

**Author Contributions** NP implemented the conversion code and was the main author of that section. WA was the main author of the section on Solidity. MF was the main author of the abstract and the conclusions. All authors were involved with the whole document. All authors reviewed the manuscript in detail and edited all parts.

**Funding** Open access funding provided by Chalmers University of Technology. This work was funded by Chalmers University of Technology.

**Data Availability** The models together with the code for the automatic conversion are available from https://github.com/nishantparekh01/Solidity_to_EFSM/

## Declarations

**Competing interests** The authors declare no competing interests.

## References

Ahrendt W, Bubel R (2020) Functional verification of smart contracts via strong data integrity. In: Margaria T, Steffen B (eds) Leveraging applications of formal methods, verification and validation: applications. Springer International Publishing, Cham, pp 9–24. https://doi.org/10.1007/978-3-030-61467-6_2

Akesson K, Fabian M, Flordal H et al (2006) Supremica – an integrated environment for verification, synthesis and simulation of discrete event systems. In: 2006 8th international workshop on discrete event systems. IEEE, pp 384–385. https://doi.org/10.1109/WODES.2006.382401

Atzei N, Bartoletti M, Cimoli T (2017) A survey of attacks on Ethereum smart contracts (SoK). In: Maffei M, Ryan M (eds) Principles of security and trust. Springer Berlin Heidelberg, Berlin, Heidelberg, pp 164–186. https://doi.org/10.1007/978-3-662-54455-6_8

Bai X, Cheng Z, Duan Z et al (2018) Formal modeling and verification of smart contracts. In: Proceedings of the 2018 7th international conference on software and computer applications. Association for Computing Machinery, New York, NY, USA, ICSCA '18, pp 322–326. https://doi.org/10.1145/3185089.3185138

Baier C, Katoen JP (2008) Principles of model checking. MIT Press

Barbosa H, Barrett C, Brain M et al (2022) cvc5: a versatile and industrial-strength SMT solver. In: Fisman D, Rosu G (eds) Tools and algorithms for the construction and analysis of systems. Springer International Publishing, Cham, pp 415–442. https://doi.org/10.1007/978-3-030-99524-9_24

Barrett C, Sebastiani R, Seshia SA et al (2009) Satisfiability modulo theories. In: Handbook of satisfiability, frontiers in artificial intelligence and applications, vol 185. IOS Press, pp 825–885. https://doi.org/10.3233/978-1-58603-929-5-825, https://ebooks.iospress.nl/publication/5011

Bartoletti M, Ferrando A, Lipparini E et al (2024) Solvent: liquidity verification of smart contracts. In: Kosmatov N, Kovács L (eds) Integrated formal methods. Springer Nature Switzerland, Cham, pp 256–266. https://doi.org/10.1007/978-3-031-76554-4_14

Cavada R, Cimatti A, Dorigatti M et al (2014) The nuXmv symbolic model checker. In: Biere A, Bloem R (eds) Computer aided verification. Springer International Publishing, Cham, pp 334–342. https://doi.org/10.1007/978-3-319-08867-9_22

Chen YL, Lin F (2000) Modeling of discrete event systems using finite state machines with parameters. In: Proceedings of the 2000. IEEE international conference on control applications. Conference Proceedings (Cat. No.00CH37162), pp 941–946. https://doi.org/10.1109/CCA.2000.897591

Cheng KT, Krishnakumar AS (1993) Automatic functional test generation using the extended finite state machine model. In: Proceedings of the 30th ACM/IEEE design automation conference, pp 86–91. https://doi.org/10.1145/157485.164585

Cheng KT, Krishnakumar AS (1996) Automatic generation of functional vectors using the extended finite state machine model. ACM Trans Des Autom Electron Syst 1(1):57–79. https://doi.org/10.1145/225871.225880

CoinGeek (2020) Over $1 million permanently locked in DeFi smart contract. https://coingeek.com/over-1-million-permanently-locked-in-defi-smart-contract/. Accessed 19 Nov 2024

de Moura L, Bjørner N (2008) Z3: an efficient SMT solver. In: Ramakrishnan CR, Rehof J (eds) Tools and algorithms for the construction and analysis of systems. Springer Berlin Heidelberg, Berlin, Heidelberg, pp 337–340. https://doi.org/10.1007/978-3-540-78800-3_24

Fekih R, Lahami M, Jmaiel M et al (2022) Towards model checking approach for smart contract validation in the EIP-1559 Ethereum. In: 46th IEEE annual computers, software, and applications conference, COMPSAC 2022, Los Alamitos, CA, USA, June 27 - July 1, 2022, pp 83–88. https://doi.org/10.1109/COMPSAC54236.2022.00020

Godoy J, Galeotti JP, Garbervetsky D et al (2022) Predicate abstractions for smart contract validation. In: Proceedings of the 25th international conference on model driven engineering languages and systems. association for computing machinery, New York, NY, USA, MODELS ’22, pp 289–299. https://doi.org/10.1145/3550355.3552462

Hoare CAR (1985) Communicating sequential processes. Prentice Hall

Holzmann G (1997) The model checker SPIN. IEEE Trans Software Eng 23(5):279–295. https://doi.org/10.1109/32.588521

ISO/IEC 21778 (2017) Information technology – the JSON data interchange syntax. https://www.iso.org/standard/71616.html. Accessed 23 Nov 2024

Konstantopoulos G (2018) How to secure your smart contracts: 6 solidity vulnerabilities and how to avoid them (part 2). https://medium.com/loom-network/how-to-secure-your-smart-contracts-6-so lidity-vulnerabilities-and-how-to-avoid-them-part-2-730db0aa4834. Accessed 24 Nov 2024

Madl G, Bathen L, Flores G et al (2019) Formal verification of smart contracts using interface automata. In: 2019 IEEE international conference on blockchain (Blockchain), pp 556–563. https://doi.org/10.1109/Blockchain.2019.00081

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

Matsui S, Lafortune S (2022) Synthesis of winning attacks on communication protocols using supervisory control theory: two case studies. Discrete Event Dyn Syst 32(4):573–610. https://doi.org/10.1007/s10626-022-00369-1

Mavridou A, Laszka A (2018) Designing secure Ethereum smart contracts: a finite state machine based approach. In: Meiklejohn S, Sako K (eds) Financial cryptography and data security. Springer Berlin Heidelberg, Berlin, Heidelberg, pp 523–540. https://doi.org/10.1007/978-3-662-58387-6_28

Mohajerani S, Malik R, Fabian M (2016) A framework for compositional nonblocking verification of extended finite-state machines. J Discrete Event Dyn Syst 26(1):33–84. https://doi.org/10.1007/s10626-015-0217-y

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

Mohajerani S, Malik R, Fabian M (2013) Partial unfolding for compositional nonblocking verification of extended finite-state machines. Tech. rep., Chalmers University of Technology, Göteborg, Sweden; also The University of Waikato, Hamilton, New Zealand. https://hdl.handle.net/10289/7140, https://research.chalmers.se/publication/172205

Parekh N, Ahrendt W, Fabian M (2024) Automatic conversion of smart contracts for non-blocking verification. IFAC-PapersOnLine 58(1):282–287. https://doi.org/10.1016/j.ifacol.2024.07.048, 17th IFAC Workshop on Discrete Event Systems WODES 2024

Parizi RM, Amritraj, Dehghantanha A (2018) Smart contract programming languages on blockchains: an empirical evaluation of usability and security. In: Chen S, Wang H, Zhang LJ (eds) Blockchain – ICBC 2018. Springer International Publishing, Cham, pp 75–91. https://doi.org/10.1007/978-3-319-94478-4_6

Ramadge PJG, Wonham WM (1989) The control of discrete event systems. Proc IEEE 77(1):81–98

Richter Vidal F, Ivaki N, Laranjeiro N (2024) OpenSCV: an open hierarchical taxonomy for smart contract vulnerabilities. Empir Softw Eng 29. https://doi.org/10.1007/s10664-024-10446-8

Skoldstam M, Akesson K, Fabian M (2007) Modeling of discrete event systems using finite automata with variables. In: 2007 46th IEEE conference on decision and control, pp 3387–3392. https://doi.org/10.1109/CDC.2007.4434894

Suvorov D, Ulyantsev V (2019) Smart contract design meets state machine synthesis: case studies. arXiv:1906.02906

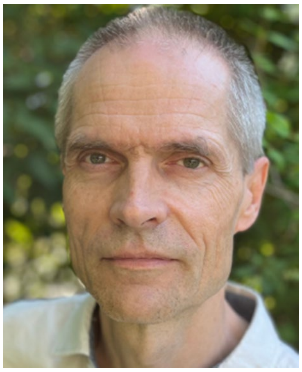
SWC Registry (2020) Smart contract weakness classification (SWC). https://swcregistry.io/docs/SWC-113/. Accessed 21 Nov 2024

VerifyThis (2021) VerifyThis long-term challenge. https://verifythis.github.io/ltc/02casino/. Accessed 21 Nov 2024

Wikipedia (2024) Abstract syntax tree. https://en.wikipedia.org/wiki/Abstract_syntax_tree. Accessed 23 Nov 2024

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

**Nishant Parekh** is a PhD student within the Automation group, at the Department of Electrical Engineering of Chalmers University of Technology, Gothenburg, Sweden. His research focuses on formal verification of smart contracts, applying and advancing formal methods to reason about correctness of smart contracts.



**Wolfgang Ahrendt** is Professor at Chalmers University of Technology, Gothenburg, Sweden, and received his Ph.D. in Computer Science from the University of Karlsruhe, Germany, in 2001. His contributions lie in deductive verification of software, runtime verification, and combinations of static verification with runtime verification and testing. Wolfgang Ahrendt is one of the people behind the software verification approach and system KeY. Recent application areas of his work include automotive software safety, smart contracts and, and AI assisted development of reliable software.

**Martin Fabian** is Full Professor in Automation and Head of the Automation Research group at the Department of Electrical Engineering, Chalmers University of Technology. His research interests include formal methods for automation systems in a broad sense, spanning the fields of Control Engineering and Computer Science. He has authored more than 200 publications, and is co-developer of the formal methods tool Supremica, which implements several state-of-the-art algorithms for supervisory control synthesis.

## Authors and Affiliations

**Nishant Parekh**[1] · **Wolfgang Ahrendt**[2] · **Martin Fabian**[1]

✉ Nishant Parekh
nishantp@chalmers.se

Wolfgang Ahrendt
ahrendt@chalmers.se

Martin Fabian
fabian@chalmers.se

1 Department of Electrical Engineering, Chalmers University of Technology, Gothenburg, Sweden

2 Department of Computer Science, Chalmers University of Technology and University of Gothenburg, Gothenburg, Sweden